CTOS/VM™ CONCEPTS

First Edition (August 1987) 09–00945–01

CONTENTS

LIST OF FIGURES

LIST OF TABLES

RELATED DOCUMENTATION

This manual is one of a set that documents the Convergent family of information processing systems. The set can be grouped as follows:

Introductory

Context Manager/VM Manual
Diagnostics Manual (NGEN)
Executive Manual Installation Guide (NGEN)
Operator's Guide (NGEN)
Quarter-Inch Cartridge Tape for NGEN
Status Codes Manual

Hardware

Color Monitor Manual
Dual Floppy Disk Manual
Ethernet Hardware Manual
Floppy/Hard Disk Manual (see Dual Floppy Disk description)
Graphics Controller Manual: Model GC-001
Graphics Controller Manual: Model GC-003
Hard Disk Upgrades and Expansions Manual
Keyboard Manual
Monochrome Monitors Manual
Mouse Hardware Manual
Multiline Port Expander Manual
PC Emulator Hardware Manual
Power System Manual
Processor Manuals
Quarter-Inch Cartridge Tape Hardware Manual
Voice Processor Manual

Operating Systems

- CTOS Programmer's Guide
- CTOS/VM Concepts Manual
- DISTRIX Operating System Manual
- MS-DOS Manuals

Programming Languages

- iAPX286 Programmer's Reference Manual (Intel)
- 80386 Programmer's Reference Manual (Intel)
- Assembly Language Manual
- BASIC Compiler Manual
- BASIC (Interpreter) Manual (see BASIC Compiler description)
- COBOL Manual (see BASIC Compiler description)
- FORTRAN-86 Reference Manual (see BASIC Compiler description)
- GW-BASIC Operations Manual
- GW-BASIC Reference Manual
- Pascal Reference Manual (see BASIC Compiler description)
- Workstation C Programmer's Guide (see Workstation C Language Manuals description)

Program Development Tools

- COBOL Animator Manual
- Debugger Manual
- Editor Manual
- Font Designer Manual
- Forms Manual
- Graphics Terminal Font Designer
- Linker/Librarian Manual
- Mouse Services Manual
- Raster Font and Icon Designer Manual

System Administration

Generic Print System Programmer's Guide
Printing Guide

Data Management Facilities

CT-DBMS Manual
ISAM Manual
Sort/Merge Manual

Office Automation

Graphics
Graphics Programmer's Guide

Voice
Voice/Data Services Manual

Communications

Asynchronous Terminal Emulator Manual
CT-Net Reference Manual
Modem Server Reference Manual

Other

80286 Architecture

The following section outlines the contents of these manuals.

INTRODUCTORY

The Context Manager/VM Manual describes and teaches the use of the Context Manager/VM, which allows the user to run applications concurrently and transfer from one application to another. It also describes the interaction between the Context Manager and the Window Services, in which the user can simultaneously view several applications on the screen.

The NGEN Diagnostics Manual outlines the tests used to verify proper operation of the modules of a workstation. The manual describes tests for individual modules, along with bootstrap procedures and customization programs.

The Executive Manual describes the interactive command interpreter that interacts with the CTOS and CTOS/VM operating systems. The manual is both a user's guide and a reference to the available commands. It addresses command execution, file management and protection, and program invocation. The manual also provides descriptions and details about parameter fields for Executive commands.

The NGEN Installation Guide describes procedures for unpacking, assembling, cabling, and powering up an NGEN workstation.

The NGEN Operator's Guide describes the operator controls, use of the floppy disk drives, verification of workstation operations, and use of software release notices.

The Quarter-Inch Cartridge Tape for NGEN Manual explains the use of quarter-inch cartridge tape software, primarily for backing up and restoring hard disks. The manual also describes the use of the Quarter-Inch Tape maintenance utilities and the Tape Copy utility.

The Status Codes Manual contains a complete list of all the status codes that can be generated by a CTOS workstation or a Shared Resource Processor (SRP), including bootstrap ROM error codes and CTOS initialization codes. The manual also describes and interprets crash status codes.

HARDWARE

The Color Monitor Manual describes the operation and connections of the 15-inch Color Monitor used with the NGEN workstation.

The Dual Floppy Disk Manual and the Floppy/Hard Disk Manual describe the architecture and theory of operation for the respective NGEN disk modules. They discuss the applicable disk drives and controllers, and contain the applicable OEM disk drive manuals.

The Ethernet Hardware Manual describes the Ethernet Module in terms of its software and hardware interfaces to the NGEN workstation. The manual also provides detailed information on installing the Ethernet Module into an NGEN configuration, and on various networking and cabling options.

The Graphics Controller Manual: Model GC-001 describes the architecture, theory of operation, and external interfaces for model GC-001 of the Graphics Controller Module, which accommodates either a Monochrome or Color Monitor.

The Graphics Controller Manual: Model GC-003 gives instructions for installing Model GC-003 of the Graphics Controller Module. The manual also provides the functional description and theory of operation for the module, and describes software interfaces and external interfaces.

The Hard Disk Upgrades and Expansions Manual describes the architecture and theory of operation of the Disk Upgrade and Disk Expansion Modules.

The Keyboard Manual describes the architecture, theory of operation, and external interfaces for the NGEN keyboard.

The Monochrome Monitors Manual describes the operation and connections of the Standard and High Resolution Monochrome Monitors used with the NGEN workstation.

The Mouse Hardware Manual describes the architecture, theory of operation, and external interfaces for the NGEN mouse.

The Multiline Port Expander Manual describes the architecture, theory of operation, and external interfaces for the NGEN Multiline Port Expander Module.

The PC Emulator Hardware Manual describes the PC Emulator hardware at a functional block and component level. The manual also describes the PC Emulator Module register set and explains how to attach the module onto the workstation's X-Bus.

The Power System Manual describes the operation and connections for the 36-Volt Power Supply and the dc/dc converters used with the NGEN workstation.

The Workstation C Language Manuals (includes the Workstation C Programmer's Guide and C Programming Language Manual) describe the C programming language, enhancements to the language, library functions, and operating instructions for running Workstation C on the CTOS and DISTRIX operating systems. The manuals also provide troubleshooting information.

PROGRAM DEVELOPMENT TOOLS

The COBOL Animator Manual describes the COBOL Animator, a debugger that allows the user to interact directly with the COBOL source code during program execution.

The Debugger Manual describes the Debugger, which is designed for use at the symbolic instruction level. It can be used in debugging C, FORTRAN, Pascal, and assembly language programs. (COBOL and BASIC, in contrast, are more conveniently debugged using special facilities described in their respective manuals.)

The Editor Manual describes the test editor that interacts with the CTOS and CTOS/VM operating systems.

The Font Designer Manual describes how to design a new character set for display on the workstation monitor. The Font Designer produces vector fonts, as opposed to the raster fonts that are produced with the Raster and Icon Font Designer.

The Forms Manual describes the Forms facility that includes the Forms Editor, which is used to interactively design and edit forms, and the Forms run time, which is called from an application program to display forms and accept user input.

The Graphics Terminal Font Designer Manual describes how to use the Graphics Terminal Font Designer package to create, edit, and load fonts.

The Linker/Librarian Manual describes both the Linker, which links together separately compiled object files, and the Librarian, which builds and manages libraries of object modules.

The Mouse Services Manual describes the Mouse Server and the object module library for applications programmers. It also includes a short description of end-user commands.

The Raster Font and Icon Designer Manual describes the interactive utility for designing new fonts (character sets) for the video display.

SYSTEM ADMINISTRATION

The Generic Print System Programmer's Guide is a guide for writing applications that use the Generic Print System or the Generic Print Access Method. It addresses applications that transfer data to the printer as well as more sophisticated applications with status checking and printer control. The manual includes descriptions of the Generic Print System and Generic Print Access Method procedural interfaces.

The Printing Guide provides information on how to install any supported printing device on your standalone workstation or a workstation within a cluster. It describes the Print Manager, which is the interface to the Generic Print System, and how to use the Print Manager to control and monitor the status of printing devices. Printer troubleshooting is also discussed.

The Processor Manuals describe the respective Processor Modules. Each manual in this two-volume set covers one processor module and details the architecture and theory of operation of the printed circuit boards, external interfaces, and memory expansion, as well as X-Bus specifications.

The Quarter-Inch Cartridge Tape Hardware Manual describes the architecture, theory of operation, and hardware specifications for the Quarter-Inch Cartridge Tape Module.

The Voice Processor Manual describes the architecture, theory of operation, external interfaces, and hardware specifications for the Voice Processor Module.

OPERATING SYSTEMS

The CTOS Programmer's Guide is a reference guide for programming under the CTOS operating system. It describes CTOS programming practices and introduces the system to programmers who are using it for the first time.

The CTOS/VM Concepts Manual together with the CTOS/VM Reference Manual, describes the CTOS/VM operating system. The CTOS/VM Concepts Manual introduces the CTOS/VM operating system to the programmer by presenting concepts in a basic-to-advanced order. Included among the concepts in this manual are management of processes, messages, memory, exchanges, video, keyboard, files, disks, printers, communications, tape, and timers. CTOS/VM operations pertaining to each concept are described briefly at the end of each chapter. The manual also explains how to use the CTOS/VM operations and provides information on the administrative aspects of the operating system.

The DISTRIX Operating System Manual describes DISTRIX, an operating system derived from the UNIX System V operating system. It describes commands, application programs, system calls, subroutines, special files, file formats, games, miscellaneous facilities, and system maintenance procedures.

The MS-DOS Manuals describe the single-user operating systems originally designed for the 8086-based personal computer systems.

PROGRAMMING LANGUAGES

The IAPX286 Programmer's Reference Manual (Intel) describes the architecture of the Intel 80286 microprocessor.

The 80386 Programmer's Reference Manual describes the 80386 32-bit microprocessor.

The Assembly Language Manual describes the machine architecture of the associated CPU, the assembly language, instruction set, and programming at the symbolic instruction level.

The BASIC Compiler and BASIC (Interpreter), COBOL, FORTRAN, FORTRAN-86 Reference, and Pascal Reference manuals describe the system's programming languages. Each manual specifies both the language itself and operating instructions for that language.

The GW-BASIC Manuals describe the version of BASIC that runs on the MS-DOS operating system.

DATA MANAGEMENT FACILITIES

The CT-DBMS Manual describes the CT-DBMS database management system, which consists of a data manipulation language for accessing and manipulating the database, as well as utilities for administering database activities such as maintenance, backup and recovery, and status reporting.

The ISAM Manual describes both the single-user and the multiuser Indexed Sequential Access Method (ISAM). It specifies the procedural interfaces (and how to call them from various languages) and the utilities.

The Sort/Merge Manual describes the Sort and Merge utilities that run as a subsystem invoked at the Executive command level, and the Sort/Merge object modules that can be called from an application program.

OFFICE AUTOMATION

GRAPHICS

The Graphics Programmer's Guide describes the graphics library procedures for applications and systems programmers. In addition to an alphabetic reference section describing all graphics procedures, the manual includes annotated program examples that explain important graphics concepts and show typical sequences of procedure calls.

VOICE

The Voice/Data Services Manual describes the Voice Data Services, a device driver that provides a request and procedural interface between applications software and the Voice Processor Module.

COMMUNICATIONS

The Asynchronous Terminal Emulator Manual describes the asynchronous terminal emulator.

The CT-Net Reference Manual provides information for system administrators on installing, configuring, maintaining, and monitoring their local nodes, and on communicating with remote nodes.

The Modem Server Reference Manual describes the configuration, installation, maintenance, modems, and programmatic interface of the Modem Server. This system service controls up to six asynchronous communications lines, accommodating up to four clients per line. The Modem Server is used with CT-Net, CT-MAIL, and the Multimode Terminal Emulator (MTE); it can also be used with user-defined communications agents.

OTHER

The 80286 Architecture by Stephen P. Morse and Douglas J. Albert describes the architecture of the Intel 80286 microprocessor (John Wiley & Sons, Inc., New York, N.Y.).

1 INTRODUCTION

WHAT IS CTOS/VM?

CTOS/VM is Convergent Technologies' operating system with virtual machine (VM) capability. It is designed for microprocessors that support protected mode operation. Currently, these microprocessors are the 80286 and 80386 (available on workstations only). This manual also describes the real mode operating systems based on the 80186 microprocessor (available on Shared Resource Processors and workstations).

WHAT DOES CTOS/VM OFFER?

CTOS/VM offers a CTOS software foundation shared by all Convergent proprietary operating systems. CTOS features include the following:

- multiprogramming
- multitasking
- event-driven, priority-ordered process scheduling
- messaged-based operation
- nationalization

Additionally, CTOS/VM offers the following enhancements:

- protected mode operation
- Real Mode Operating System (RMOS)
- virtual 8086 mode
- variable partitions with code sharing capability

CTOS FEATURES

MULTIPROGRAMMING

Multiprogramming is the ability to run more than one program in memory at the same time. Multiprogramming supports the independent invocation and scheduling of multiple processes. Additionally, it supports concurrent I/O and multiple processor implementations.

MULTITASKING

Multitasking is the ability for any program to have more than one process (thread of execution). (Note that in this manual, multitasking is called multiprocessing.)

The Executive, for example, consists of two processes: one accepts your keystrokes, while a second displays the time of day.

EVENT-DRIVEN, PRIORITY-ORDERED PROCESS SCHEDULING

Each process (thread of execution) is assigned a priority and is scheduled for execution based on that priority. The Kernel scheduler uses this priority scheme to provide efficient scheduling. In the Executive, for example, the clock process runs at a higher priority than the process accepting user keystrokes.

Scheduling is driven by system events. Whenever an event, such as the completion of an I/O operation, makes a higher priority process eligible for execution, that process is scheduled to execute immediately.

This scheduling technique is called event-driven, priority scheduling. It simplifies scheduling and provides faster response times than scheduling techniques that are entirely time-based.

MESSAGE-BASED OPERATION

CTOS/VM is message-based. Programs, as well as the operating system, consist of processes, each managing various resources and communicating by means of messages. Overall execution occurs because messages requesting services are dispatched and processed.

Message-based operation permits the dynamic installation/deinstallation of system services without regenerating the system or altering operating system code. Dynamic installation/deinstallation provides the convenience of adding services, such as printing, queue management, the mouse, or windowing support, at any time. Services can be Convergent-provided or user-written.

Unlike subroutine calls, messages can be filtered and redirected across networks, simplifying the development of distributed and multiprocessing applications.

NATIONALIZATION

Native language support (NLS) provides a set of utilities, run time libraries, and data structures that can be used for the easy portation of software to run in various languages.

CTOS/VM ENHANCEMENTS

PROTECTED MODE OPERATION

Protected mode operation provides the advantages of extended memory and protection. Programs can reference memory extending beyond the first megabyte up to the maximum allowed by the processor and hardware. Protected mode system structures place limitations on the memory programs can access, thereby preventing programs from overwriting code or referencing static memory allocated to other programs.

REAL MODE OPERATING SYSTEM (RMOS)

Real mode operating system (RMOS) support allows you to run any existing real mode application program on a protected mode operating system without modifying code, recompiling, or relinking. The real mode program has virtual machine capability. This means that it appears to be executing autonomously in a multiprogramming environment.

VIRTUAL 8086 MODE

Virtual 8086 mode is a virtual machine implementation that supports the execution of multiple operating systems, such as MS-DOS, in a multiprogramming environment. In virtual 8086 mode, a region of memory is allocated and assigned the operating system characteristics of an 8086 microprocessor. (For details, see the 80386 Programmer's Reference Manual.) Each memory region, thus, provides a 1 megabyte address space within which a program can execute. Concurrently, application programs can execute in real mode (RMOS) or in protected mode in other memory regions.

VARIABLE PARTITIONS WITH CODE SHARING CAPABILITY

Variable partitions and code sharing provide efficient memory usage. A variable partition can change in size dynamically to meet the requirements of the program currently executing. The code of the executing program can be shared by the same type of program in a different variable partition.

HOW THE OPERATING SYSTEM IS STRUCTURED

The basic components of the operating system are

- the Kernel
- system service processes
- system-common procedures
- object module procedures
- device and interrupt handlers

The Kernel, the most primitive yet most powerful operating system component, provides process management and message-based process communication facilities.

System service processes manage system resources, such as files and memory.

The operating system's device handlers and interrupt handlers are accessed indirectly through the convenient interfaces provided by the system service processes.

System-common procedures are procedures that perform some common system functions. The Video Access Method is a collection of system-common procedures.

Object module procedures are procedures that are supplied as part of an object module library file and can be linked with the application program. They are not part of the System Image itself. The Sequential Access Method (SAM) is a collection of object module procedures.

USING THIS MANUAL

This manual guides you through an overview of how the operating system works. This manual and the CTOS/VM Reference Manual are a set that describes the CTOS/VM operating system.

The CTOS/VM Reference Manual contains a description of each operation in the System Image and in the standard operating system library, CTOS.lib. Use the CTOS/VM Reference Manual as a programming guide. You can use CTOS/VM with several different programming languages.

ORGANIZATION

This manual is organized as follows:

- Chapter 1 introduces you to the CTOS/VM operating system: it highlights those features that are unique to the operating system and summarizes this manual's organization.

- Chapter 2 provides an overview of the operating system concepts described in detail in later chapters.

- Chapter 3 introduces you to the various types of CTOS/VM operations and explains ways you can use these operations in your programs.

- Chapter 4 describes program management, which consists of those operations used by a program to self-load into memory, to self-exit from memory, and to handle error conditions. This subject is presented at a more advanced level in Chapter 32, "Program and Partition Management," which describes how a partition managing program performs comparable operations to manage several programs in memory at once.

- Chapter 5 presents parameter management, a method of passing information from one program to its successor within the same partition of memory.

- Chapters 6 through 23 describe how I/O can be performed to devices, such as disks, video, tape, and communication channels.

- Chapters 24 through 34 cover operating system theory. These chapters describe such subjects as memory and partition management, system services, and how the operating system uses interprocess communication (IPC) and inter-CPU communication (IPC). These chapters also present more advanced programming concepts.

- Chapter 35 describes how queues are managed.
- Chapters 36 and 37 are related to the I/O chapters (Chapter 6 through 23) but cover the more advanced concepts of interrupts and X-Bus management.
- Chapters 38 through 40 are dedicated to the administrative aspects of the operating system. As a programmer, you may not be involved in customizing your system. You may find it beneficial, however, to nationalize your programs so that they can be used on operating systems in other countries.
- Appendix A describes spooler management.

CHAPTER ORDERING

Figure 1-1 gives you a visual overview of the organization of this manual and shows the relationships of the operating system concepts. Each box contains a chapter title and its corresponding chapter number.

In Figure 1-1, the chapters are prioritized in a need-to-know order. Program management, parameters, and I/O, for example, are programming concepts that you need to know early to get started with programming. These chapters are among the first presented in the manual.

Later chapters are located toward the right and lower-right regions in the figure. These chapters contain more advanced concepts and operating system theory.

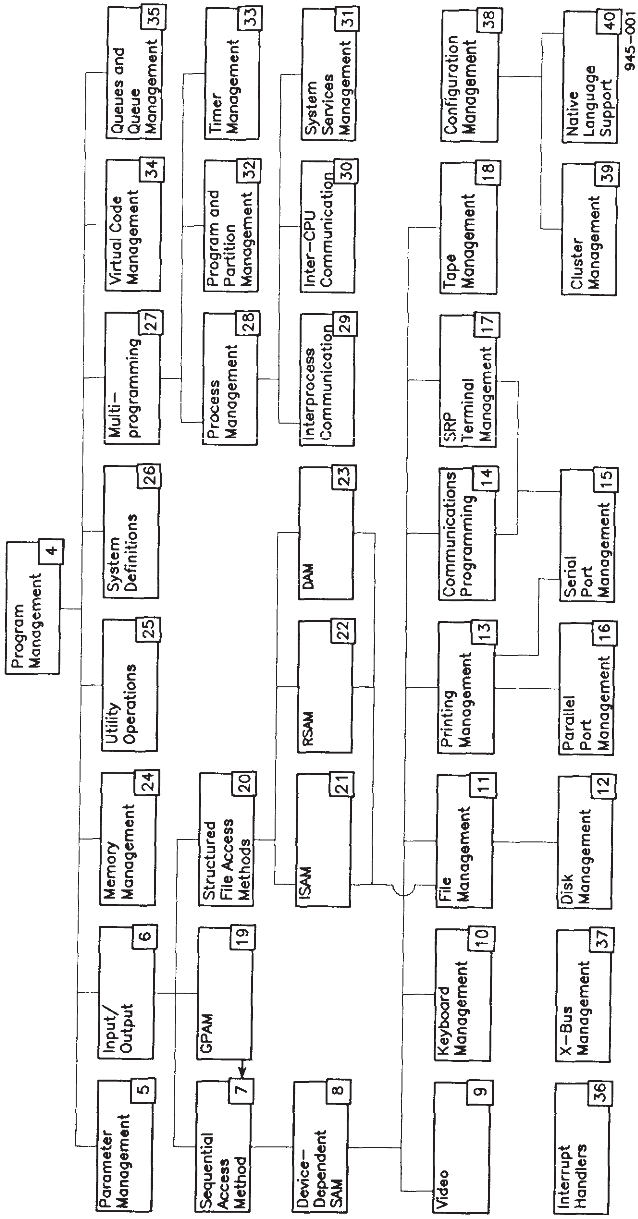


Figure 1-1. Relationships of Operating System Concepts

The chapter boxes associated with multiprogramming (Chapter 27), for example, provide advanced concepts. You do not need to understand these concepts right away. You can use the operations in the CTOS/VM Reference Manual without ever knowing the concept of messages and message passing, for example, which is the basis of IPC (Chapter 29).

As you become more familiar with the operating system, you can take advantage of the more advanced programming techniques.

Note that the chapter boxes associated with configuration management (Chapter 38) are not connected to the other chapter boxes. This is because you may never be involved in the administrative activities described in these chapters. If you are a system administrator, you have reason to investigate this area.

Use Figure 1-1 as a quick reference guide as you are getting acquainted with the operating system. You will find these chapters (and the list of operations described at the end of each) presented again in an overview figure at the beginning of the CTOS/VM Reference Manual.

2 OVERVIEW OF OPERATING SYSTEM CONCEPTS

This chapter is an overview of operating system concepts. These concepts are described in detail in later chapters of this manual.

OPERATING SYSTEM STRUCTURE

PROCESS

A process is an independent thread of execution for a program. It carries with it the context (that is, the processor registers) necessary to that thread. One or more processes are created each time a program is scheduled for execution.

The operating system assigns each process a priority to schedule its execution appropriately: priorities range from 1 (highest) to 255 (lowest/ null).

System service processes are processes that manage system resources. All processes, including system service processes, are scheduled for execution in the same way based on their assigned priority.

KERNEL

The Kernel is the most primitive yet most powerful component of the operating system. It provides

- event-driven priority scheduling
- Interprocess Communication (IPC)
- Inter-CPU Communication (ICC)

Event-Driven Priority Scheduling

To meet the need for high performance, the operating system Kernel provides efficient event-driven priority scheduling.

Each process is assigned one of 255 priorities and is scheduled for execution based on that priority. Whenever an event, such as the completion of an I/O operation, makes a higher priority process eligible for execution, rescheduling occurs immediately. This results in a more responsive system than scheduling techniques that are entirely time-based.

Interprocess Communication (IPC)

The Kernel's IPC primitives, such as Request and Wait (or Check), are the primary building blocks for synchronizing process execution and transmitting information between processes.

Messages and Exchanges. A process can send a message, wait for a message, or poll (check) for a message. When a process waits for a message, its execution is suspended until a message is sent to it, thus allowing processes to synchronize execution. A process can also check to determine if a message is available without suspending its execution.

The operating system is message-based. When a process sends a message, it actually sends the message to an exchange rather than directly to another process. Exchanges function as message centers where processes send messages or processes wait or check for messages. Within a single processor, overhead is minimized, because only the address of the message is moved, not the message itself.

A single process can serve several exchanges, in which case it can select one of several kinds of messages to process next. This feature can be used to set priorities for the work the process is to perform.

Also, several processes can serve the same exchange, thereby sharing the processing of a single kind of message.

System Service Processes. The operating system includes a number of system service processes. A system service process receives IPC messages to request the performance of its services. Examples of operating system services include opening or closing disk files, sending output to a printing device, or accepting keyboard input. A process requesting a system service is a client process. Any process, including another system service process, can be a client. The use of system service processes and the formalized interface provided by IPC results in a highly modular environment that increases reliability and flexibility.

System services can be linked-in system services in the operating system. The file management system and the keyboard services are examples.

A system service also can be dynamically installable. The Queue Manager and CT-Mail are examples. Once installed, a dynamically installable system service is indistinguishable in operation from a linked-in service.

Each of the functions provided by the system service can be accessed by a procedural call from a high-level language, such as Pascal or C, as well as from assembly language. The request procedural interface masks all the complexities of using IPC: it automatically uses a default response exchange and builds the request block message on the stack of the client process.

Kernel primitives also can be called directly. This allows an increased degree of concurrency between multiple I/O operations and computation. The calling process, for example, can perform calculations while it is waiting for other data to be written to a disk file.

Filters. You can customize the function of a system service by writing a filter for that service.

A filter intercepts messages destined for another system service. It may modify the effect of the messages, but it does not modify either the calling process or the system service for which the messages were intended.

Inter-CPU Communication (ICC)

The ICC facility provides for communication between CPUs among the different processor boards on the SRP. ICC is an extension of IPC.

If the requested system service is on the same SRP processor board as the client process, the Kernel uses IPC. If, however, the service is on a different processor board, the Kernel uses ICC. ICC passes request and response messages between processor boards.

The SRP is compatible with the workstations at the request level. Whether your program runs on an SRP or on a workstation, your program can access system services in the same way (that is, either by using the request procedural interface or by calling the Kernel primitives).

CONFIGURABLE COMMAND INTERPRETER

Interaction with the workstation operator is a function of the Executive, not the operating system. This allows you to choose how to use the screen and the keyboard.

The Executive is an interactive command interpreter providing a user interface that includes a HELP facility, command files, and the interactive addition of new commands. The Executive is also a normal application-level program.

You can easily replace the Executive with a customized command interpreter of your own design. (For details on the Executive, see the Executive Manual.)

OTHER OPERATING SYSTEM FEATURES

File System Management

The file system management provides a hierarchical organization by node, volume, directory, and file. A volume (formatted disk) is automatically recognized when you place it online (mount it) . A file can be dynamically expanded or contracted as long as it fits on one disk (1 gigabyte), and it can be protected by password (optionally encrypted) and protection level number. Concurrent file access is controlled by read (shared), peek (shared), and modify (exclusive) access modes.

While providing convenience and security, the file management system supplies you with the full throughput capability of the disk hardware. This includes reading or writing any 512 byte sector of any open file with one disk access, reading or writing up to 65K bytes (127 sectors) of any open file with one disk access, overlapping I/O with process execution, and optimizing disk arm scheduling.

The duplication of critical volume control structures protects the integrity of disk file data against hardware malfunction. Two Volume Home Blocks can be created for each volume. In addition, two File Header Blocks can be created for each file on a volume.

In the Executive, you can use the Backup Volume command to recover a file if either of its redundant File Header Blocks is valid. The **IVolume** command can be used to suppress the duplication of volume control structures. (This reduces reliability, however, and is not recommended.)

Device Handlers

The operating system is designed to accommodate user-written device handlers. A device handler can be part of the application program, or it can be a system service. The Kernel can either save process context, allowing the use of handlers written in high-level languages, or an assembly language interrupt handler can receive the interrupt directly from the hardware. IPC provides an efficient, yet formal, interface from interrupt handler to device handler and from device handler to application program.

DISTRIBUTED ENVIRONMENT AND CLUSTERING

LOCAL RESOURCE-SHARING NETWORKS (CLUSTERS)

The operating system provides support for local resource-sharing networks (clusters), as well as for standalone workstations.

A cluster configuration consists of cluster workstations connected to a master. The master can be a master workstation or the SRP. Essentially the same operating system executes in each cluster workstation as in the master workstation (or in the combined processors of the SRP). The master provides resources, such as file system management and queue management, for all workstations in the cluster. Concurrently, a master workstation can support its own interactive application program processing.

In the cluster configuration, the IPC facility is extended to provide transparent access to system services that execute in the master. While some services (such as queue management, 3270 emulator, and database management) migrate to the master, others (such as video management and keyboard management) remain at the cluster workstation. A cluster workstation with its own file system can service file requests locally as well as send file requests to the master.

One high-speed, RS-422 channel is standard on each workstation. In cluster configurations connected to a master workstation, the master and all of the workstations connected to it use this channel for intercluster communications. For large clusters with an SRP master, multiple RS-422 channels are provided.

CT-NET NETWORK

The CT-Net network extends the operating system resource-sharing capability. CT-Net provides for sharing resources (such as the file system, CT-ISAM, X.25 Network Gateway, and printing services) between workstations in clusters that are connected by communications lines over long distances.

OPERATING SYSTEM TYPES

Operating systems are available for workstations and for the SRP.

Workstation operating systems are of the following types:

- standalone workstation (Stnd)
- master workstation (Mstr)
- cluster workstation (Clstr)
- cluster workstation with local file system (ClstrLfs)

An SRP operating system can contain the following processors:

- Cluster Processor (CP)
- Data Processor (DP), which is a Storage Processor (SP) and Storage Controller (SC) combination
- File Processor (FP)
- Storage Processor (SP)
- Terminal Processor (TP)

WORKSTATION OPERATING SYSTEMS

Table 2-1 summarizes features available on each workstation operating system.

Table 2-1
WORKSTATION OPERATING SYSTEM FEATURES

Operating System	Cluster Agent	Master Agent	File System
Stnd			X
Mstr		X	X
Clstr	X		
ClstrLfs	X		X

The differences between each workstation operating system are a function of the services each has to offer.

The cluster workstation operating system differs from the standalone workstation in the (optional) exclusion of the file management service and the disk handler, and the inclusion of the Cluster Agent. The cluster workstation with a local file system includes a file management service.

The master workstation operating system differs from the standalone only in its inclusion of the Master Agent. The master workstation can provide file services for the entire cluster configuration.

SRP OPERATING SYSTEMS

An SRP operating system comprises several different processor boards. Each processor board contains a CTOS Kernel and memory, and generally provides a subset of the services offered by a workstation operating system. Services provided by individual processor boards can be shared among all others. Interboard communication is achieved by means of a high-speed bus using the ICC facility. Together, the processor boards function as a unified operating system.

In general, an SRP operating system consists of at least one FP or DP and one CP.

Table 2-2 summarizes features provided by each SRP processor.

Table 2-2
SRP PROCESSOR BOARD FEATURES

SRP Processor	Master RS-422 Agent	RS-232-C	File System	Half-Inch Tape
FP			X	
DP (SP+SC)			X	X
CP	X	X		
TP		X		
SP				X

The FP as well as the DP provide file management services, differing only in the type of hardware upon which the service is performed. The FP services hard disks, whereas the DP services the SMD class of disk drives. The DP, in addition, supports half-inch tape. Note that the SP handles half-inch tape exclusively.

The CP and TP contain peripheral ports for cluster and network communications. The CP provides a Master Agent to transport messages over RS-422 channels (to locally clustered workstations) and an RS-232-C communications service to support asynchronous terminals and communications media. The TP specializes in RS-232-C communications services only.

PROGRAM AND PARTITION

An executable program can consist of code, data, and one or more processes in a memory partition.

NOTE: The term partition, as used in this manual, shows the bounds of a program while that program is in memory. Actual partition sizes and locations vary with each operating system. In addition, partition contents (protected mode operating systems) are not contiguous in physical memory, and portions (such as code) may be shared between partitions. In previous operating system versions, a partition actually was a static memory cell into which various programs were loaded.

A program is loaded into a memory partition from a disk-resident file or run file. Run files are created by compiling and/or assembling source language modules into object modules and linking the object modules together into code and data segments.

When a currently active program such as the Executive requests it to do so, the operating system reads the run file into memory, relocates intersegment references, and schedules the program for execution.

NOTE: This manual generally describes a logical model of the operating system rather than a particular implementation. In certain cases, however, such as in the description of "System Memory Organization" that follows, the implementation is indicated to point out significant feature differences. (For details, see the Release Notice for your version of the operating system.)

SYSTEM MEMORY ORGANIZATION

System memory consists of two types of partitions:

- System partitions: A system partition can contain the operating system or a dynamically installed system service.

- Application partitions: An application partition can contain an application program.

When a system is initiated, the operating system is loaded into system partitions at the low and high address ends of memory. (See Figure 2-1.)

Operating system data is loaded at low and high addresses.

- Data at the low address end includes the system structures and the Interrupt Vector Table (real mode only).

- Data at the high address end includes the loadable request files and the NLS tables.

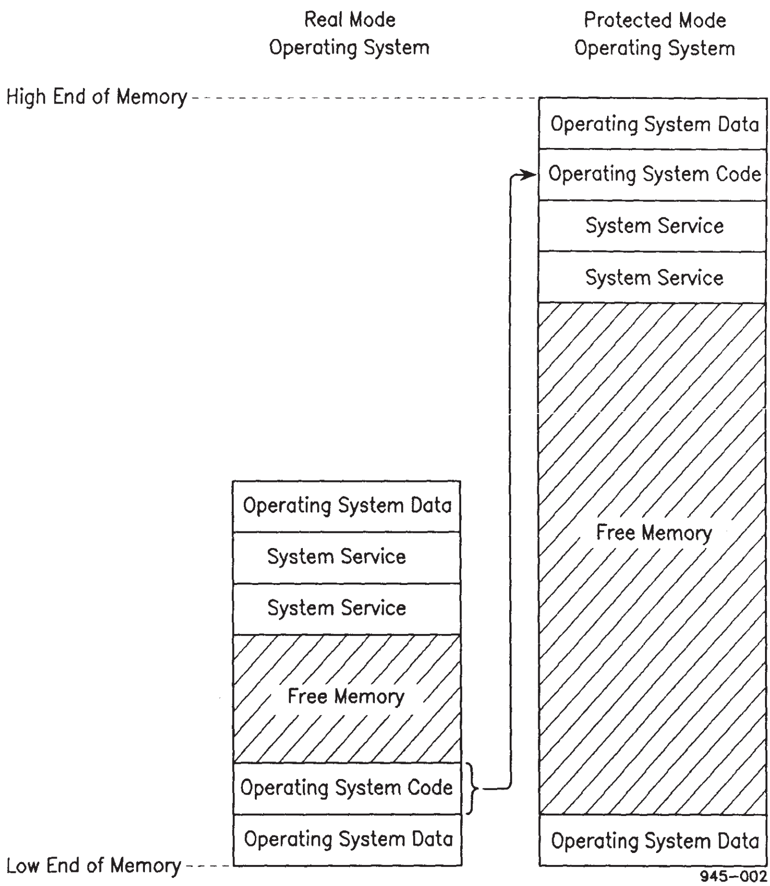


Figure 2-1. Memory Organization

Operating system code is loaded at the low address end for real mode and at the high address end for protected mode. Code includes the System Image and the file system, if present. For protected mode, a resident Debugger optionally can be loaded as part of the code.

As shown in Figure 2-1, most of the operating system is loaded at the high end of memory for protected mode. This is one of the advantages of protected mode: it frees more memory for application programs to run in the first megabyte.

In either mode, dynamically installed system services are loaded into system partitions located at the high address end of memory.

The remaining memory at initialization is defined as free memory.

To bring an application program into memory, the operating system creates a new application partition in free memory into which it loads the program. The partition is placed at the high address end of free memory. (See Figure 2-2.)

PARTITION MANAGING PROGRAMS

A partition managing program is a program that can create new application partitions and load programs into them. The Context Manager is such an example. (For details on the Context Manager, see the Context Manager/VM Manual.)

If a partition managing program exists in memory, additional application partitions also can exist in memory.

SWAPPING

When space for new partitions is needed, the operating system swaps partition managed programs out of memory to a disk file or to upper memory (above the first megabyte).

Figure 2-3A shows the Context Manager and Program W, Program X, and Program Y in memory. Figure 2-3B shows Program X swapped out and Program Z swapped in.

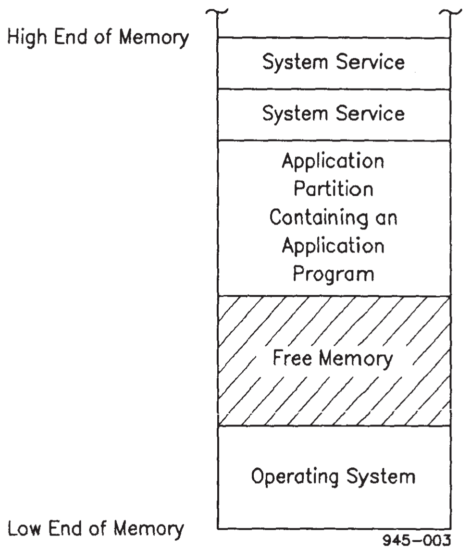


Figure 2-2. Memory Organization with Application Partition and Free Memory

USER NUMBER

Each partition has a unique user number (historically the same as a partition handle) that is shared by all processes in the partition. The user number refers to the resources associated with the specified partition. It does not refer to a partition's particular size or physical location in memory.

In a cluster or network environment, the resources of each cluster workstation partition are identified at the other workstations by a user number, which has been translated so as to be unique among all workstations.

As an example of a user number, each partition containing a program in Figure 2-3 is a different user number. Note that Program Z's partition is in the same basic location that Program X's partition occupied when it was resident in memory. The user number of Program X's partition, however, can be used to refer to Program X, even when Program X is not resident in memory.

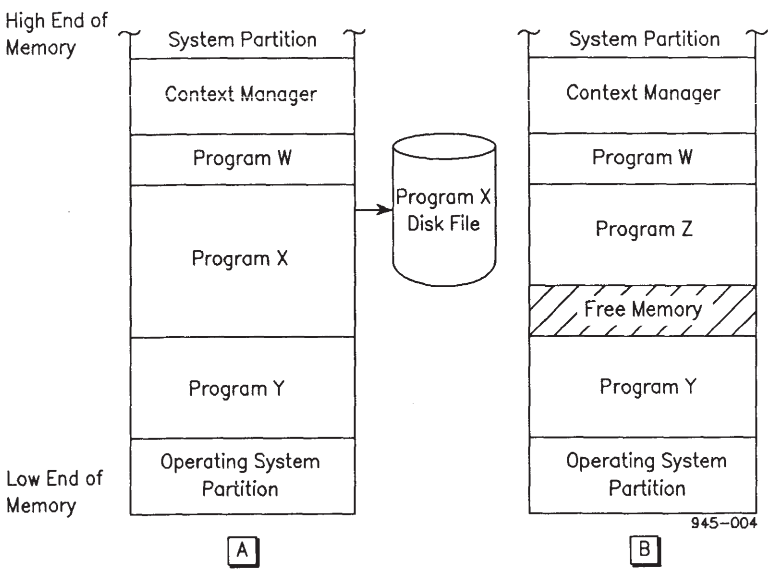


Figure 2-3. Memory Organization Under Partition Management

APPLICATION PARTITION MEMORY ORGANIZATION

The two types of memory allocation available to an application program are short-lived and long-lived. Within each application partition, short-lived memory expands downward from high memory locations, while long-lived memory expands upward from low memory locations. (See Figure 2-4.)

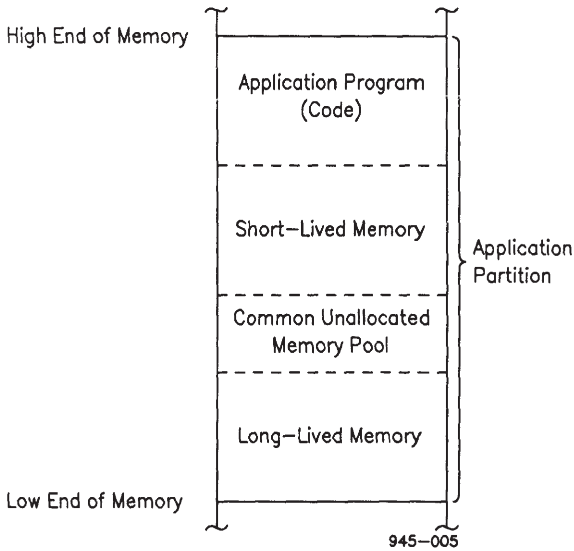


Figure 2-4. Memory Organization of an Application Partition

A program allocates short-lived memory to hold information it needs while executing. For example, it may need to build a record structure. Short-lived memory cannot be used to pass information to other partitions.

When the execution of a program is terminated, the short-lived memory of its partition is automatically deallocated.

Long-lived memory, however, is deallocated only at the specific request of the program. It is, therefore, useful for passing information from one program to another. The Executive uses long-lived memory for passing parameters to application programs that will run in the same partition. The Executive typically deallocates long-lived memory whenever it is reloaded.

Programs can allocate and deallocate short-lived and long-lived memory by making operating system requests. A program in one partition cannot allocate or deallocate memory in another partition.

VIRTUAL CODE MANAGEMENT FACILITY

The Virtual Code Management facility permits the execution of an application program that exceeds the physical memory of an application partition, by the use of relocatable overlays. To ensure optimal performance, the use of this facility is under the programmer's control.

FIXED AND VARIABLE PARTITIONS

A partition can be a fixed partition or a variable partition. A fixed partition always uses a fixed amount of memory. A variable partition (protected mode operating systems only) can use up to the maximum amount of memory that the program executing in it may allocate. (For details, see the Linker/Librarian Manual.)

CODE SHARING

Variable partitions (protected mode operating systems only) permit a program's code to be shared by the same type of program in another variable partition. Shared code can be located anywhere in physical memory.

3 USING CTOS/VM OPERATIONS

This chapter is provided to help you get started using the CTOS/VM operations in the programs that you write.

ASSUMPTIONS

It is assumed that the operating system has been successfully installed on your workstation. In addition, you should have installed the language compiler for the high-level language you will be using and the Software Development Utilities. The Software Development Utilities include the Linker, the Librarian, the Assembler, CTOS.lib, and so forth. (See the Release Notice for Standard Software for more information.)

If the above assumptions are correct, you can use your workstation for writing software programs.

You also should have available the documentation you will need to refer to while you are writing your programs. At a minimum, you will need the CTOS/VM Reference Manual. The Linker/Librarian Manual, the Assembly Language Manual, the Debugger Manual, and the appropriate programming language manual are other supporting software manuals that you should have when you are ready to compile, link, and run your program.

NAMING CONVENTIONS

You will notice that certain conventions are used to name variables in the CTOS/VM Reference Manual and other supporting software manuals. You need to familiarize yourself with the naming conventions used in these manuals to understand what the variables mean when you write programs that use the CTOS/VM operations.

See the Quick Reference card on Naming Conventions that is packaged with this manual. It provides information on the naming conventions most commonly used.

It is recommended that you follow the same naming conventions when you are developing software.

INTERFACE

The programmatic interface to any of the CTOS/VM operations is a procedural call.

FORMAT

The format of the procedural interface is given for each operation in the CTOS/VM Reference Manual. The following are examples of what this format looks like for three CTOS/VM operations:

```
WildCardInit (pb, cb, pBuf, sBuf): ercType

PutFrameCharsAndAttrs(iFrame, iCol, iLine,
        pbText, cbText, pbAttrs, cbAttrs):
        ercType

OpenFile(pFhRet, pbFilespec, cbFileSpec,
        pbPassword, cbPassword, mode):
        ercType
```

The operation name is to the left of the left parenthesis. You cannot change this name. The names enclosed within the parentheses are variable names representing parameters. Note that these variable names follow the naming conventions described in the Quick Reference card.

For example,

    pFhRet

means the memory address (p) of a file handle (Fh) returned (Ret) to your program.

The CTOS/VM Reference Manual includes a description for each operation. The description tells you what to fill in for each parameter to the procedural interface. (See the following example for details.)

Almost all CTOS/VM operations are written as function calls. A function call returns a one-word status code commonly known as an erc. Each of the preceding examples is an operation that returns a status code and, therefore, is labeled ercType.

If an ercType operation returns with no error, it returns a status code of 0 or ercOK. The operating system itself does not report any errors to the user; it simply returns status codes to programs that use operating system services. Programmers should always check the returned status code and provide for error reporting or recovery.

EXAMPLE STATEMENT

To use the procedural interface format, you must write it as a language statement. For example, the format of OpenFile looks like

```
OpenFile(pFhRet,   pbFilespec,   cbFileSpec,
         pbPassword,   cbPassword,   mode):
         ercType
```

The following is an example of how you can fill in the parameters to OpenFile in Pascal. Each variable name (from left to right) is described and followed by what you write for it.

1. pFhRet is the address to which the file handle for the open file will be returned, for example:

   ADS fh

2. pbFileSpec is the address of a file specification. You might declare the file specification as an LSTRING type and address it by reference, for example:

   ADS lsFileSpec[1]

3. cbFileSpec is the length in bytes of the specification, for example:

   lsFileSpec.len

4. pbPassword is the memory address of the file password. For example, no password required is indicated as

   NULL

5. cbPassword is the length in bytes of the password. For example, no password is indicated as

   0

6. mode is a two-letter constant indicating the mode in which the file is to be opened. For example, read mode is indicated as

   'mr'

The completed OpenFile statement in Pascal is thus

```
erc := OpenFile(ADS fh, ADS lsFileSpec[1],
         lsfileSpec.len, NULL, 0, 'mr');
```

OPERATION TYPES

Your program can use the procedural interface with any of the following types of operations:

- object module procedure
- system-common procedure
- (operation that uses the) request procedural interface to system services
- Kernel primitive

Each CTOS/VM operation in the CTOS/VM Reference Manual is identified as one of these types.

Each operation type functions in the operating system in a different way.

OBJECT MODULE PROCEDURE

An object module procedure is a procedure in a library. It is not part of the operating system code itself. The Linker links an object module with your program as part of the code that is executed when your program is run.

WildCardInit is an example of an object module procedure. When your program executes a call to WildCardInit, control is transferred to the WildCardInit code. When WildCardInit has completed executing, it returns to the next executable instruction in your program.

WildCardInit is in the standard operating system library, CTOS.lib. All of the CTOS.lib object module procedures are included in the "Operations" chapter in the CTOS/VM Reference Manual.

SYSTEM-COMMON PROCEDURE

A system-common procedure does not reside in a library nor is it linked with your program. It is a procedure within the operating system itself. A system-common procedure is either so common that it should not have to be duplicated, or it is hardware-dependent code too extensive to be included in every program written. System-common procedures increase program performance.

PutFrameCharsAndAttrs is an example of a system-common procedure.

KERNEL PRIMITIVES

The Kernel primitives are part of the operating system. They are

- Check
- CreateProcess
- ForwardRequest
- PSend
- Request
- RequestDirect
- RequestRemote
- Respond
- Send
- Wait
- WaitLong

These primitives are described in Chapter 29, "Interprocess Communication," and Chapter 30, "Inter-CPU Communication."

ACCESSING SYSTEM SERVICES USING THE REQUEST PROCEDURAL INTERFACE

The request procedural interface is a routine within the operating system used to access a system service. It calls the Kernel primitive, Request, to do this. The request procedural interface is not linked with your program. Instead, an interrupt is generated, which transfers control to the request procedural interface routine. Your program is placed in a waiting state while the routine executes.

The request procedural interface first constructs a request block. The request block is a message used by all interprocess communications. It is constructed according to specific conventions from the parameters you supplied in the procedural interface.

The request procedural interface then calls Request to route the request block to the system service. When the system service completes its service, it fills in its response in the request block and calls the Kernel primitive, Respond. Respond routes the request block back to your program.

Upon completion, a status code is returned to your program. A status code of 0 (ercOK) indicates that the system service performed the operation with no error.

The CTOS/VM operations that use the request procedural interface are request-based operations. OpenFile is an example. You can identify the request-based operations in the CTOS/VM Reference Manual by the request block format following the operation description.

ACCESSING SYSTEM SERVICES USING THE KERNEL PRIMITIVES

To access a system service using Kernel primitives, you are required to construct the request block yourself for the specified request-based operation. Then you must call the Kernel primitives, Request and Wait (or Check), for the request to be serviced.

This method of accessing a system service has the advantage of allowing your program to continue execution while it periodically checks for the response from the system service. The request procedural interface always requires that your program wait for the response. The request procedural interface, however, is easier to use.

It is recommended that you read the advanced chapters in this manual before you use the Kernel primitives in this way. (See Chapter 29, "Interprocess Communication," for more information.)

INTERFACE LEVELS

Figure 3-1 shows the I/O chapters in this manual. Each chapter (except Chapter 6, "Input/Output," which is introductory) presents the interfaces you can use to perform I/O to and from hardware devices.

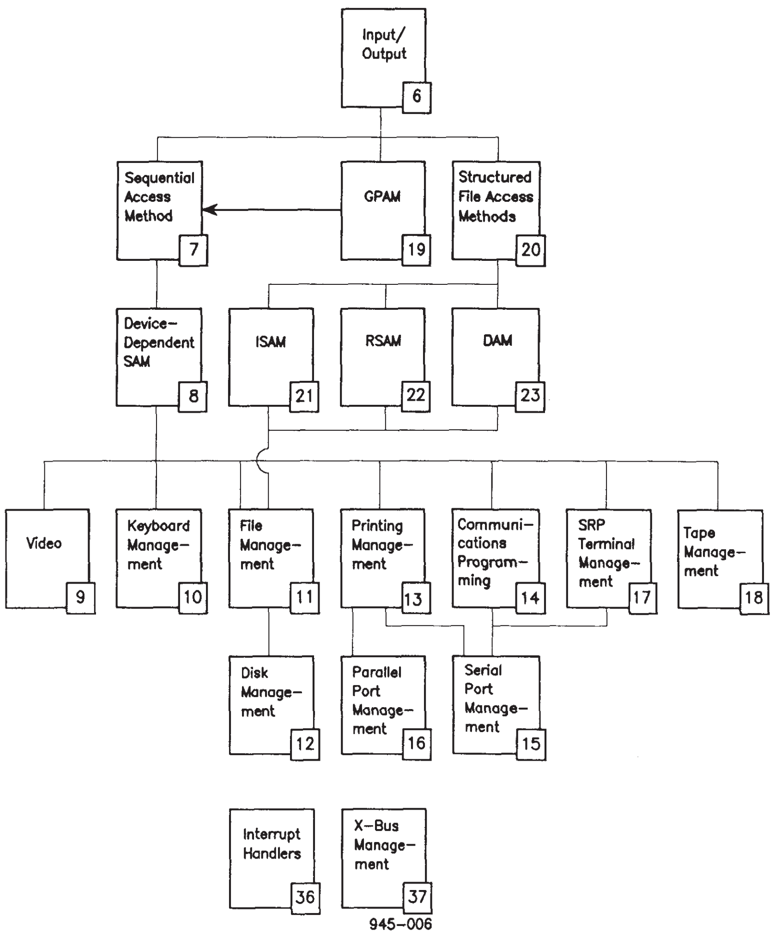


Figure 3-1. Interface Levels

I/O interfaces are available for the same device at different interface levels. The level of an interface implies the degree of control a program has over a hardware device when it uses that interface. Low-level interfaces provide greater hardware control than high-level interfaces but, at the same time, restrict a program to performing I/O to fewer devices.

The chapters closer to the top of Figure 3-1 describe high-level interfaces. Low-level interfaces are described in the chapters towards the bottom. The chapters with device names such as "Video" and "Disk Management," for example, describe low-level interfaces.

If you are getting acquainted with the CTOS/VM operations, the easiest way to access a device is at a high level. For example, you can use the operations in the Sequential Access Method (SAM) chapter to access the video device. The SAM interfaces are easier to use than the low-level video interfaces, because you write fewer statements in your program.

You will discover that there are advantages and disadvantages to using different interface levels.

The subject of interface levels is discussed at length in the I/O chapters. (See these chapters for more information.)

ADDRESSING MEMORY

In real mode, you are limited to a 1 megabyte physical address space. This means that your program can reference each of the 1,048,576 bytes by a unique physical address.

The physical memory address (PA) is the actual location in system memory.

In protected mode, the physical address space extends beyond the first megabyte. The amount of physical memory your program can address is determined by your system's processor and its hardware limitations. A 80286 processor, for example, is capable of providing a 16 megabyte physical address space. The actual address space, however, is determined by the hardware.

(For details on protected mode addressing, see the iAPX 286 Programmer's Reference Manual, the 80286 Architecture, and the 80386 Programmer's Reference Manual.)

A segment is a contiguous area of less than 64K bytes within the physical address space. The operating system uses segmented addressing. This means every address is relative to a segment. (See Chapter 24, "Memory Management," for details.)

You can think of a memory address as having a logical, a linear, and a physical translation. Figure 3-2 summarizes these translations.

LOGICAL MEMORY ADDRESS

The logical memory address is the 32 bit memory address as viewed by an application program. For example,

    pFh

is the logical memory address (denoted by p) of a file handle (denoted by Fh) . The logical memory address is used more frequently than either its physical or linear memory address translation.

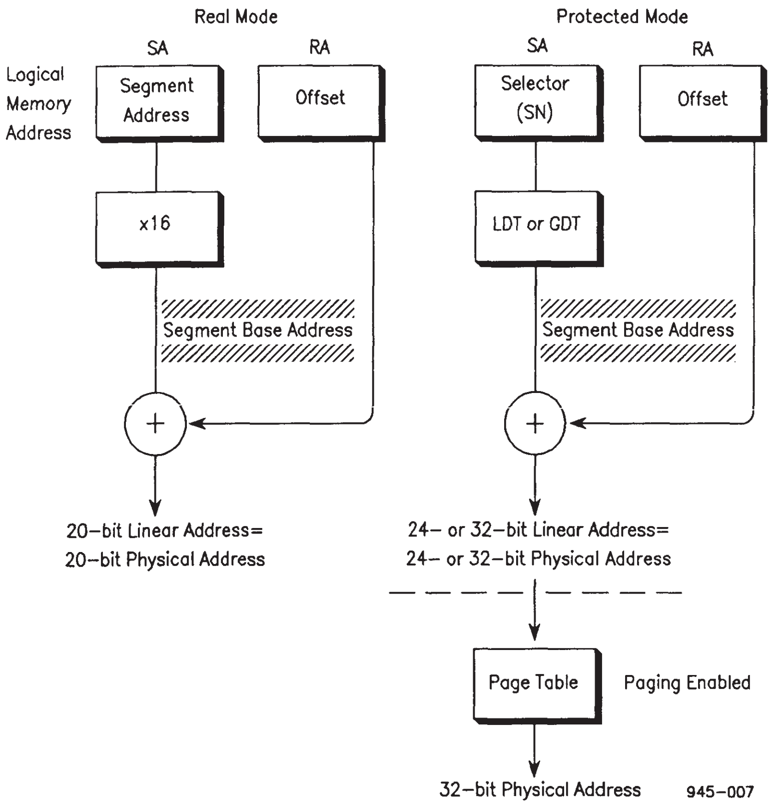


Figure 3-2. Memory Address Translations

The logical memory address consists of a <u>segment address</u> (SA) and a <u>relative address</u> (RA). (The relative address is commonly called the <u>offset</u>.) The syntax of a logical memory address in assembly language is

SA:RA

The <u>SA</u> portion is the high-order 16 bits of the logical memory address.

The SA is interpreted differently, depending upon whether the processor is executing in real or in protected mode.

- In real mode, the SA is multiplied by 16 to determine the segment base address in physical memory.

- In protected mode, the SA is a selector (SN). It selects a segment descriptor entry in a protected mode system structure [either a Local Descriptor Table (LDT) or a Global Descriptor Table (GDT)].

  The segment descriptor selected by the SN contains a segment base address, which may be located anywhere in physical memory. For this reason, if you are writing a program you want to execute in protected mode, your program should not depend upon the value of the SN. (For details on writing protected mode programs, see the Engineering Update for 2.0 CTOS/VM.)

The RA (or offset) is the low-order 16 bits of a logical address. It is the distance, in bytes, of the target location from the beginning of the segment.

LINEAR MEMORY ADDRESS

The linear memory address is computed differently in real and in protected modes. (See Figure 3-3.)

- In real mode, a 20 bit linear memory address is computed by multiplying the SA of the logical address by 16 and adding the RA.
- In protected mode, a 24 or 32 bit linear memory address is computed by adding the RA to the 24 or 32 bit segment base address.

PHYSICAL MEMORY ADDRESS

The physical memory address is the actual location in system memory.

- In real mode, the physical memory address is equivalent to the linear memory address.
- In protected mode, the physical memory address is equivalent to the linear memory address unless paging is enabled.

  If paging is enabled, the 32 bit linear memory address maps to a 32 bit physical memory address via a page table structure.

MEMORY ADDRESSING IN THIS MANUAL

A byte of memory does not have a unique logical memory address. The same byte of memory can be referred to by many different combinations of SAs and RAs.

In this manual, the term memory address means the logical memory address. (Chapter 30, "Inter-CPU Communication," describes a linear address used for routing requests between processor boards on the SRP. This is the only case in which the memory address has a different meaning.)

ADVANTAGES TO PROTECTED MODE MEMORY ADDRESSING

Protected mode addressing provides certain advantages over real mode.

EXTENDED MEMORY

Protected mode extends memory, allowing you to run programs beyond the first megabyte of physical memory. Real mode programs, however, are limited to the first megabyte.

As an end user, this means you can run more programs in memory. As a programmer, you can reference physical memory addresses extending beyond the first megabyte up to the maximum allowed by your processor and hardware.

PROTECTION

In protected mode, programs are prevented from referencing static memory allocated to other programs, or from overwriting code. This is because LDTs and GDTs provide for limit and type checking, which place limitations on the memory programs can access.

4 PROGRAM MANAGEMENT

The Program Management facility provides operations used by a program to self-load into memory, to self-exit from memory, and to handle error conditions.

WHAT IS A PROGRAM?

An executable program can consist of code, data, and one or more processes in a partition in memory.

A program is loaded into memory from a disk-resident file or run file. Run files are created by compiling and/or assembling source language modules into object modules and linking the object modules together into code and data segments. (See Figure 4-1.)

SEGMENTS

A code segment contains only processor instructions (code) and is never modified once it is loaded into memory. Several processes can execute instructions from the same code segment. (For details, see "Code, Static Data, and Dynamic Data Segments" in Chapter 24, "Memory Management.")

A static data segment contains initial values of program data structures and is writable once in memory. Every invocation of a program gets a new static data segment.

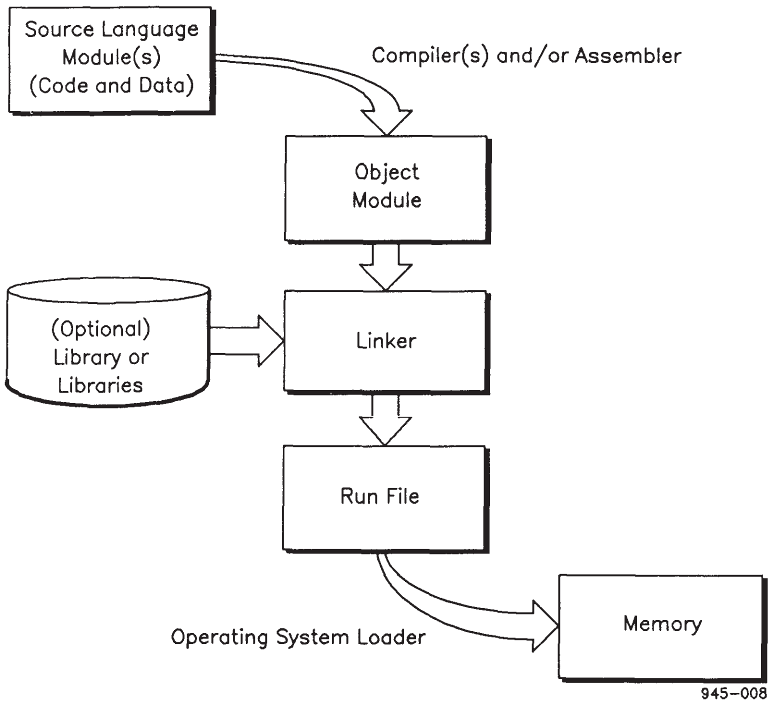


Figure 4-1. From Source Language Modules to Program in Memory

LINKER

The Linker reads the object module(s) and combines the segment elements contained within the modules according to their segment names, class names, and directives from the user. (For details, see the Linker/Librarian Manual.)

The run file that is created by the Linker consists of segments. Segments can be combined based on a series of different segmentation models. Most programming languages use the medium model, although the operating system also supports small and large model. (For details, see the CTOS Programmer's Guide.)

A run file created by linking object modules produced by the Pascal compiler, for example, consists of one code segment for each object module included in the link and a single static data segment. The single static data segment, or DGroup, combines the static data and stack requirements of all the object modules.

A run file of this form is considered standard; assembly language programmers are urged to adopt this standard unless other considerations are overriding. The COBOL compiler and BASIC interpreter do not produce object modules. (For details, see the Linker/Librarian Manual.)

PROGRAM LOADING INTO MEMORY

When a program is loaded into memory, the run file is read into the short-lived memory of the application partition. For real mode programs, any logical memory addresses existing in either the code or data segments (intersegment references) are adjusted to reflect the memory address at which the program is loaded. For protected mode programs, the Loader adjusts the base addresses in each Local Descriptor Table (LDT) descriptor.

The Virtual Code Management facility allows you to run a program that is larger than the available memory in an application partition. If the Virtual Code Management facility is in use, all the static data segments and the resident code segment are loaded in memory. The nonresident code segments are loaded in memory only as needed. (See Chapter 34, "Virtual Code Management," for details.)

A program is loaded by the Chain, Exit, ErrorExit, LoadPrimaryTask, or LoadInteractiveTask operation.

Note that LoadPrimaryTask and LoadInteractiveTask must be followed by a call to SwapInContext or AssignKbdOwner if a program is to be loaded into memory by a partition managing program. (For details on partition managing programs, see Chapter 32, "Program and Partition Management.")

EXIT RUN FILE

When the currently executing program exits, the exit run file is the next program that is loaded into the partition. Exit run files are user-specified. Each application partition has its own. For example, the Executive sets itself as the exit run file: the user starts the application from the Executive, and when the application is done, the Executive is reloaded.

A program can specify an exit run file for its partition by using the SetExitRunFile operation. A program can determine the exit run file of its partition by using the QueryExitRunFile operation.

If no exit run file is specified in a partition, the partition becomes vacant.

TERMINATING PROGRAMS

The application program terminates itself by using the Chain, Exit, or ErrorExit operation.

When a program terminates, the operating system issues termination requests. Termination requests (system requests) are messages that notify system services of a program's termination. Upon receipt of a termination request, system services release resources, such as open files, that may be allocated to the terminating program. (For details, see Chapter 31, "System Services Management.")

DEALLOCATION OF SYSTEM RESOURCES

Only the resources allocated to an exiting program are deallocated when that program terminates.

The resources that are deallocated include

- Short-lived memory. (See Chapter 24, "Memory Management.")
- Exchanges. (See Chapter 29, "Interprocess Communication.")
- Files opened by the OpenFile operation (except long-lived files). (See Chapter 11, "File Management.")
- Timer Request Blocks allocated by the OpenRTCClock operation. (See Chapter 33, "Timer Management.")
- Communications channels allocated by the InitCommLine operation. (See Chapter 15, "Serial Port Management.")
- Global Descriptor Table selectors (SGs) (protected mode). (See the iAPX 286 Programmer's Reference Manual, the 80286 Architecture, and the 80386 Programmer's Reference Manual.)

OPERATIONS

The Program Management operations described below are categorized as error handling and normal program exit operations. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

ERROR HANDLING

FatalError — Terminates operation of the application program and passes an abnormal status code to the exit run file.

CheckErc — Checks status codes. If CheckErc is called with a nonzero status code, FatalError is called with that value.

ErrorExit* — Terminates the current application program in an application partition and passes an abnormal status code to the exit run file.

ErrorExitString* — Returns a string (usually printed) to the exit run file.

---

*Dynamically installed system services use these operations at a certain time during installation. (For details, see Chapter 31, "System Services Management.")

Crash — Causes system operation on a workstation to terminate, a crash dump to be written, the operating system to be reloaded, and SignOn to display the cause of the crash when it is restarted.

SetMsgRet — Same as ErrorExitString except the program does not exit.

NORMAL PROGRAM EXIT

Exit* — Terminates the current application program in an application partition and passes a normal status code to the exit run file.

Chain* — Replaces the current application program in an application partition with the specified run file.

SetExitRunFile — Establishes a new exit run file for an application partition.

QueryExitRunFile — Returns the name, password, and priority of the exit run file of an application partition.

---

*Dynamically installed system services use these operations at a certain time during installation. (For details, see Chapter 31, "System Services Management.")

5 PARAMETER MANAGEMENT

The Parameter Management facility provides a structured mechanism for passing limited information from one application program to its successor within the same application partition.

EXAMPLE PROGRAM

The Executive is a typical example of an application program that uses the Parameter Management facility.

The Executive interfaces with the user through a forms-oriented interface. A forms-oriented interface accepts parameters from the user.

The Executive thus passes user-supplied parameters to other programs. The way that the Executive does this is described below. (See the Executive Manual for details.)

In the Executive, the user types a command name on the command line. When the user presses **Return**, the Executive is given the command.

The Executive responds by writing the user-requested command form to the screen. The command form contains the appropriate prompts for the user to enter data.

If the user, for example, types **Delete** on the command line and presses **Return**, the following command form appears:

```
Delete

    File list        ______________________
    [Confirm each?]  ______________________
```

The command form consists of a list of prompts. The user enters data on the lines (parameter fields) in the form next to the prompts, correcting typing errors if necessary. When satisfied with the contents of the fields, the user presses **Go** to execute the command.

The Executive passes the parameters to the Delete program. The Delete program, in turn, deletes the user-specified files.

A forms-oriented interface, such as the Executive, is one type of program that can use the Parameter Management facility to its advantage. Parameter Management, however, can be used by any application program in a partition that needs to provide information to any other program that will run in the same partition.

PARAMETERS

A parameter consists of zero or more subparameters.

In the Executive **Delete** command described above, the prompt [Confirm each?], for example, accepts either

- zero parameters (meaning the user did not enter any information)
- one parameter (a Yes answer)

A subparameter typically consists of an arbitrary sequence of characters not including a space.

The prompt [File list] in the Executive **Delete** command allows the user to enter one or more file names. Each file name is a subparameter; the parameter is the complete file list the user entered on the File list line. (For details on Executive parameters, see the Executive Manual.)

As another example, the parameter

    1 abc Work.Fri

contains three subparameters, which are 1, abc, and Work.Fri. The space is the delimiter that separates the subparameters.

A space can be embedded within a subparameter by including the entire subparameter in single quotes. For example, the parameter

    '1 abc' Work.Fri

contains two subparameters: 1 abc and Work.Fri.

OVERVIEW OF PARAMETER MANAGEMENT STRUCTURES AND OPERATIONS

Programs using the Parameter Management facility must organize parameter data to simplify the method in which other programs extract the parameters.

The organized data is stored in the Variable Length Parameter Block (VLPB), a data structure in long-lived memory of the application partition. [For details, see "Variable Length Parameter Block (VLPB)," later in this chapter.] The memory address of the VLPB is stored in the Application System Control Block (ASCB) of the partition. [For details, see "Application System Control Block (ASCB)," later in this chapter.]

To place parameter data in an organized fashion into the VLPB, programs can use the Parameter Management operations for constructing the VLPB. (These operations are described in "Operations for Constructing the Variable Length Parameter Block," later in this chapter.)

To extract parameters from the VLPB, programs can use the Parameter Management operations for querying the parameters stored in that structure. (These operations are described in "Querying Parameters in the Variable Length Parameter Block," later in this chapter.)

APPLICATION SYSTEM CONTROL BLOCK (ASCB)

An Application System Control Block (ASCB) is automatically created in an application partition when the partition is created. The ASCB contains the memory addresses of various types of partition-specific information, such as the VLPB. This information is available to be queried by programs, such as the Executive, which execute in the partition. (See Chapter 26, "System Definitions," for details on how a program can obtain partition information from the ASCB. For details on the ASCB structure, see Table 4-1 in the CTOS/VM Reference Manual.)

VARIABLE LENGTH PARAMETER BLOCK (VLPB)

The Variable Length Parameter Block (VLPB) is a partition structure used by the Parameter Management facility to communicate parameters to programs.

The VLPB is created in the long-lived memory of an application partition. Its memory address is stored in the pVLPB field of the ASCB.

Conceptually, the VLPB can be described as a two-dimensional sparse array of strings. The Executive command form illustrates the parts of this array as follows:

- Each element (iParam, jParam) in the array is the value of a subparameter entered into an Executive command form.

- Each row (iParam) of the array corresponds to a line in the command form, with one row for each parameter.
- Each column (jParam) of the array corresponds to a subparameter.

QUERYING PARAMETERS IN THE VARIABLE LENGTH PARAMETER BLOCK

A program can query the VLPB to obtain parameter information by using three operations: RgParam, CParams, and CSubParams.

- RgParm returns the memory address of the array element specified by (iParam, jParam). Each element of the array returned by RgParam is actually a 6 byte block of memory called an sdType. The first 4 bytes are the memory address of the string. The last 2 bytes are the length of the string.
- CParams returns the number of parameters stored in the VLPB. CParams, for example, is the number of fields in an Executive command form plus 1.
- CSubParams returns the number of subparameters stored in the VLPB for a specified parameter. CSubParams, for example, is the number of subparameters the user entered in a specified field of an Executive command form.

Figure 5-1 shows the matrix of a VLPB array for the Executive.

rgParams (VLPB)	SubParam (jParam) 0	SubParam (jParam) 1	SubParam... (jParam) ... 2 ...	SubParam (jParam) n
Param 0 (iParam 0)	<command name>	<case>	<Redo keystroke buffer>	
		. . .		
Param n* (iParam n)	(n,0)	(n,1)	(n,2)	

*where the values in row n are the subparameters of the nth parameter

945–009

Figure 5-1. Matrix of a Variable Length Parameter Block for the Executive

The Executive places the following information in row 0 (iParam 0):

- The Executive command name, such as **Delete**, is placed into element (0,0).

- The case value entered when the command was created is placed into element (0,1). The case value specifies which command invoked the current run file (disk resident file) when more than one possibility exists. The case value can be queried by a run file to determine which command invoked it.

- The Redo keystroke buffer is placed into element (0,2). The Redo keystroke buffer contains the entire series of keystrokes that the user typed.

Rows 1 through n store the parameters and sub-parameters that the user entered in the command form.

EXAMPLE OF A VARIABLE LENGTH PARAMETER BLOCK FOR THE DELETE COMMAND

If the Executive **Delete** command were filled out as follows:

```
0  Delete
1  File list          abc  def  gh
2  [Confirm each?]    y
```

the VLPB would look like the matrix shown in Figure 5-2.

rgParams (VLPB)	SubParam (jParam) 0	SubParam (jParam) 1	SubParam (jParam) 2
Param (iParam) 0	Delete	00	Delete abc def gh y
Param (iParam) 1	abc	def	gh
Param (iParam) 2	y		

945-010

Figure 5-2. Filled-in Variable Length Parameter Block

When the user presses **Go**, the Executive organizes the data to simplify the extraction of the parameters.

The RgParam operation provides access to the parameters by returning to the caller the memory address of the array element specified by (iParam, jParam).

In Figure 5-2, for example, the memory address of abc is returned by RgParam (1,0); the address of def is returned by RgParam (1,1).

OPERATIONS FOR CONSTRUCTING THE VARIABLE LENGTH PARAMETER BLOCK

Initialization

The following operation sequence is recommended to initialize a VLPB:

- Call ResetMemoryLL to reset the long-lived memory of the partition. Note that ResetMemoryLL also deletes the contents of the Redo keystroke buffer.

- Call AllocMemoryLL to allocate the number of bytes required for containing the VLPB structure.

- Call RgParamInit to initialize the specified memory for the VLPB.

Parameter Construction

The construction of parameters for a VLPB is supported by three object module procedures: RgParamSetSimple, RgParamSetEltNext, and RgParamSetListStart.

RgParamSetSimple creates one subparameter per row of the VLPB sparse array.

To construct a VLPB array with more than one subparameter per row, a program must first call RgParamSetListStart. RgParamSetListStart sets the global variable for placing the subparameters in the VLPB. Following a call to RgParamSetListStart, a call to RgParamSetEltNext must be made for each subparameter to be created in the row.

The VLPB and the parameter-passing services of the Executive are applicable to any application program using the operating system.

VARIABLE LENGTH PARAMETER BLOCK STRUCTURE

The VLPB structure is a self-describing, two-dimensional array of character strings. Each element of the array rgSdoParam is a pair (ob, cb) of words, where

- ob is the offset within the VLPB of the corresponding row of the two-dimensional array
- cb is the number of bytes occupied by the row

The strings that make up a row are prefixed with a 1 byte count and packed together without padding.

When a program uses the operations for constructing a VLPB (described previously), the VLPB structure is filled in with values.

(See Table 4-31 in the CTOS/VM Reference Manual, for the format of the VLPB.)

OPERATIONS

The Parameter Management operations described below are categorized by function. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

QUERYING PARAMETERS

The operations below are used by every program to query parameters in the VLPB.

CParams — Returns the number of parameters stored in the VLPB.

CSubParams — Returns the number of subparameters stored in the VLPB for a specified parameter.

RgParam — Provides access to the parameters stored in the VLPB.

CONSTRUCTING PARAMETERS

The operations below are used by only a few systems programs to construct parameters.

RgParamInit — Initializes the specified memory to be the VLPB.

RgParamSetSimple — Creates a parameter with one sub-parameter.

RgParamSetEltNext — Creates an additional subparameter of the current parameter in the VLPB.

RgParamSetListStart — Initiates the creation of a parameter with multiple subparameters.

6 INPUT/OUTPUT

This chapter is a guide to the CTOS/VM I/O facilities. It presents interface options available and discusses considerations you need to make regarding these options.

Figure 6-1 shows the I/O chapters in this manual. Each chapter (except this chapter, which is introductory) presents interfaces you can use to send I/O to a hardware device.

I/O interfaces are available for the same device at different interface levels. The level of an interface implies the relative degree of program control over a hardware device. Low-level interfaces provide greater hardware control than high-level interfaces but, at the same time, limit program I/O to fewer devices.

In Figure 6-1, the chapters towards the top describe high-level interfaces. Low-level interfaces are described in the chapters towards the bottom.

DEVICE INDEPENDENCE VERSUS DEVICE DEPENDENCE

A program's capability to run on various devices is a characteristic built into the program's design as a result of its interface level.

A device-independent program is capable of performing I/O to devices of different types, such as video, tape, or keyboard. Using high-level interfaces, thus, results in device independence.

You can write a device-independent program, for example, by using the Sequential Access Method (SAM) operations.

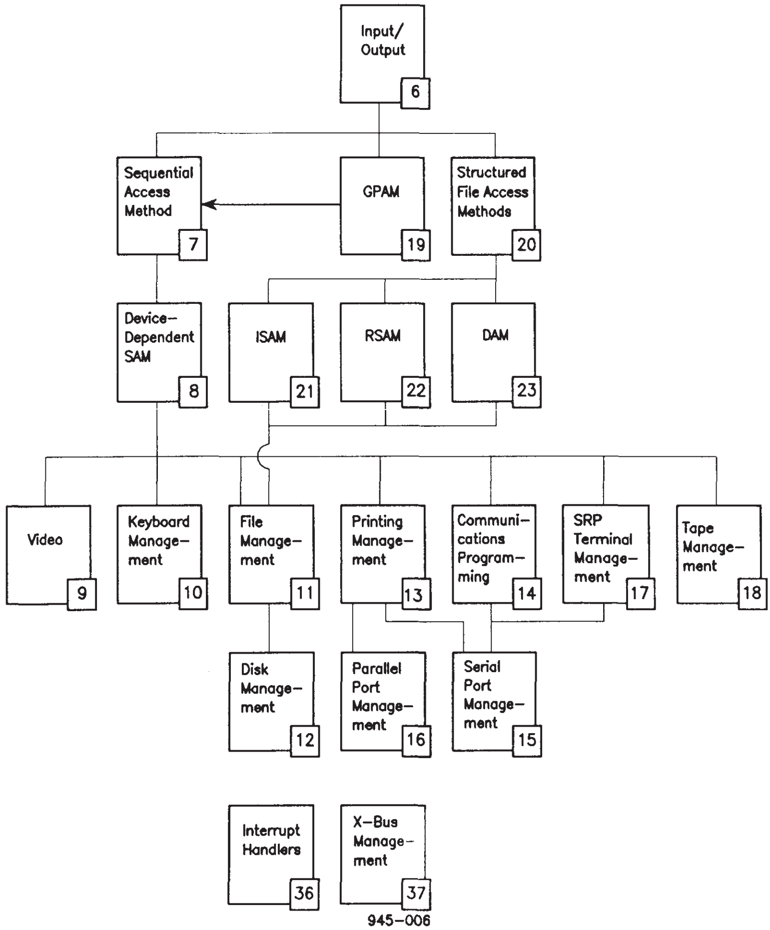


Figure 6-1. Interface Levels

A device-dependent program is limited to performing I/O to a limited number of devices or a particular type of device. Using the low-level interfaces, thus, results in a device-dependence.

The video device, for example, has its own group of video operations for performing I/O functions. These operations result in video device-dependence. The file system, keyboard, and other devices have their own comparable operations.

A device-dependent program provides for more direct control over the physical device but, at the same time, requires more effort to write.

I/O FACILITIES

Any of the following I/O facilities can be used for the same device:

- High-level to low-level device access to data:

  - Sequential Access Method (SAM). SAM is known more familiarly as byte streams. Using these high-level interfaces results in device-independence. (See Chapter 7, "Sequential Access Method.")

  - Device-Dependent SAM. Using certain operations at this lower level results in device-dependence. (See Chapter 8, "Device-Dependent SAM.")

- Device level. The operations at this level are specific to a given type of device and thus result in device-dependence. (See the device-named chapters, such as Chapter 9, "Video," and Chapter 10, "Keyboard Management." These are shown towards the bottom in Figure 6-1.)

  The chapters entitled "Disk Management," "Parallel Port Management," and "Serial Port Management" describe operations that are even closer to the actual device details than the operations described in the other device-named chapters. (See Figure 6-1.)

- Interrupt handlers and X-Bus. The chapters entitled "Interrupt Handlers" and "X-Bus Management" describe operations associated with more than one device.

• High-level device access to special kinds of data:

  - Generic Print Access Method (GPAM). GPAM provides high-level I/O for complex documents that may include text, graphics, or special text attributes. GPAM is an object module library that provides device independent formatting commands used for printing. (See Chapter 19, "Generic Print Access Method.")

• High-level device access to structured data files:

  - File access methods. Chapter 20, "Structured File Access Methods," is a guide to three high-level I/O interfaces to structured data files.

7 SEQUENTIAL ACCESS METHOD

The Sequential Access Method (SAM) provides device-independent access to a default set of real devices, such as the screen, printer, files, and keyboard. To transfer data to or from the device, SAM uses a character-oriented sequence of bytes known as a byte stream.

SAM consists of object module procedures in the standard operating system library, CTOS.lib.

SAM provides an alternative to the direct programming interfaces available at the device-dependent level. (The device-dependent interfaces are listed in chapters, such as "Video" or "Serial Port Management," which are associated with device names.)

With SAM you can write a program that can be

- used flexibly to access any of the available devices
- written with a minimum amount of code

If, for example, you want to write a compiler program that accepts its data from either the keyboard or a file and directs its listing to the screen, a printer, or a file, it would be to your advantage to use SAM's device-independent level of interface.

If, however, you know that your program will always perform I/O to a single device, it would be to your advantage to use the device-dependent level of interfaces for that device.

Device-dependent interfaces are specific to each kind of peripheral device available on a workstation. Programming at the device-dependent level has the advantages of

- maximizing run time efficiency
- providing access to the specialized features of the peripheral device hardware (for example, controlling the cursor at the video level)

CUSTOMIZING THE SEQUENTIAL ACCESS METHOD

The default devices that SAM supports are as follows:

- disk
- parallel printer
- spooler
- keyboard
- null
- video

For some applications, you may not need to use all of the devices supported by SAM. For example, a program might use SAM only to obtain keyboard input and to display text on the screen.

If this is the only way you use SAM in a particular application, you can configure SAM's device-dependent object modules selectively to support only the devices you need.

You generate SAM (SAMGen) by editing a configuration file, assembling it, and linking the resulting object module with your program.

Specific uses of a SAMGen are

- reduction of the memory needed by an application program by eliminating unneeded device support
- inclusion of support for communications and RS-232-C serial communications printers
- inclusion of support for tape
- inclusion of support for the Generic Print System (GPS)
- inclusion of user-written, device-specific SAM object modules

(For details on customizing SAM object modules, see "Building a Customized SAM" in the CTOS Programmer's Guide.)

BYTE STREAM

A byte stream is a readable (input) or writable (output) sequence of 8-bit bytes. An input byte stream can be read until either the reader chooses to stop reading or until status code 1 ("End of file") is returned. An output byte stream can be written until the writer chooses to stop writing. (Of course there are physical limitations: a file could expand, for example, to fill all available disk storage.)

A Byte Stream Work Area (BSWA) is a 130 byte memory work area for the exclusive use of SAM operations. Any number of byte streams can be open concurrently, using separate BSWAs. A BSWA must be allocated for each byte stream opened. (For details on the BSWA, see the CTOS Programmer's Guide.)

USING A BYTE STREAM

To open a byte stream, call the OpenByteStream operation, supplying the following parameters:

- the device/file specification string from the list in "Device/File Specifications," presented later in this chapter
- a password if appropriate
- the mode (indicating whether I/O is needed)
- the address of the 130 byte BSWA
- the address and size of the user-allocated buffer

When calling other device-independent operations such as ReadBsRecord, WriteBsRecord, or CloseByteStream, you supply the address of the same BSWA.

There are two predefined and already allocated BSWAs (bsVid for video frame 0 and bsKbd for the keyboard). These special BSWAs are defined in SAM standard object modules. Because these BSWAs are already opened, it is not necessary (nor allowed) to specify them as arguments to OpenByteStream or CloseByteStream. These byte streams may be used by passing the memory address of bsVid or bsKbd to the appropriate byte stream operations.

TYPES OF BYTE STREAMS

The types of byte streams that SAM supports are described below.

DISK BYTE STREAMS

A disk byte stream is a byte stream that uses a file on disk. A valid file name follows the standard file naming conventions. (For details on file naming, see Chapter 11, "File Management.")

Disk byte streams permit both input and output to be directed to the same open byte stream (that is, the same BSWA).

The standard operations of SAM are augmented by two operations that allow random access to files: GetBsLfa and SetBsLfa. These device-dependent operations are available only for disk byte streams and return status code 7 ("Not implemented") if attempted on other byte streams. (For details, see Chapter 8, "Device-Dependent SAM.")

PRINTER BYTE STREAMS

A printer byte stream is a byte stream that performs direct printing. Valid strings for printer byte streams are [LPT] and [PTR]n. n is any valid RS-232-C serial communications channel in a [COMM] device specification if a printer is attached to that serial port. (For details on communications channels, see "Device/File Specifications," later in this chapter.)

Direct printing transfers text directly from application program memory to the specified parallel or serial printer interface of the workstation on which the application program is executing. A printer byte stream cannot be used to access a printer assigned to the GPS or to the spooler. (See "Generic Print System Byte Streams" and "Pre-GPS Spooler Byte Streams," next in this chapter.)

The selected configuration file determines the printer characteristics. (See the **Create Configuration File** command in the CTOS System Administrator's Guide.) For example, the configuration file controls whether a printer byte stream suspends execution of the caller until the workstation operator corrects a condition requiring manual intervention or reports it to the calling program.

Normally printer byte streams change tab and end-of-line characters to the form expected by the printer. Return (code 0Ah), for example, can be transformed to a Carriage Return/Linefeed combination for some printers, or just to a Carriage Return (code 0Dh) or to a Linefeed (code 0Ah) for others. Tab characters can be transformed to spaces for printers without mechanical tabs. These transformations are controlled by the selected configuration file.

Any of three printing modes can be specified with the SetImageMode operation: normal, image, or binary. SetImageMode sets the printing mode any time following the opening of the printer byte stream. This differs from the effect of SetImageMode when used with pre-GPS spooler byte streams.

For compatibility between spooled and direct printing, SetImageMode should be used before the first WriteBsRecord or WriteByte operation.

Normal mode converts tabs into spaces and converts end-of-line characters to device-dependent codes.

Image mode and binary mode perform no code conversion.

Binary mode does not print the banner page or send any extra code not in the file to the printer, nor does it recognize the escape sequences controlling special video capabilities. (For details on the video escape sequences, see Chapter 9, "Video.")

GENERIC PRINT SYSTEM BYTE STREAMS

A Generic Print System (GPS) byte stream is a byte stream that is sent to a GPS printing device. GPS byte streams supersede pre-GPS spooler byte streams. (See "Pre-GPS Spooler Byte Streams," next in this chapter. Also see "Device/File Specification Parsing," later in this chapter.)

For compatibility with pre-GPS spooler byte streams, GPS byte streams implement the SetImageMode operation in the same way as pre-GPS spooler byte streams.

PRE-GPS SPOOLER BYTE STREAMS

(See the Printing Guide before using a pre-GPS spooler byte stream and for details on pre-GPS spooler escape sequences. For details on pre-GPS printing, see Appendix A, "Spooler Management.")

A pre-GPS spooler byte stream automatically creates a uniquely named disk file for temporary text storage. It then transfers the text to the disk file and expands the disk file as necessary. When the spooler byte stream is closed, a request is queued for the spooler by the Queue Manager for later printing of the previously created disk file. The temporary file is deleted after it is printed. This is spooled printing.

Normally, pre-GPS spooler byte streams change tab and end-of-line characters to the form expected by the printer. For example, a system Return (code 0Ah) can be transformed to a Carriage Return/ Linefeed combination for some printers, or just to a Carriage Return (code 0Dh) or a Linefeed (code 0Ah) for others. Tab characters can be transformed to spaces for printers without mechanical tabs. These transformations are controlled by the selected configuration file. (For details, see the **Create Configuration File** command in the CTOS System Administrator's Guide.)

Any of three printing modes can be set with the SetImageMode operation: normal, image, or binary. SetImageMode sets the printing mode only if it is called immediately following the opening of the spooler byte stream. This differs from the effect of SetImageMode when used with printer byte streams. (See "Printer Byte Streams," earlier in this chapter.)

For compatibility between spooled and direct printing, SetImageMode should be used before the first WriteBsRecord or WriteByte operation.

Normal mode prints the banner page between files, converts tabs into spaces, converts end-of-line characters to device-dependent codes, and recognizes the escape sequences for manual intervention. (For details on banner pages, see the Printing Guide.)

Image mode prints the banner page between files and recognizes the escape sequences, but performs no code conversion.

Binary mode does not print the banner or send any extra code not in the file to the printer, nor does it recognize the escape sequences. Escape sequences are special character sequences that invoke special functions.

KEYBOARD BYTE STREAMS

A keyboard byte stream is equivalent to using the ReadKbd operation in character mode. (For details on keyboard program modes, see Chapter 10, "Keyboard Management.") The keyboard byte stream does not support unencoded keyboard mode.

To support device-independence, keyboard byte streams return status code 1 ("End of file") when the FINISH (ASCII value 4) key is pressed, and status code 4 ("Operator intervention") when the CANCEL (ASCII value 7) key is pressed.

(For details on submit file escape sequences, see Chapter 10, "Keyboard Management.")

COMMUNICATIONS BYTE STREAMS

A communications byte stream is a byte stream that uses an RS-232-C serial communications channel (serial port). Communications byte streams provide support for the two communications channels of the serial input/output (SIO) communications controller. Operation is in asynchronous, full-duplex mode without explicit modem control. Like disk byte streams, communications byte streams permit both input and output to be directed to the same open byte stream (that is, the same BSWA). Only one byte stream can be opened for each communications channel of the SIO controller.

The selected configuration file determines the communications characteristics. (For details, see the **Create Configuration File** command in the CTOS System Administrator's Guide.)

Normally, communications byte streams strip null (00h) and delete (7Fh) characters from the stream of received data characters. Image mode (set with the SetImageMode operation) specifies that communications byte streams pass all incoming characters to the requesting program exactly as received.

X.25 BYTE STREAMS

An X.25 byte stream is a byte stream that enables data transmission via the X.25 Network Gateway. (For details, see the X.25 Network Gateway Manual.)

Each open X.25 byte stream corresponds to a virtual circuit that is initiated when the byte stream is opened, and cleared when the byte stream is closed. Setting up and clearing of the virtual circuit is .controlled through the use of a configuration file.

VIDEO BYTE STREAMS

A video byte stream is a byte stream that uses the video display. The standard SAM operations are augmented by

- Certain characters that have special interpretation.
- Multibyte escape sequences. The multibyte escape sequences (beginning with the character 0FFh) can be used to control the special workstation video capabilities.
- One device-dependent operation. The QueryVidBs operation returns information about video byte streams.

(See Chapter 9, "Video," for details on video byte streams and on other ways to control the video subsystem.)

TAPE BYTE STREAMS

A tape byte stream reads or writes a tape as a purely sequential device. It looks for the pattern of file marks that designate the beginning and end of a file. Within the limits specified by the tape configuration file, tape byte streams for half-inch tape ignore exact record and block sizes when reading.

With tape byte streams, you can read or write to tape using the standard byte stream interface. Valid tape names include the characters [TAPE] or [QIC] plus additional information. (For details on tape naming, see Chapter 18, "Tape Management.")

In read mode, records are read from the tape as a sequence of bytes until a file mark is encountered. The user is not aware of the record size.

For half-inch tape in Write mode, the record size is obtained from the tape configuration file.

Tape byte streams are not included in the standard SamGen. They must be included by performing a custom SamGen. (For details, see the CTOS Programmer's Guide.)

DEVICE/FILE SPECIFICATIONS

The device/file specification string is any of the following:

{node}[volname]<dirname>filename
: File identified by its full file specification. Abbreviated specifications are also allowed. (See Chapter 11, "File Management," for details on file names.)

[LPT]&[volname]<dirname> filename

Centronics-compatible printer connected to the parallel printer port. (See Appendix A, "Spooler Management.")

&[volname]<dirname>filename is optional. It describes a configuration file containing the printer characteristics. A default configuration file is used if none is specified. (For details, see the **Create Configuration File** command in the CTOS System Administrator's Guide.)

[PTR]n&[volname]<dirname>filename

RS-232-C-compatible printer, where n identifies the serial I/O (SIO) communications channel to which the printer is connected and can be any of the channels listed below.

&[volname]<dirname>filename is optional. It describes a configuration file containing the printer characteristics. A default configuration file is used if none is specified. (For details, see the **Create Configuration File** command in the CTOS System Administrator's Guide.)

[COMM]n&[volname]<dirname>filename

Communications channel n of the SIO communications controller, where n identifies the channel.

&[volname]<dirname>filename is optional. It describes a configuration file containing the communications characteristics. A default configuration file is used if none is specified. (For details, see the **Create Configuration File** command in the CTOS System Administrator's Guide.)

Valid channel identifiers are listed below:

Channel Synonyms	Processor Channel	Device
A 0 0A	A	Workstations, SRP, TP and CP
B 1 0B	B	Workstations, SRP, TP and CP
C 2	C	SRP - TP and CP
D 3	D	SRP - TP only
E 4	E	SRP - TP only
F 5	F	SRP - TP only
G 6	G	SRP - TP only
H 7	H	SRP - TP only
I 8	I	SRP - TP only
J 9	J	SRP - TP only

The following specifications support the XC-002 port expander module:

1A	Leftmost XC-002, Channel A
1B	Leftmost XC-002, Channel B
1C	Leftmost XC-002, Channel C
1D	Leftmost XC-002, Channel D
2A	Second XC-002, Channel A
2B	Second XC-002, Channel B
2C	Second XC-002, Channel C
2D	Second XC-002, Channel D

[QICm]n — Quarter-inch cartridge (QIC) tape. (For details on tape naming, see "Tape Names" in Chapter 18, "Tape Management.")

[TAPEsm]n — Half-inch tape. (For details on tape naming, see "Tape Names" in Chapter 18, "Tape Management.")

{node}[queuename]reportname

Spooled printer. The queue name is the name of the pre-GPS scheduling queue associated with the spooler. [SPL] is the default name of the first spooled printer.

The report name is a text string of up to 12 characters that is included in the **Spooler Status** command's status display. (For details, see the CTOS System Administrator's Guide.)

[KBD] Keyboard. This also includes the system input process used for submit files and batch jobs. (For details on the system input process, see Chapter 10, "Keyboard Management," in this manual. For details on batch, see the *CTOS System Administrator's Guide*.)

[X25]*n*&[volname]<dirname>filename
X.25 virtual circuit, where n is a network identification that currently must be zero.

&[volname]<dirname>filename is optional. It describes a configuration file containing the circuit characteristics. (For details, see the *X.25 Network Gateway Manual*.)

[NUL] Null device. Input operations always return status code 1 ("End of file"). Output operations discard all output but return status code 0 (ercOK).

[VID] Video frame 0. The frame must be established in advance using the Video Access Method (VAM) or the Executive. (For details, see Chapter 9, "Video.")

[VID]n Video frame n.

DEVICE/FILE SPECIFICATION PARSING

To determine the type of byte stream you are specifying, SAM parses the device/file specification string supplied to OpenByteStream. This string parsing process is described below.

Scanning from left to right, SAM first looks for a left bracket ([).

If a left bracket ([) is not found and disk byte streams are included in the SAM configuration, SAM assumes the string to be a file name. The byte stream is a disk byte stream, which is directed to a disk file.

If a left bracket ([) is found, Sam attempts to match the string characters and the string length within the square brackets to the reserved words for system devices, such as KBD, LPT, and PTR.

1. If a match occurs, SAM specifically looks for any characters to the right of the right square bracket (]).

   a. If a left angle bracket (<) is found, the string is assumed to be a file name, and the byte stream is therefore a disk byte stream.

   b. If no characters are found, the string is a reserved word for a device, and the device byte stream is directed to the specified device.

2. If no match occurs and GPS is installed, SAM assumes the byte stream is a GPS byte stream. Otherwise, if the spooler is installed, the byte stream is assumed to be a pre-GPS spooler byte stream.

OPERATIONS

The SAM operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

BASIC

OpenByteStream	Opens a device/file as a byte stream.
ReadBsRecord	Reads the specified count of bytes from the open input byte stream to the specified memory area.
ReadByte	Reads 1 byte from the open input byte stream.
WriteBsRecord	Writes the specified count of bytes to the open output byte stream from the specified memory area.
WriteByte	Writes 1 byte to the open output byte stream.
CloseByteStream	Closes the open byte stream.
OutputToVid0	Provides programs, such as system services, with the ability to perform minimal output to the video device without linking to a full video byte stream.

ADVANCED

ReadBytes	Reads up to the specified count of bytes from the open input byte stream. ReadBytes returns the memory address of the start of the byte stream but does not move the bytes to a separate buffer.
CheckpointBs	Writes any partially full buffers of the open output byte stream and waits for all write operations to complete successfully before returning.
ReleaseByteStream	Abnormally closes the device/file associated with the open output byte stream.
QueryVidBs	Allows your program to obtain information about a video byte stream.

8 DEVICE-DEPENDENT SAM

The Sequential Access Method (SAM) discussed in Chapter 7 highlights the device-independent aspect of SAM. By using the basic operations described in that chapter, you are allowing your program to be portable to a number of devices.

SAM, however, has a device-dependent portion to its code for each type of device it supports.

The device-independent operations map to device-dependent operations specific to each device. Mapping is done automatically each time a device-independent operation is called. It is based on information stored in the Byte Stream Work Area. (See Chapter 7, "Sequential Access Method.")

DEVICE-DEPENDENT OPERATIONS WITH GENERIC PREFIXES

Calling a device-independent operation results in mapping to a device-dependent operation with a generic prefix.

To send output to an open line printer byte stream, for example, you would call the device-independent operation, WriteBsRecord. WriteBsRecord, in turn, calls the device-dependent operation, FlushBufferLP. The generic prefix is FlushBuffer. LP (the name of the specific device) is appended to the prefix.

The device-independent operations and the generic prefixes to their device-dependent versions are as follows:

Device-Independent Operation	Generic Prefix
OpenByteStream	OpenByteStream...
ReadByte, ReadBsRecord	FillBuffer...
WriteByte, WriteBsRecord	FlushBuffer...
part of CloseByteStream	CheckPointBs...
part of CloseByteStream	ReleaseByteStream...
SetImageMode	SetImageMode...

(For details, see the CTOS Programmer's Guide, "Building a Customized SAM.")

DEVICE-SPECIFIC OPERATIONS

To handle select types of byte streams in special ways, you can incorporate certain device-specific operations directly into your program. The device-specific operations are as follows:

Operation	Applicable Byte Streams
SetImageMode	Communications, printer, Generic Print System (GPS), pre-GPS spooler
PutBackByte	Disk (async and sync)
GetBsLfa	Disk (async and sync)
SetBsLfa	Disk (async and sync)
QueryVidBs	Video

If you use these operations, you limit your program to specific devices. If, for example, you use GetBsLfa in your byte stream, your program will work only if you specify a disk file name.

Note that, although GetBsLfa and SetBsLfa pertain to files, these operations are called only through byte streams and are therefore included in this chapter rather than in Chapter 11, "File Management." The same is true of QueryVidBs, which is included here instead of in Chapter 9, "Video." QueryVidBs is a byte stream path for manipulating the video device.

(See the CTOS Programmer's Guide, "Building a Customized SAM," for details on how to use these operations in customizing your program.)

OPERATIONS

The device-dependent SAM operations described below are categorized by interface function. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

GENERIC PREFIXES

Every type of byte stream has operations whose names begin with the prefixes below.

OpenByteStream...
: Opens a specific device/file as a byte stream.

FillBuffer...
: Reads data from the device into a user-specified buffer.

FlushBuffer...
: Writes data from a user-specified buffer to the device.

CheckPointBs...
: Ensures that all data in the buffer has been output to the device (forms part of the CloseByteStream operation).

ReleaseByteStream...
: Releases the device for use by other programs (completes the CloseByteStream operation).

SetImageMode...
: Affects the interpretation of bytes read from or written to the device (for example, controls whether tabs are expanded or not).

DEVICE-SPECIFIC

The operations below limit your program to specific devices.

SetImageMode	Sets the normal, image, or binary mode for printer, spooler, and communications byte streams.
PutBackByte	Returns 1 byte to the open input disk byte stream.
GetBsLfa	Returns the logical file address at which the next I/O operation will occur for the open disk byte stream.
SetBsLfa	Sets the logical file address at which the I/O operation is to continue for the open disk byte stream.
QueryVidBs	Returns video information about the type of video device associated with an open video byte stream.

9 VIDEO

This chapter describes the video facility. The video facility is a highly flexible means for the display of alphanumeric and graphic information. Workstation video is of two types: character map and bit map.

Although most character map workstations can be equipped to display graphics, the primary feature is the video hardware contained to support the character map. The hardware reads characters and attributes from memory. It then converts them from the extended ASCII (8 bit) memory representation to a pattern of illuminated dots, called pixels, that it displays on the screen. During this conversion, the video hardware references a translation table (font) that is loaded into the video hardware under program control. Character map fonts are created with the Font Designer.

A bit map workstation does not contain hardware to support the character map (although it contains graphics hardware). Instead, the video software provides character map emulation to support character-only application programs. The font can be modified, but it is of a different format from the character map font. Bit map fonts are created with the Raster Font and Icon Designer.

The video facility is described here from the viewpoint of

- how you can use it to your advantage in your programs

- what video capabilities are available to you with each hardware type

(The details of programming using color are described in the CTOS/VM Reference Manual, Appendix F, "Using Color.")

VIDEO ATTRIBUTES

Video attributes can be either screen or character attributes and control the visual presentation of characters on the screen.

- Screen attributes control the presentation of the entire screen. Examples are blank, reverse video (dark characters on a light background), half-bright, number of characters per line, and the presence or absence of character attributes.

- Character attributes control the presentation of a single character. Examples are reverse video, blinking, half-bright, underlining, bold, and struck-through.

VIDEO SOFTWARE

The video software consists of a device-independent and a device-dependent level of interface to the video facility. Each level provides varying degrees of screen and character attribute control.

The screen consists of a number of separate, rectangular areas called frames. Each frame can be scrolled up or down independently of other frames. You can select from several features, including multiple frames and scrolling of each frame, to enhance your program video output.

The video software consists of the following two interface levels:

- At the device-independent level, you can use the Sequential Access Method (SAM). SAM provides device-independent access to devices such as the printer, files, keyboard, as well as the screen. (See Chapter 7, "Sequential Access Method.") SAM provides automatic scrolling. Video-specific extensions to the SAM provide direct cursor addressing, control of character attributes, and so on.

- At the device-dependent level,* you can use

  - The Video Access Method (VAM). VAM operations provide you with direct access to the characters and character attributes of each frame. They include explicit control of scrolling.

  - The Video Display Management facility (VDM). VDM consists of operations for screen setup: VDM controls the way that the screen appears. For example, the VDM operations enable you to split the screen into frames. VDM and VAM can be used together or independently, as described in "Program/Video Subsystem Interaction," which follows.

*Actually, VAM and VDM are device type-dependent operations. Although they limit your program output to a video device, they allow you to write to the video on any type of workstation.

PROGRAM/VIDEO SUBSYSTEM INTERACTION

You can choose to direct your program output to the screen using any of several methods. The methods described below range from simple (requiring little programming effort) to more complex (requiring more programming effort but providing greater output control).

SEQUENTIAL ACCESS METHOD (SAM)

You can use SAM's device-independent operations in two basic ways, as described below.

Using the Current Screen Setup

If you are writing a program such as a compiler that will be invoked by the Executive to display messages in a streaming or sequential way, you do not need to initialize the video display. Instead, you can take advantage of the Executive's screen setup. Screen setup allows you to use the device-independent SAM operations, such as OpenByteStream, specifying the video as your device string. SAM then generates a video byte stream for use by the video display. You can alternately use the pre-opened byte stream, bsVid.

The Executive eliminates the need to reinitialize the video because your program, when invoked, inherits the Executive's

- character font
- character map (in system memory)
- three frames (Command Frame, Event Frame, and Status Frame)

which comprise the Executive's current screen setup.

SAM's video byte stream extensions support multiple frames, character attributes, and explicit positioning of characters in a frame, but do not support line attributes (other than cursor position). SAM recognizes a few special cursor-positioning characters including **Return**, **Next Page**, **Backspace**, and **Tab**. When a special character or full line would cause the cursor to move below the bottom line of the frame, SAM automatically scrolls the frame and repositions the cursor.

Using SAM Directly

If you choose not to have your program use the Executive screen setup, you can still use SAM's device-independent operations as above, but you also must initialize the screen. [See "Video Display Management (VDM)," later in this chapter.] For example, if you want your program to be invoked <u>directly</u> by the Context Manager, you must use VDM to initialize the screen.

AUGMENTING THE SAM OPERATIONS

If you want greater control over the video byte stream, you can augment the SAM device-independent operations by the following:

- Special interpretation of certain characters.
- Multibyte escape sequences. The multibyte escape sequences (beginning with the character 0FFh) can be used to control the special video capabilities of the Convergent workstations.
- One device-dependent operation. The operation QueryVidBs returns information about video byte streams.

Each of these methods is described below.

Special Characters in Video Byte Streams

(See Table J-7 in the CTOS/VM Reference Manual for the special characters interpreted by video byte streams.) Note that a multibyte escape sequence is available to disable all these special interpretations except 0FFh.

Multibyte Escape Sequences

Multibyte escape sequences can

- control screen attributes
- control character attributes
- control scrolling and cursor positioning
- dynamically redirect a video byte stream
- automatically pause between full frames of text
- perform various other miscellaneous functions

Note that where the escape sequences include alphabetic characters, uppercase and lowercase are equivalent.

Controlling Screen Attributes. Screen attributes can be controlled with four multibyte escape sequences. (See Table J-4 in the CTOS/VM Reference Manual.) Each of the 3 byte sequences begins with the escape byte 0FFh and continues with a pair of characters represented by the specified 8 bit ASCII character codes.

Controlling Character Attributes. Character attributes can also be controlled with multibyte escape sequences. (See Table J-2 in the CTOS/VM Reference Manual.)

Workstations support six character attributes: blinking, bold, half-bright, reverse video, struck-through, and underline.

You can use the escape sequence for subsequent characters in video byte streams to set all six character attributes in any combination.

Controlling Scrolling and Cursor Positioning. Characters are normally written to the frame sequentially, with the cursor advancing one character position at a time, from left to right and top to bottom. A cursor is normally displayed at the character position where the next character will be displayed. Text is automatically scrolled each time a character is written to the lower right corner of a frame. When such a scroll occurs, the entire contents of the frame scroll up one line, and the contents of the previous top line of the frame disappear.

(See Table J-5 in the CTOS/VM Reference Manual for the escape sequences that directly control scrolling and cursor positioning.)

Dynamically Redirecting a Video Byte Stream. When a video byte stream is opened, it is designated as directed to one of the frames. However, a special escape sequence makes it possible to dynamically redirect a video byte stream.

An independent cursor position is recorded for each frame. The position within frame **i** is restored automatically when a video byte stream is redirected to frame **i**. (See Table J-1 in the CTOS/VM Reference Manual.)

Automatically Pausing Between Full Frames of Text. Automatic pausing between full frames of text can be controlled through multibyte escape sequences. When this pause facility is enabled and further output to the frame would cause text to be scrolled off the top of the frame, the message

Press NEXT PAGE or SCROLL UP to continue

is displayed on the last line of the frame. At this point, if the user presses **Next Page**, output continues for another full frame of text. If the user presses **Cancel**, status code 4 ("Operator intervention") is returned to the calling process. If the user presses **Finish**, status code 1 ("End of file") is returned to the calling process. If the user presses any other key, the audio alarm is momentarily activated. (See Table j-3 in the CTOS/VM Reference Manual for the escape sequences controlling pause.)

Since the automatic pause facility reads characters from the keyboard (using the operation ReadKbdDirect), there is potential for interaction with the client's use of the keyboard. (See Chapter 10, "Keyboard Management," for a description of the ReadKbdDirect operation.)

A single client using a keyboard byte stream and one or more video byte streams will operate correctly. A more complex environment may require using program-specific logic to control pauses in scrolling. Automatic pausing can be affected by

- use of the unencoded keyboard mode
- use of ReadKbd instead of a keyboard byte stream

- keyboard input performed by one client while another uses a video byte stream

- keyboard input initiated by the Kernel primitive, Request, but not immediately followed by the Kernel primitive, Wait

Miscellaneous Functions. See Table J-6 in the CTOS/VM Reference Manual for a description of the escape sequences that perform miscellaneous functions.

QueryVidBs

The QueryVidBs operation returns information about a video byte stream, such as frame number or current line number. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of this operation.)

VIDEO ACCESS METHOD (VAM)

If you want more direct control over the screen than SAM provides, you can use the Video Access Method (VAM) operations. If your program does not require special screen setup, you can use the VAM operations independently of the Video Display Management (VDM) operations. [See "Video Display Management (VDM)," next in this chapter.]

VAM provides direct access to the characters and character attributes of each frame. VAM operations can

- Put a string of characters anywhere in a frame.

- Specify character attributes for a string of characters.

- Scroll a frame up or down a specified number of lines.

- Position a cursor in a frame. (Each frame can have its own cursor.)

VIDEO DISPLAY MANAGEMENT (VDM)

If you choose not to use the Executive's screen setup or if your program is not invoked by the Executive, you can reinitialize the video subsystem using the VDM facility before using the VAM or SAM operations.

The VDM facility sets up the screen. By using the VDM operations, your program can

- determine the video capability present

- load a new character font into the font RAM

- stop video refresh on a character map workstation (useful when moving or changing the size of the frames or the character map)

- change screen attributes, such as reverse video and half-bright, while the screen is being video-refreshed

- calculate the amount of memory needed for the character map based on the preferred height and width of the characters, and the presence or absence of character attributes

- initialize each of the frames

- initialize the character map

Once the character map is set up and video refresh is started, you can use the VAM or the SAM operations to control the screen image by modifying the characters and attributes stored in the character map.

Reinitializing the Video Subsystem

Your program needs to reinitialize the video display only if the intended state is not the same as that provided by the Executive.

To reinitialize the video display, you must include a particular sequence of software operations similar to the following:

1. Use the QueryVidHdw operation to determine the level of video capability present on the workstation in use.

2. Optionally use the LoadFontRam operation to read the character font from a file to memory and then load this font into the font RAM.

3. Use the ResetVideo operation to place the following information in the Video Control Block:

   - number of characters per line

   - number of lines per screen

   - the presence or absence of character attributes

4. Use the InitVidFrame operation to specify the screen coordinates and dimensions of each of the frames.

5. Use the SetScreenVidAttr operation to set reverse video or half-bright, if wanted.

6. Use the InitCharMap operation to initialize the character map.

7. Use the SetScreenVidAttr operation to initiate video refresh.

On bit map workstations, you do not have to turn video refresh off and on during initialization.

On character map workstations that have graphics capability, using the SetScreenVidAttr operation to turn off video refresh turns off only the characters, not the graphics. However, on bit map workstations, where graphics and characters are not separated, both are turned off.

Following reinitialization, your program can display information by using VAM or SAM.

The Executive also allows you to use the **Screen Setup** command to respecify the following video characteristics:

- reverse video
- number of characters per line
- number of lines
- the presence or absence of character attributes
- suppress pause between pages
- color
- screen timeout

(For details on the **Screen Setup** command, see the Executive Manual.)

FORMS-ORIENTED INTERACTION

VAM is ideal for forms-oriented interaction, that is, interaction in which a form is displayed in a frame and the workstation user enters data into the blank fields of the form. Direct cursor addressing and modification of individual characters and character attributes support this interaction.

For example, the PutFrameAttrs operation is used to highlight the field to be entered next. It sets reverse video for the range of character positions that constitute the field. After the field is entered, PutFrameAttrs is used again to reset the reverse video attribute on the character positions of the field.

ADVANCED TEXT PROCESSING

VAM is also ideal for applications that perform advanced text processing, because it provides scrolling up and down of entire or partial frames. It is easy, for example, to scroll up the top four lines of a frame and insert a new line of text between the old fourth and fifth lines. During scrolling, character attributes scroll along with the text they affect.

WORKSTATION VIDEO CAPABILITIES

The workstation types and models have different video capabilities. These are summarized in Table 9-1. (See the CTOS System Administrator's Guide for information on configuring the video for your workstation.)

VIDEO CAPABILITIES

Note that, in the discussion below, the descriptions of video capabilities apply to either character map or bit map workstations, unless specified otherwise.

Table 9-1
VIDEO CAPABILITIES

	CHARACTER MAP	BIT MAP Low-res	Hi-res	Hi-res zoomed
Character Attributes				
Blinking	Yes	*	*	*
Bold	Yes	Yes	Yes	Yes
Half-bright	Yes	†	Yes	No
Reverse video	Yes	Yes	Yes	Yes
Struck-through	Yes	Yes	Yes	Yes
Underline	Yes	Yes	Yes	Yes
Loadable font	Yes	Yes	Yes	Yes
Number of characters/line	80	80	80	146
Number of lines/ screen	29	29	38	38

* Blinking is substituted with an outline character.

† Half-bright is emulated for consistency across the hardware, but it is recommended that you do not use it in your programs for the low-resolution monitor.

Character Cell

Table 9-2 shows the character cell sizes available for character map and bit map workstations.

Table 9-2
CHARACTER CELL SIZE

WORKSTATION TYPE/MODEL	SIZE
Character map	9 x 12
Bit map Low-resolution monitor	9 x 12
High-resolution monitor	12 x 20
High-resolution zoomed monitor	7 x 20

Based on the character cell size of your particular workstation, you can obtain other information describing the level of video capability programmatically by using the QueryVidHdw or the QueryVideo operation. (For details, see the descriptions of these operations in the CTOS/VM Reference Manual, Chapter 3, "Operations.")

Character Map

On a character map workstation, characters displayed on the screen are stored in a contiguous area called the character map. The video controller has its own RAM containing a 4K byte character map and a 4K byte soft font.

The character map consists of 2K bytes of words. Each word in the character map contains one ASCII character byte (low byte) and one attribute byte (high byte) that applies only to that specific character. The map and font can be updated at any time and the result is immediately visible on the screen.

On a bit map workstation, there is no video controller with its own character map: the character map is a software virtual map.

Video Attributes

Screen attributes control the presentation of the entire screen. The screen attributes are blank, half-bright, and reverse video.

Character attributes control the presentation of a single character. Character attributes can be present or absent, depending on the value of a screen attribute. If character attributes are present, then each character has an 8 bit character attribute field; 6 of the 8 bits in the character attribute field are used to specify the presence or absence of the attributes: blinking, bold, half-bright, reverse video, struck-through, and underline.

Font

You can create workstation fonts using one of the font design applications provided. For character map workstations, use the Font Designer; for bit map workstations, use the Raster Font and Icon Designer.

The font contains pixel information for all 256 characters. Character map workstations also support half-pixel shift in any pixel row of a character. This allows the Font Designer to maximize resolution.

Cursor

On a character map workstation, the standard cursor is a blinking underline and is not changeable by software. Bit map workstations have a software-loadable cursor. The cursor bit array is superimposed in the character.

VIDEO REFRESH

On character map workstations, the video RAM is contained within the processor module and is accessible to the processor at a fixed location in the processor's address space. The location of the character map cannot be changed. To switch screens, it is necessary to copy the contents of the character map.

WRITING PROGRAMS THAT RUN ON DIFFERENT WORKSTATION MODELS

Different workstation models have different numbers of lines on the screen. Therefore, care must be taken to write code that can run on a screen with a variable number of lines. This type of code can be written as follows.

During initialization, include a call to QueryVidHdw or QueryVideo. (For details on these operations, see Chapter 3, "Operations," in the CTOS/VM Reference Manual.) The memory address of a block of video information is returned. At offset 1 in this block is a 1 byte field called nLinesMax. This field contains the number of lines on the screen. The lines are numbered from 0 to n-1 (where n is equal to the nth line).

When writing calls to operations that require row and column coordinates (such as PutFrameChars or PutFrameAttrs), the row coordinate should be used as a variable rather than as a constant.

For example, to write a message on the line of the screen that is 2 from the bottom, the row coordinate used is nLinesMax-3.

SYSTEM DATA STRUCTURES: THE VIDEO CONTROL BLOCK AND FRAME DESCRIPTOR

The Video Control Block (VCB) contains all information known to the operating system about the video display, including the location, height, and width of each frame, and the coordinates at which the next character is to be stored in the frame by SAM. You can obtain the memory address of the VCB by calling the GetpStructure operation with a structCode value of 2. (GetpStructure is described in Chapter 3, "Operations," in the CTOS/VM Reference Manual. See Table 4-32 in that same manual for the format of the VCB.)

The VCB contains an array of frame descriptors. A frame descriptor is a component of the VCB and contains all information known about one of the frames. The number of frame descriptors in the VCB is specified at system build. (See Table 4-13 in the CTOS/VM Reference Manual for the content of a frame descriptor.)

COLOR GRAPHICS ATTRIBUTE PROCEDURES

Alphanumeric color procedures are available on color monitor workstations. Character attributes such as blinking, half-bright., reverse video, and underlining are ordinarily under hardware control through the alphanumeric style RAM. The graphics control board has an alternate style RAM that enables eight different attribute combinations to be used on a screen.

The graphics style RAM includes color and intensity specification with reverse video and underlining. Blinking cannot be specified with this style RAM.

An 8 byte memory work area is allocated to specify the entries that are passed to the graphics style RAM. Each byte uses the low-order 6 bits for the color specification and the high-order 2 bits for reverse video and underlining, respectively.

If you want to use color in your programs or if you want to program the graphics control registers, you must use the ProgramColorMapper operation. (For details and examples of how this is done, see the CTOS/VM Reference Manual, Appendix F, "Using Color.")

OPERATIONS

The video operations described below are categorized by software function. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

VAM OPERATIONS

PutFrameChars — Overwrites the specified character positions in the specified frame with the specified text string.

PutFrameAttrs — Establishes the same character attribute for a range of character positions within a specified frame.

PutFrameCharsandAttrs — Combines the PutFrameChars and PutFrameAttrs functions so that a sequence of characters can be written in a single call.

QueryFrameChar — Returns a single character located in the character map at the specified coordinates of the specified frame.

QueryFrameCharsandAttrs — Returns a character string and its associated attributes from the character map at the specified coordinates.

PosFrameCursor — Establishes a visible cursor within the specified frame at the specified coordinates.

QueryFrameCursor
: Returns the cursor position for the specified frame.

ScrollFrame
: Scrolls the specified portion of the specified frame up or down by the specified number of lines.

MoveFrameRectangle
: Moves an arbitrary rectangle of characters and corresponding attributes within a frame of the character map to another position in the map.

QueryFrameBounds
: Returns the size in number of columns and lines for the specified frame.

VDM OPERATIONS

QueryVidHdw
: Places information describing the level of video capability of the workstation in the specified memory area. QueryVidHdw fills in only certain fields in the specified memory area according to the operating system version.

QueryVideo
: Performs the same function as QueryVidHdw except QueryVideo fills in all fields in the specified memory area.

LoadFontRam
: Reads the character font from the specified open file to the specified memory area and then transfers the font to the font RAM.

ResetVideo	Suspends video refresh, resets all screen attributes, and changes the values stored in the VCB to reflect the specified parameters.
ResetFrame	Restores the frame to its initial state,- that is, all character positions are blanked and all character attributes are reset.
InitVidFrame	Defines the screen coordinates and dimensions of one of the frames.
SetScreenVidAttr	Sets/resets a specified screen attribute.
InitCharMap	Initializes the character map.
SetVideoTimeOut	Causes the screen refresh to turn off after a specified time has elapsed during which no keyboard activity has occurred.
QueryFrameBounds	Returns the size in number of columns and lines for the specified frame.

COLOR PROGRAMMING OPERATIONS

ProgramColorMapper
: Sets and queries the palette and or control structure.

SetAlphaColorDefault
: Sets up a default alpha palette and control structure.

LoadColorStyleRam
: Specifies 8 bytes that are passed to the color graphics style RAM. These attribute settings display different combinations of color, reverse video, and underlining.

SetStyleRam
: Sets a flag that indicates which of the following style RAMs is to be used: the graphics style RAM or the standard alphanumeric style RAM.

SetStyleRamEntry
: Modifies a single 1 byte entry in the graphics style RAM.

DIRECT ACCESS TO VIDEO DATA STRUCTURES OPERATIONS

It is possible, although not recommended, to access the video data structures directly at an interface level below VAM and VDM. Although programming at this lower level can be more efficient than using VAM or SAM, your program will not be compatible among the several workstation models. Specifically, it will not work on a bit map workstation.

The following operations provide direct access to the video data structures.

LockVideo — Locks the video structures used by the operating system.

UnLockVideo — Is used after calling LockVideo to remove a lock on the video structures used by the operating system.

LockVideoForModify — Modifies the video structures used by the operating system.

UnLockVideoForModify — Is used after LockVideoForModify is called to remove a lock on the video structures used by the operating system.

10 KEYBOARD MANAGEMENT

The Keyboard Management facility enables an application program to control the keyboard.

The keyboard microprocessor transmits each event of a sequence of pressed/released keys to keyboard management.

Although this chapter refers to the keys by the standard symbols engraved on them, the function of each key is completely under the control of the application program.

KEYBOARD MODES

The application program can request input from the keyboard in either of two modes: unencoded or character.

In unencoded mode, the program receives an 8 bit keyboard code for each key depressed/released. For example, in the following sequence of pressed/released keys, the program would receive a keyboard code for each of the four key transitions:

1. Press **Shift**.

2. Press **A**.

3. Release **A**.

4. Release **Shift**.

The program also would receive a different keyboard code for the depression/release of the left Shift key than it would for the depression/release of the right Shift key.

Unencoded mode provides maximum flexibility. With unencoded mode, a program can, for example, use any key as a **Shift** key, provide a hierarchy of **Shift** keys, and make decisions based on how long a key remains pressed. These are only three of many possibilities. The Editor makes extensive use of the flexibility afforded by unencoded mode. (See the Editor Manual. Note especially the description of **Move** and **Copy**.)

In character mode, the program receives an 8 bit character code when a key other than **Shift**, **Code**, **Lock**, or **Action** is pressed.

In the same four-event key sequence described above (for unencoded mode), a program in encoded mode would receive only one character code, the code for uppercase A.

Character mode provides the program with the same kind of information as a traditional n-key rollover encoded keyboard. However, even character mode provides greater flexibility than an encoded keyboard. As keyboard management converts the sequence of keyboard codes to character codes, it accesses a keyboard mapping table to direct its translation.

KEYBOARD MAPPING TABLE

A keyboard mapping table maps keyboard codes to character codes. Keyboard mapping is implemented by the Keyboard Encoding Table included in the operating system at system build or by the NLS Keyboard Mapping Table loaded as part of the Nls.sys file. (For details on the NLS Keyboard Mapping Table and the Nls.sys file, see Chapter 40, "Native Language Support.")

To modify the built-in table, you must regenerate the operating system. (For details, see the CTOS System Administrator's Guide and the Release Notice for your version of the operating system.) The contents of the table loaded as part of Nls.sys can be modified dynamically. (For details, see Chapter 40, "Native Language Support.")

Modifying the Keyboard Encoding Table allows the keyboard to be customized without requiring the program to support the complexity of directly interpreting the unencoded keyboard.

SYSTEM INPUT PROCESS

Keyboard management is augmented by the system input process. The system input process permits all the characters typed at the keyboard to be recorded in a file, in addition to returning them to the application program requesting them. (Note that the application program must be in character mode.)

The file can be used as a record of all data typed by the user. The file also can be played back as a submit file, in which the sequence of characters it contains is substituted for characters typed at the keyboard. The use of submit files allows the convenient repetition of command sequences. A submit file might be used, for example, to run the sequence of programs necessary to produce end-of-month reports.

The Editor can be used to prepare a submit file containing the same sequence of characters that would be typed to the desired programs. When this submit file is activated by a request from a program or an Executive command, a character from the file is returned to the program whenever it requests a character from the keyboard. (Since the system input process always operates in character mode, this is not applicable to a program that uses the keyboard in unencoded mode.)

A submit file does not preclude direct access to the keyboard. The program can bypass an active submit file and read characters directly from the keyboard. This is necessary when the program needs confirmation that a physical action was performed. For example, if a submit file is used to produce a sequence of reports, the program needs to accept confirmation from the keyboard, rather than from the submit file, that the correct report forms are loaded into the printer.

When requesting a character, a program can specify that the character must come from the keyboard rather than the submit file. Also, a special sequence of characters (an escape sequence) in the submit file can cause input to be accepted temporarily directly from the keyboard. Pressing a special key causes the input source to revert to the submit file.

(For details, see "Using the System Input Process," later in this chapter.)

PHYSICAL KEYBOARD

The physical keyboard is shown in Figure 10-1. The keyboard includes special function keys and keys with LEDs. Application programs control some of the keyboard LEDs. In unencoded mode, application programs control the LED in the **Lock** key; in character mode, this LED is under the control of keyboard management.

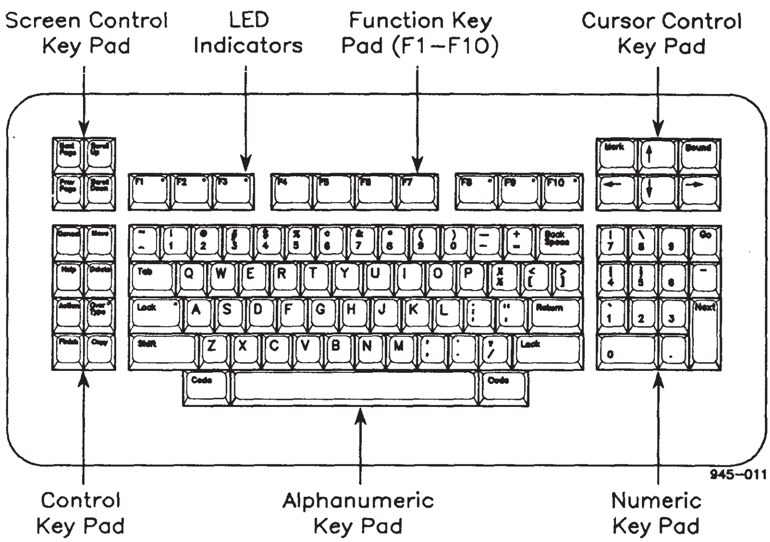


Figure 10-1. Keyboard

The keyboard microprocessor transmits each event of a sequence of pressed/released keys to keyboard management.

When a key is pressed or released, the keyboard microprocessor transmits a sequence of bytes to indicate all keys currently pressed.

Keyboard management memory retains which keys are pressed. When it receives a byte sequence from the keyboard microprocessor, it compares the keys currently reported as pressed to the ones it stored as pressed. The differences are the keys pressed/released. This information is represented in the keyboard code for each key.

USING THE KEYBOARD MODES

An SetKbdUnencodedMode operation can be used by an application program to specify the mode (character or unencoded) in which the ReadKbd and the ReadKbdDirect operations are to function.

UNENCODED MODE

In unencoded mode, the program receives the keyboard code returned by ReadKbd or ReadKbdDirect. The 7 low-order bits of the 8 bit keyboard code identify the key; the high-order bit is 0 to indicate key depression and 1 to indicate key release. (See the CTOS/VM Reference Manual, Appendix C, for the specific 7 bit code generated for each key of the physical keyboard.)

CHARACTER MODE

In character mode (the default mode) the program receives the character code returned by ReadKbd or ReadKbdDirect. The 8 bit character code signifies a key pressed other than **Shift**, **Code**, **Lock**, or **Action**. Pressing **Shift**, **Code**, or **Lock** does not generate a character code, but influences the character codes generated for other keys pressed simultaneously. **Action** has a special, system-wide meaning. (For details, see "Action Key," later in this chapter.)

TYPE-AHEAD BUFFER

Keyboard management provides a type-ahead buffer to store character codes (or keyboard codes, if in unencoded mode) not yet read by a program. If the user types too many characters before processing, the excess is discarded. When a program reads beyond the characters buffered successfully, it receives status code 610 ("Type-ahead buffer overflow"). The size of the type-ahead buffer is usually 128 characters but can be changed at system build. The content of the type-ahead buffer is discarded by

- SetKbdUnencodedMode, if the mode is actually changed.
- Chain and ErrorExit, if the status code is abnormal (nonzero). (For details, see "Application Program Termination," later in this chapter.)

ACTION KEY

Action is a special kind of **Shift** key; it is processed specially, even in unencoded mode. The interpretation of all other keys is modified while **Action** is pressed.

Key combinations that include **Action** are processed independently of calls by the program to ReadKbd or ReadKbdDirect and are not affected by character or keyboard codes stored in the type-ahead buffer.

The key combination **Action-Delete** clears the type-ahead buffer.

The key combination **Action-Overtype** blanks out the screen. It does not affect any ongoing activity, but simply makes the screen blank. To reactivate the video display, press any nonediting key, such as **Shift** or **Code**.

The key combination **Action-Finish** terminates the execution of the current program and invokes the exit run file. The DisableActionFinish operation disables this feature.

The key combinations **Action-A** and **Action-B** invoke the Debugger if the Debugger is included in the operating system at system build.

Key combinations that include **Action** are available for program interpretation. Pressing **Action** in conjunction with any other key causes the keyboard code for that key to be stored in keyboard management memory. The keyboard code (also called an action code) can be obtained by calling ReadActionCode or ReadActionKbd. Calling either of these operations avoids changing modes to obtain this information, thereby allowing the type-ahead buffer to continue while the program tests for special user intervention.

The BASIC interpreter, for example, uses **Action-Cancel** to interrupt computation without interfering with type-ahead. The Context Manager uses **Action-Go**, **Action-Next**, and **Action-F1** to **Action-F10** for switching from one context (user number) to another.

ReadActionKbd can be called to determine immediately if an **Action** key sequence is used. Typically, ReadActionKbd is used asynchronously. (For details on the asynchronous use of requests, see Chapter 29, "Interprocess Communication.")

KEYBOARD AND VIDEO INDEPENDENCE

Keyboard management does not automatically echo characters to the video device. A program can assign various functions to each character and can select whether or not to echo the characters. Keyboard management attaches no special significance to keys such as **Finish**, **Help**, **Return**, or **Delete**. **Action** is the only key with special significance.

USING THE KEYBOARD ENCODING TABLE

The Keyboard Encoding Table translates keyboard codes to character codes. The table provides translation of the following:

- the character code to generate if **Shift** is pressed
- whether Lock has the effect of **Shift** for this key
- whether the key is typematic (repeats)
- the initial delay before beginning typematic repeating
- the frequency of typematic repeating
- whether a key responds to diacritical key handling

Diacritical key handling is useful for displaying characters with diacritical marks, such as the German a with an umlaut. The first key of a diacritical key pair enables diacritical mode; the second key displays the diacritical result. Any of the character codes can be assigned diacritical key handling.

You can use either of two methods to set up diacritical key handling. You can modify the built-in keyboard table, which requires regenerating the operating system; or (an easier method) you can edit the Keyboard Mapping table in the Nls.sys file and rebootstrap your system. (For details, see Chapter 40, "Native Language Support.")

The Keyboard Encoding Table provides an 8 bit superset of the ASCII printable characters. (See the Standard Character Set in Appendix B in the CTOS/VM Reference Manual.) All 256 8 bit character codes can be generated from the keyboard. Each of the first 128 character codes (and some of the second 128) can be generated either by pressing a single key or by pressing **Shift** while pressing another key. Pressing **Code** while pressing another key causes the high-order bit to be set (80h to be inclusive ORed) in the character code that would otherwise be generated. Thus, the use of **Code** (or **Code** and **Shift**) permits the generation of the remainder of the 256 character codes.

USING THE SYSTEM INPUT PROCESS

The system input process permits all the characters typed at the keyboard to be recorded in a file, in addition to returning them to the application program requesting them. The application program must be in unencoded mode.

The system input process provides for three modes of operation: normal, recording, and submit.

- In submit mode, input is read from the submit (recorded) file rather than from the keyboard.

- In recording mode, a copy of the keyboard input is written to a recording file.
- In normal mode, neither recording mode nor submit mode is active.

The system input process is shown in Figure 10-2.

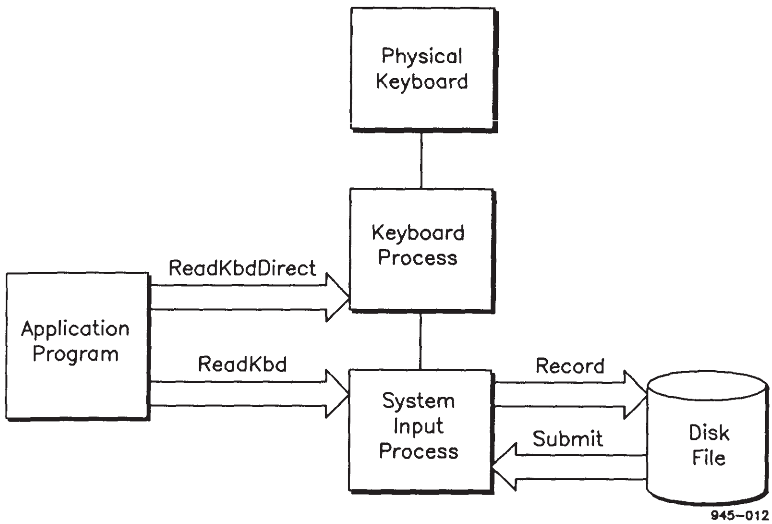


Figure 10-2. System Input Process

SUBMIT FILE MODE

In submit mode, input is read from the submit (recorded) file rather than from the keyboard. Submit files can provide the convenience of automatically repeating command sequences.

To activate a submit file, SetSysInMode can be called by an application program or through an Executive command. (For details, see SetSysInMode in the CTOS/VM Reference Manual, Chapter 3 "Operations.")

A submit file remains active until

- all characters in the file are read
- an end-of-file escape sequence is read
- SetSysInMode is called again

Calling the ReadKbd operation while a submit file is active causes a character to be read from the file and returned to the calling program. After all characters are read from the submit file, it is automatically closed. Subsequent calls to ReadKbd cause characters to be read directly from the keyboard. Transition of input source from submit file to keyboard is totally transparent to the application program. If, however, a program needs to know whether a submit file is active, the QueryKbdState operation can be called to provide this information.

A submit file can be disabled temporarily by the SetKbdUnencodedMode operation or by a read-direct escape sequence. (See "Submit File Escape Sequences," later in this chapter, for details on the read-direct escape sequence.)

The system input process is not available to application programs that use the keyboard in unencoded mode. This is because, in unencoded mode, the ReadKbd operation reads keyboard codes from the keyboard, not the submit file. Calling SetKbdUnencodedMode with an fOn parameter value of FALSE, however, sets character mode again and reactivates the submit file. Subsequent characters thus are read from the submit file.

The ReadKbdDirect operation is available to read from the keyboard at all times, regardless of whether a submit file is active.

The submit file is disabled temporarily when a read-direct escape sequence is read from the submit file. (See "Submit File Escape Sequences," later in this chapter for, details.)

RECORDING MODE

SetSysInMode can specify recording mode. When recording mode is activated, all characters typed at the keyboard and read in character mode by ReadKbd (but not by ReadKbdDirect) are written to a recording file, in addition to being returned to the application program calling ReadKbd. (Note that **Action** keys are not recorded.)

A recording file can be used later as a submit file to repeat the same sequence of input characters. A recording file and a submit file cannot be active simultaneously.

SUBMIT FILE ESCAPE SEQUENCES

Certain sequences of characters (escape sequences) invoke special functions when read from a submit file. A submit file escape sequence consists of two or three characters.

- The first character of the escape sequence is the character code 03h (¢), which indicates the presence of an escape sequence.
- The second is a code to identify the special function.
- The third character, if present, is an argument to the special function.

The permitted codes are shown in Table 10-1. Additional escape sequences are used by the **Submit** command. (See the Executive Manual for details.)

Table 10-1
PERMITTED CODES IN SUBMIT FILE ESCAPE SEQUENCES

Character	Code	Function
¢	03h	A two-character escape sequence that represents the character code 03h. Since 03h (¢) is used to introduce escape sequences, this escape sequence (that is, two consecutive ¢'s) is the only way to represent the ¢ in a submit file.
1	31h	A three-character, read-direct escape sequence. (See the discussion following this table.)
2	32h	An end-of-file escape sequence. When this two-character escape sequence is read during a ReadKbd operation, the submit file is closed. The current and subsequent ReadKbd operations read characters directly from the keyboard. (This escape sequence is meaningful only in submit files that were created through the Editor rather than as recording files.)

The read-direct escape sequence is a three-character submit file escape sequence that causes ReadKbd to read characters directly from the keyboard until a specified key is pressed. The third byte of the escape sequence specifies the key that is to terminate input from the keyboard. When the specified key is pressed, its keyboard code is not returned to the program. Rather, the current and all subsequent ReadKbd operations read characters from the submit file (unless another escape sequence redirects the input source).

For example, it is frequently useful to have the user enter data into a single field of an Executive command form during the operation of a submit file. (See the Executive Manual for details.) To accomplish this, the submit file should contain the following line of code:

.
.
.

data for the previous field

0Ah (**Return/Next**)

the 3 character escape sequence 03h, 31h, 0Ah ((¢, 1, **Return/Next**)

0Ah (**Return/Next**)

data for the next field

.
.
.

When the escape sequence is read from the submit file, the cursor is blinking in the leftmost character position of the field that is to be entered manually. The user then enters the selected data into the field and presses either Return or Next (symbolized by **Return**|**Next**). Pressing **Return**|**Next** resumes the execution of the submit file, but control is not returned to the program. The second **Return**|**Next** in the submit file ends the entry of data into the field and advances to the next field of the form.

As another example, it may be useful to have the user enter data into all the fields of a form during playback of the submit file. To accomplish this, include the four characters

03h, 31h, 1Bh, 1Bh

in the submit file. This causes all characters except **Go** (1Bh) to be read from the keyboard. When the operator completes the form and presses **Go**, the **Go** read from the keyboard resumes the playing of the submit file. The **Go** in the submit file (the 1Bh following the three-character escape sequence) completes the processing of the form. (See the Executive Manual for details.)

APPLICATION PROGRAM TERMINATION

When an application program terminates (because of the Chain, Exit, or ErrorExit operations, or **Action-Finish**), termination has the following effects on keyboard management:

- If the keyboard was in unencoded mode, it is reset to character mode, and the content of the type-ahead buffer is discarded.
- The **Action-Finish** feature is reenabled.
- The action code, if any, is discarded.

If the program terminates abnormally (because of the Chain or ErrorExit operations with a nonzero status code, or **Action-Finish**), termination has the following additional effects:

- The content of the type-ahead buffer is discarded.
- The submit or recording file is closed.

Termination of the program does not affect the keyboard LEDs. The Executive, however, resets the LEDs when it is loaded.

THE MOUSE SYSTEM SERVICE

If the Mouse system service is installed, use the Mouse operation, ReadInputEvent, rather than ReadKbd or ReadKbdDirect for Mouse and keyboard input. (See the Mouse System Services Manual for details on the Mouse system service and the Mouse operations.)

OPERATIONS

The keyboard management operations described below are categorized by use. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

COMMONLY USED

ReadKbd	Reads one character from the keyboard, or from a submit (submit) file if one is active.
Beep	Activates an audio tone for .3 second.
SetKbdLed	Turns on/off one of the keyboard LEDs.
QueryKbdLeds	Returns the status (on/off) of the keyboard LEDs.

LESS FREQUENTLY USED

SetKbdUnencodedMode	Selects unencoded or character mode.
ReadKbdDirect	Reads one character code (or keyboard code, if in unencoded mode) from the keyboard.
DisableActionFinish	Disables operating system interpretation of **Action-Finish**.

SetSysInMode	Changes the state of the system input process.
CheckpointSysIn	Writes the content of the current, partially filled, output buffer to the recording file if the system input process is in recording mode.
QueryKbdState	Returns the status of the keyboard and of the system input process to a structure provided by the program.
ReadActionCode	Returns the action code, if any, and resets the indication that an action code is available.
ReadActionKbd	Detects **Action** key sequences.

11 FILE MANAGEMENT

The file management system provides a hierarchical organization of disk file data by node, volume, directory, and file. The operating system automatically recognizes a volume when you place it online (mount it). A file can have a 50 character file name, a 12 character password, and a file protection level. A file can be dynamically expanded and contracted without limit as long as it fits on one disk (1 gigabyte). Concurrent access is controlled by read (shared), peek (shared), and modify (exclusive) access modes.

While providing convenience and reliability, the file management system supplies you with the full throughput capability of the disk hardware. This includes reading or writing any 512 byte sector of an open file with one disk access, reading or writing up to 65K bytes (127 sectors) of an open file with one disk operation, overlapping I/O with process execution, and optimizing disk arm scheduling.

You can access files located at a cluster workstation that has local storage as well as files located at the master.

OVERVIEW OF FILE SYSTEM CAPABILITIES

EFFICIENCY

File system efficiency is provided through the following methods:

- Careful data placement: The operating system places the volume control structures, which are resident on each volume, at locations that minimize disk arm movement.

  The operating system brings the Volume Home Block into memory when you place a volume online. In addition, it retains the most recently used directory and file information in memory.

- Randomization (hashing) techniques: The operating system uses randomization techniques for placing an entry in a directory sector and later for locating the entry. These techniques reduce the number of disk reads required to access directory information.

RELIABILITY

Reliability is provided through the following features:

- Duplication of two volume control structures: the Volume Home Block and the File Header Blocks.

  This duplication ensures that damage to one copy of a volume control structure does not cause data loss.

- Ordered updating of volume structures: This ensures that the volume will not be corrupted by power failure, hardware malfunction, or software error.

- Multilevel (volume, directory, or file) password protection.

- Multiple file protection levels: A file protection level specifies the access allowed to a file when the program requesting access does not provide a valid volume or directory password.

- Optional volume encryption: You can optionally encrypt the passwords of all files and directories created on a volume. Volume encryption ensures that a file cannot be opened without a valid password.

CONVENIENCE

Convenience is provided through the following means:

- Hierarchical organization of disk file data by node, volume, directory, and file.

- Long file names (up to 50 characters).

- Dynamic file length: You can determine the file length when you create the file, and you can change file length later.

- Removable file volumes (floppy disks).

- Automatic recognition of volumes placed online: read (shared), peek (shared), or modify (exclusive) file modes.

- Device independence: The device on which a file is located is transparent to you.

STRUCTURED FILE ACCESS METHODS

Structured file access methods augment the file management system by providing additional structured access to disk file data. The structured file access methods are

- The Record Sequential Access Method. (See Chapter 22.)

- The Direct Access Method. (See Chapter 23.)

- The Indexed Sequential Access Method. (See the ISAM Manual.)

LOCAL FILE SYSTEM

A cluster workstation can have its own local file system. The local file system allows a cluster workstation to access files on its local disks as well as files on disks at the master. The operating system routes processing requests to either the local or master file system on the basis of file specifications or handles. (For details on routing requests, see Chapter 29, "Interprocess Communication.")

You can bootstrap a cluster workstation either from a file at the master or from the local file system. A cluster workstation bootstrapped from its local file system is a self-contained entity that accesses the master only for shared files. If a malfunction occurs at the master, the cluster workstation can continue to operate normally, provided all of the files you access are on your workstation's local disks.

An application program can access a master file system in the same way the program accesses a standalone workstation's local file system. A program that works on a standalone workstation will work correctly on a cluster workstation that accesses master files.

FILE SPECIFICATIONS

The file management system organizes disk file data hierarchically by node, volume, directory, file, and (optionally) password.

NODE

A system connected to CT-Net can access the files of other network nodes, subject to password protection. If the file you are requesting is not on your node, you must specify the different node when attempting to access the file.

A node name is a string of characters. It can have a maximum of 12 characters.

VOLUME

The files of the system are located on volumes. In the Executive, use the IVolume command to format and initialize a volume. (For details on IVolume, see the CTOS System Administrator's Guide.) You can protect a volume by a volume password and by volume encryption.

A floppy disk and the media sealed inside a hard disk drive are examples of volumes. A floppy disk is a removable volume.

Volume Name

A volname (volume name) is a string of characters. It can have a maximum of 12 characters.

System Volume

Sys is a mnemonic for the volume name of the disk from which the operating system was bootstrapped.

For example, in a hard disk system where the operating system was bootstrapped from hard disk drive 0, you can use Sys instead of its volume name.

In a cluster workstation without local disk storage, Sys is a synonym for the volume name of the disk on the master from which the workstation was bootstrapped.

!Sys signifies the volume name of the disk from which the master of the cluster was bootstrapped.

Scratch Volume

You can reference the volume on which scratch (temporary) files are placed either by its mnemonic, Scr, or by its real name. The volume to be used as the scratch volume (Sys by default) is determined at system build (SysGen). For protected mode, the scratch volume also can be determined by an entry in [Sys]<Sys>Config.sys. (For details, see the CTOS System Administrator's Guide.)

Volume Control Structures

A volume contains several volume control structures: the Volume Home Block, the File Header Blocks, and the Master File Directory, among others.

The Volume Home Block is the root structure of information for a disk volume.

The File Header Block of each file contains information about that file and about the disk address and size of each of its Disk Extents. (A Disk Extent is one or more contiguous disk sectors.)

The Master File Directory contains an entry for each directory on the volume. The directories provide fast access to the File Header Block of a specific file. They do not, however, contain any information about the file that is not also contained in its File Header Block.

Volume Home Blocks (working and initial copies) and File Header Blocks (primary and secondary copies) each have duplicates on the volume for reliability.

The location on the volume of the Volume Home Blocks, the File Header Blocks, and the other volume control structures minimizes disk arm movement and therefore maximizes efficiency. The File Header Blocks are located in a single area of the volume, the disk address and size of which are recorded in the working and initial copies of the Volume Home Block. Volume control structures that the operating system accesses frequently, including the primary and secondary copies of the File Header Blocks, are located near the middle of the disk.

DIRECTORY

The files of a volume are divided into one or more directories. A <u>directory</u> is a collection of related files on one volume. The maximum number of directories that you can create on a volume depends on the size of the Master File Directory, which you can specify when you initialize the volume. The maximum number of files that you can create in a directory depends on two factors:

- the directory size that you specified when you created the directory

- the length of all names of all files in that directory

A directory can be protected by a directory password.

You can create a directory with the CreateDir operation and delete it with the DeleteDir operation.

A <u>dirname</u> (directory name) is a string of characters. It can have a maximum of 12 characters.

FILE

A file is a set of bytes (on disk) that are treated as a unit. The files of a volume consist of integral numbers of 512 byte sectors and must be completely contained on one disk (1 gigabyte).

You can create a file with the CreateFile operation and delete it with the DeleteFile operation. Once you create a file, you can access it with the OpenFile operation and close it with the CloseFile operation.

The ChangeFileLength operation changes the length of an open file.

The RenameFile operation renames an existing file.

A file is protected by a file protection level and by an optional file password.

A filename (file name) is a string of characters. It can have a maximum of 50 characters.

PASSWORD

Four types of password protection are available:

- volume
- directory
- file
- device

A volume password protects a volume. A directory password protects a directory on a volume. A file password protects a file in a directory on a volume. A device password is used with operations that work directly with the disk.

You can specify a volume password at the time you initialize the volume using the **IVolume** command. Use the CreateDir operation to specify a directory password. You can specify a file password using the SetFileStatus operation.

Volume, directory, and file passwords can consist of all alphanumeric characters, plus the period (.) and the hyphen (-). A volume, directory, or file password can have a maximum of 12 characters.

You can access a file if you know its volume, directory, or file password. Knowing a volume password allows you to access all of the directories and files of that volume. Knowing a directory or file password permits access that is dependent on the file protection level specified for each file. (For details, see "File Protection," later in this chapter.)

The OpenFile operation accepts a single password. This password is compared first against the volume password, then against the directory password, and last against the file password (if one was specified). You are granted access to open the file if any of these comparisons matches provided the file protection level permits access. (For details, see "File Protection," later in this chapter.)

The CreateFile operation accepts a single password that authorizes you to create a file in the specified directory. It is not a password to be assigned to the file being created. This password is compared first against the volume password and then against the directory password. You are granted access to create the file if either of these comparisons matches. (The SetFileStatus operation assigns a password to the file being created. The CreateDir operation assigns a password to the directory being created.)

You can specify a default password using the SetPath operation. The default password is used whenever an explicit password is not specified to an operation. The default password, like an explicit one, is compared to the volume, directory, and file passwords.

Valid passwords are required for some Executive commands, such as **Backup Volume**, **IVolume**, and the **User File Editor**. If you fail to supply the password or supply an incorrect one, status code 219 ("Access denied") is returned.

The protection provided by each of the four password types is discussed in "Protection by Password," later in this chapter.

DIRECTORY AND FILE SPECIFICATIONS

You refer to a directory by a directory specification. A directory specification has the form

{node}[volname]dirname

You refer to a file by a file specification. A full file specification has the form

{node}[volname]<dirname>filename^password

The distinction between uppercase and lowercase in directory and file specifications is not significant in matching directory and/or file names during directory search; the distinction is, however, preserved by the file management system to make the directory and file specifications easier to read.

It is recommended that node names, volume names, and directory names consist only of alphanumeric characters, plus the period (.) and the hyphen (-). It is recommended that file names consist of alphanumeric characters, plus the period (.), the hyphen (-), and the right angle bracket (>).

ABBREVIATED SPECIFICATIONS

If you previously established a default specification, you can refer to a file or directory by an abbreviated specification.

The SetPath operation establishes a default node, a default volume, a default directory, and a default password. The SetPrefix operation establishes a default file prefix. SetPath and SetPrefix establish defaults for the user number of the caller.

If a program has issued the SetPath operation with the default volname of [MasterVol] and the default dirname of <Susan>, you can access the files

```
[MasterVol]<Susan>Todays>work
[MasterVol]<Susan>Yesterdays>work
```

as either

```
<Susan>Todays>work <Susan>Yesterdays>work
```

if just the volname is omitted, or

```
Todays>work Yesterdays>work
```

if the default volname and default dirname are omitted; <dirname> cannot be omitted unless [volname] is also omitted.

If a program has issued the SetPrefix operation with the default file prefix of Todays>, in addition to the default volname and dirname established by the SetPath operation above, you can access the files

[MasterVol]<Susan>Todays>work
[MasterVol]<Susan>Yesterdays>work

as

work

and

<Susan>Yesterdays>work

You could no longer specify the file in the last example above as

Yesterdays>work

because the file you accessed would be

[MasterVol]<Susan>Todays>Yesterdays>work

which is not the same file.

AUTOMATIC VOLUME RECOGNITION

The operating system automatically recognizes a volume that you place online (that is, mount). For example, when you insert a floppy disk into a disk drive, the operating system reads the disk to determine whether it contains a volume and, if it does, that no other volume of the same name is already online. After this validation by the operating system, the volume responds to your requests if they contain the appropriate specifications and passwords.

When you place a volume online, the operating system reads the Volume Home Block into memory. The Volume Home Block remains there as long as the volume remains online.

If you leave a floppy drive door open, any open files on the disk in that drive are automatically put into a special dismounted state. You can close such files in the usual manner, but if you attempt to perform other operations on them, status code 216 ("Wrong volume mounted") is returned.

FILE PROTECTION

The operating system offers a file-oriented security system.

Passwords control access to a specific device, volume, directory, or file. Protection levels assigned to each file define the type of access allowed. (For details, see "Protection by Protection Level," later in this chapter.)

Using passwords and protection levels together, you can define a file security system to meet your specific needs. Optionally, you can use volume encryption to ensure security of passwords of all directories and files created on that volume. (For details, see "Protection by Volume Encryption," later in this chapter.)

PROTECTION BY PASSWORD

The four password types are volume, directory, file, and device. The type of protection provided by each password is described below.

Volume Password

You can access any file, regardless of password or protection level, with the volume password. In the absence of a volume password, the system is not protected. The volume password overrides directory or file passwords. If a volume password exists, it is required for opening the volume as a device.

For example, if you sign on with the volume password, or enter it with the Executive **Path** command, the operating system gives you access to all files on that volume, whether they are password-protected or not, without additional directory or file passwords.

NOTE: You must have a volume password for directory or file passwords to take effect.

You assign a volume password when you create the volume using the **IVolume** command. You can change the password using the **Change Volume Name** command.

Directory Password

You can use a directory password to restrict file creation or file renaming within a directory. If a directory password exists, you must specify it or the volume password to create or rename any files within the directory. A directory or volume password is required to remove a directory. You can also use a directory password to access a file, unless a protection level that ignores directory passwords has been assigned to the file. (For details, see "Protection by Protection Level," later in this chapter.)

You can establish a directory password with the Executive **Create Directory** command.

Use the Executive **Set Directory Protection** command to change or remove a directory password.

File Password

You can use a file password to restrict access to a specific file. Access depends on the file protection level. (For details, see "Protection by Protection Level," later in this chapter.) Files do not have passwords when they are created.

To add a password to a previously unprotected file, or to change a file password, use the Executive **Set Protection** command.

File passwords are most often used to allow certain files in a directory to be read, without allowing access to the other files.

Device Password

You use a device password for operations that work directly with the disk, such as the **IVolume** or **Backup Volume** commands. The operating system assigns these passwords at system build. Unless you have a customized operating system, default passwords assigned with Standard Software apply. For the hard disk, the password is the same as the device name (for example, D0 or D1). For floppy disks, the default is no password.

Using a Password for Access

If you did not assign a volume password to the volume when you initialized it, you can sign on to the system without supplying a password and have full access to all files.

If a volume password was assigned, you can enter a volume, directory, or file password when you sign on. The SignOn password is used for access, which is restricted accordingly.

You can also use the Executive **Path** command to enter a password. Thus, if you signed on with a directory password and wish to access files in a different directory, you can supply the necessary password by using the **Path** command. Also, some Executive commands include parameter prompts for a password.

You can also enter a password as a part of a device, volume, directory, or file name. The password consists of the characters between a caret (^) and the end of the parameter or subparameter name, for example:

Example: filename^password

PROTECTION BY PROTECTION LEVEL

The operating system uses a file protection level to control which types of passwords you are required to supply, if any, to open a specific file in read, peek, or modify mode.

A protection level is assigned only to files. A directory has a default protection level. The default, however, is used to assign a protection level to each file at the time that the file is created.

How Protection Levels Work

Nine protection levels are available. Table 11-1 shows the name, number, and type of access allowed for each protection level. Note that protection level numbers are not hierarchical. Because the password requirements for opening a file in peek and read mode are equivalent, read mode is used to mean either of these modes in the following discussion.

As an example of how protection levels work, the file specified by

    [Sys]<MyDir>Foo

is assigned a protection level number of 23 (Nondirectory Modify Password). The Foo file, <MyDir>, and [Sys] are assigned passwords.

You can open the Foo file in Read mode without providing a password. You cannot, however, open the Foo file in modify mode by providing the password for <MyDir>. (Note in Table 11-1 that you must supply either the volume password or the file password to gain access to the Foo file in modify mode.)

Table 11-1

PROTECTION LEVELS

Protection Level	Level Number	Password Required (Read or Peek Mode)*	Password Required (Modify Mode)*
Unprotected	15	None	None
Modify Protected	5	None	Directory
Nondirectory Modify Password	23	None	File
Modify Password	7	None	Directory or file
Access Protected	0	Directory	Directory
Read Password	1	Directory or file	Directory
Nondirectory Access Password	19	File	Directory or file
Access Password	3	Directory or file	Directory or file
Nondirectory Password	51	File	File

*You can access any file with the volume password regardless of password or protection level.

The default file protection level does not affect the passwords and protection of the directory in any way. It is used only as a default level for files created within the directory. If, for example, a directory has a password and is assigned the lowest level of protection (15, unprotected), it is not totally unprotected since you are required to provide a directory or volume password to create or rename files within that directory. When created, files within that directory are assigned a protection level of 15 (unprotected). You can change the protection level with the Executive Set Directory Protection command.

How the Operating System Validates Protection Levels

The operating system validates that a file can be opened in read or modify mode. To do this, the operating system first checks if a volume password was provided to open the file. If a volume password was provided, it is compared with the assigned volume password, if any. A match grants access to the file with no further validation.

If, however, a volume password was not provided, the operating system checks the protection level number against a bit pattern. The bit pattern is described in Table 11-2.

Bit numbers 0 through 7:

7 6 5 4 3 2 1 0

designate the file protection level, as shown in the table. The operating system checks the meanings of these bits against the password information (directory password, file password, or none) supplied to open the file. If any of the bit checks is valid, the file can be opened. Otherwise, status code 219 ("Access denied") is returned.

Table 11-2

BIT NUMBER DESIGNATIONS FOR PROTECTION LEVEL

Bit 0:	If the value is 1 and if there is a file password, it is valid for opening in read mode (mr).
Bit 1:	If the value is 1 and if there is a file password, it is valid for opening in modify mode (mm).
Bit 2:	If the value is 1, no password is required for read mode (mr).
Bit 3:	If the value is. 1, no password is required for modify mode (mm).
Bit 4:	If the value is 1, a directory password is not valid for modify mode (mm).
Bit 5:	If the value is 1, a directory password is not valid for read mode (mr).
Bit 6:	Reserved for internal use.
Bit 7:	Reserved for internal use.

As an example, the file specified by

[Sys]<MyDir>Foo

is assigned protection level number 15.

15 (Fh) in binary form is

0 0 0 0 1 1 1 1

Bit numbers 2 and 3 are set. This means the Foo file can be opened in read or modify mode without a password. Note that this agrees with protection level "15 (unprotected) in Table 11-1.

As another example, if the Foo file above is assigned protection level number 51 (33h); in binary form, this is

0 0 1 1 0 0 1 1

In this case, bits 2 and 3 are 0. As a result, a directory or a file password is required to open the file in read or modify mode.

The operating system then checks for a password supplied to open Foo. It matches this password with the one assigned.

Because bits 4 and 5 are set, a matching directory password is not valid. Bits 0 and 1 also are set, however, indicating that a matching file password is valid for opening the file.

Note that the bit interpretations agree with protection level 51 (nondirectory password) in Table 11-1.

(For details on common system protection applications, see the CTOS System Administrator's Guide.)

PROTECTION BY VOLUME ENCRYPTION

You can use an IVolume command option to encrypt the passwords of all files and directories created on a volume. (For details on the IVolume, see the CTOS System Administrator's Guide.)

An encrypted password has the following characteristics:

- The password is 12 bytes (that is, 12 characters long).
- The high-order bit is set in the byte for the rightmost character.

Figure 11-1 compares the effects of volume encryption on operations that require passwords.

All passwords provided to the OpenFile operation are encrypted for an encrypted volume.

The GetFileStatus and GetDirStatus operations return encrypted file and directory passwords, respectively, for an encrypted volume.

Note that pressing **Code** in combination with another key results in setting the high-order bit of a byte. Using this key combination for the rightmost character of a 12 character password string is not recommended. This is because the SetFileStatus and SetDirStatus operations interpret such a password as encrypted for an encrypted volume. Access using this password would be denied in a future OpenFile operation. (See Figure 11-1.)

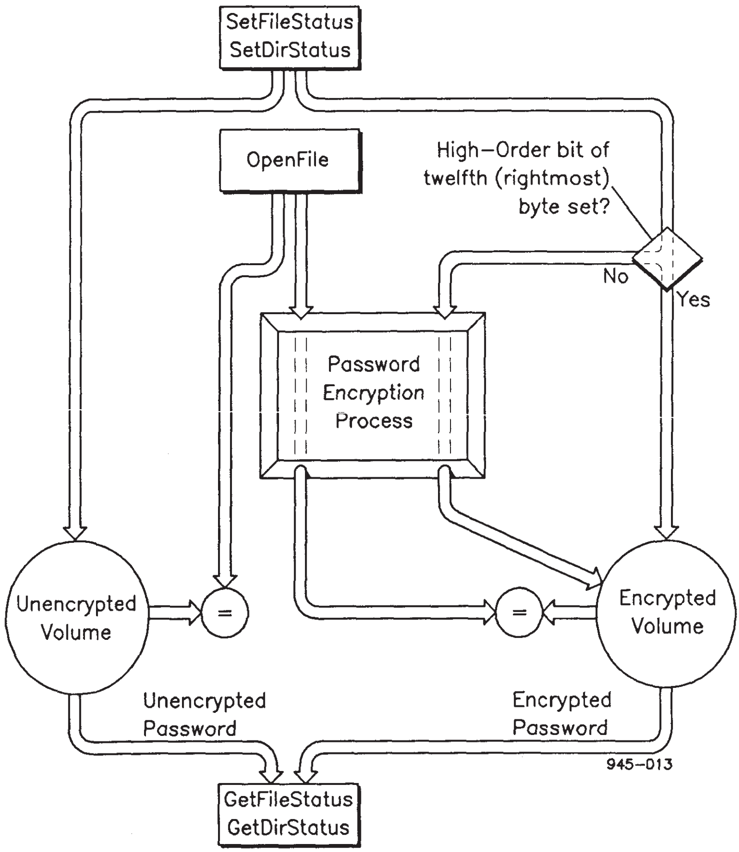


Figure 11-1. Effects of Volume Encryption

CREATING AND ACCESSING A FILE

PROGRAM INTERFACE LEVELS

You can create and access a file on a disk device using different interface levels. These are

- structured file access methods
- byte streams (Sequential Access Method)
- file management operations

Structured File Access Methods

The structured file access methods provide access to data files that are structured in specific ways. A chapter is dedicated to each of these methods in this manual. (For details, see Chapter 20, "Structured File Access Methods.")

Byte Streams

You can create and access disk files by using the Sequential Access Method (disk byte streams).

When you use disk byte streams, you are using the file management operations indirectly. The byte stream routines call the appropriate file management operations for you. You can write as many bytes as you want (provided you do not run out of disk space). When you close your file, the byte stream makes the appropriate calls to close the file.

Most programs use the byte stream interface level because it is a relatively easy and flexible way to create and to access files. (For details, see Chapter 7, "Sequential Access Method.")

File Management Operations

At the very lowest interface level (closest to the hardware), you can use the file management operations described in "Operations" at the end of this chapter. At this level, you have the greatest degree of control over the file you create. You can also use the Request and Wait primitives and build your own request block based on the request blocks for these operations.

The file management operations provide random access to 512-byte sectors of a file. (512 bytes is the size of a physical disk sector.) The operations allow you to read and write multiple sectors, starting with a particular sector of a file. Device independence is provided by masking the device characteristics of the disk on which the file is located. (Use of the file management operations is discussed in "Reading and Writing a File," later in this chapter.)

LOGICAL FILE ADDRESS

A logical file address (lfa) is a 32 bit unsigned integer that your program uses to locate a position within a file. It specifies a byte position; that is, it is the number (the offset) that would be assigned to a byte in a file if all the bytes were numbered consecutively starting with 0.

You use the lfa in file management operations (such as Read or Write) to locate a particular sector of a file. The lfa must be on a sector boundary. Therefore, you must supply an lfa (in bytes) to a Read or a Write operation that is a multiple of 512. For example, to locate the third sector in a file, you would supply an lfa of 1024.

If you are using byte streams, however, you are not required to provide an lfa that is a 512 byte multiple.

The 2 high-order bits of the lfa are reserved as special indicators. Bit 31 is set to override normal system checks and is used to attempt access to defective disks. Bit 30 is set to suppress retry of input or output to recover from errors. For example, a program logging high-speed, digitized wave forms that could accept badly written data but not the time required for retry, would specify an lfa of 40000400h to specify the third sector of a file with error retry suppressed. The returned status code reports errors in the normal way even when the special indicators are set.

FILE HANDLE

A <u>file handle</u> (fh) is a 16 bit integer that uniquely identifies an open file. It is returned by the OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

A file handle can be long-lived or short-lived. You can use the OpenFileLL or SetFhLongevity operation to set a file handle long-lived. Only a short-lived (normal) file handle is closed by a CloseAllFiles operation or automatically when an application program terminates. A long-lived, as well as a short-lived, file handle is closed by an explicit CloseFile operation or by the CloseAllFilesLL operation.

PERFORMING I/O

To perform I/O to a disk file with the file management operations, perform the following sequence of steps:

1. Create the file.

2. Open the file.

3. Write data to the file and subsequently read the data.

4. Close the file.

Each step of this sequence is described below. A comparable description is given for what happens when you use byte streams.

Creating a File

What You Do to Create a File. To create a file using the file management operations, you need to call CreateFile. You can specify the length of your file as a multiple of 512 bytes, or you can specify 0 bytes. If you specify 0 bytes, you must make a subsequent call to the ChangeFileLength operation to specify the file length.

The CreateFile and the ChangeFileLength operations are the only operations that allocate disk sectors for a file. ChangeFileLength can allocate or deallocate sectors. The operating system uses the byte value you specified to determine the number of 512 byte sectors to allocate for your file.

When you use the byte streams interface, the byte stream automatically calls the CreateFile operation. A byte stream always creates a 30 sector file, expanding the file in 30 sector increments, as required. When the file is closed, file size is contracted to the end of the sector containing the If a of the last byte written (end-of-file pointer).

What the Operating System Does to Create a File. The operating system performs the following steps when you call the operations CreateFile and ChangeFileLength. The operating system

1. Verifies that a volume of the requested name is already online. (The Volume Home Block is brought into memory when a volume is placed online.)

2. Verifies that a directory of the requested name is on that volume. (The most recently used directory information is retained in memory.)

3. Verifies that a file of the requested name does not exist in that directory. (The most recently used file information is retained in memory.)

4. Allocates a File Header Block and assigns the requested number of disk sectors by consulting the Allocation Bit Map. (The <u>Allocation</u> <u>Bit</u> <u>Map</u> controls the assignment of disk sectors. For details, see "Volume Control Structures," later in this chapter.)

5. Inserts an entry for the file in the requested directory.

Opening a File

What You Do to Open a File. To open a file using the file management operations, you call the OpenFile or the OpenFileLL operation. In either case, you supply the file specification, the password (if required), and the file mode. A file handle is returned to your program that you can use in future requests (such as Write or Read) to the opened file.

Note that the byte stream's interface opens the file for you when you open the byte stream.

What the Operating System Does to Open a File. When you open a file, the operating system

1. Verifies that a volume of the requested name is already online. (The Volume Home Block is brought into memory when a volume is placed online.)

2. Verifies that a directory of the requested name is on that volume. (The most recently used directory information is retained in memory.)

3. Verifies that a file of the requested name is in that directory. (The most recently used file information is retained in memory.)

4. Allocates a File Control Block, one or more File Area Blocks, and the memory address of the File Control Block (FCB) in the User File Block (UFB). (For details on these structures, see "System Data Structures," later in this chapter.)

5. Copies the information from the File Header Block to the File Control Block and one or more File Area Blocks.

6. Returns a file handle. The file handle serves to identify this particular File Control Block.

Reading and Writing a File

Using the File Management Operations. You can select to read from and write to a file in three ways when you use the file management operations. These are

- Using the Read and Write operations. The Read and Write operations are the simplest way of performing I/O, because constructing a request block and issuing the Request and Wait primitives are done automatically. Read and Write do not provide for any overlap between I/O operations and computation.

- Using the ReadAsync and CheckReadAsync and WriteAsync and CheckWriteAsync operations. The ReadAsync and WriteAsync operations are a more complex way of performing I/O. They allow a program to initiate an I/O transfer and then compute and/or initiate other I/O transfers before checking (with the CheckReadAsync and CheckWriteAsync operations) for the successful completion of the first transfer.

- Constructing a request block and using the Request and Wait (or Check) primitives. This is the most direct method of reading and writing a file. It also requires the most effort on your part. This method allows your program to overlap multiple I/O operations and computation.

(See Chapter 29, "Interprocess Communication," for details on the Request, Wait, and Check Kernel primitives.)

When you write to the file, you must specify where in the file your data is to be written. You can write full sectors only. However, you can write to any byte offset in the file that is a multiple of 512 (beginning of a sector).

If you write more data than can be contained within the number of sectors allocated, you must allocate more sectors by calling ChangeFileLength and supplying the new file length.

If you write to fewer sectors than you created, you can call ChangeFileLength to change the file length to a new shorter length.

Your program, however, may require the unused sectors as temporary space for holding variable amounts of data at different times. In such a case, it would be to your advantage to retain the extra sectors. If you anticipate frequent changes to the file length, you should consider the following:

- Each time you change the sector length of your file, the operating system has to allocate or deallocate sectors and consult its Allocation Bit Map. (For details on the Allocation Bit Map, see "Volume Control Structures," later in this chapter.)

- Frequent changes to the Allocation Bit Map fragment the disk space.

If you plan to use your disk file as input to a program that uses byte streams, you must call the SetFileStatus operation to specify the logical file address of the last byte you wrote (end-of-file pointer). SetFileStatus is used to set the end-of-file pointer only. To allocate additional sectors, you must use the ChangeFileLength operation.

Using Byte Streams. When you write to a file using the byte stream interface, you can write any number of bytes (versus being restricted to multiples of 512). The operating system writes your data sequentially to the disk.

When you append data to the file using byte streams, the data is written where the previous data ended.

Random access using byte streams is not as efficient as it is when using the file management operations. This is because you do not have as much control over the amount of data being read.

Closing a File

Using the File Management Operations. When you have completed the processing of a file, you close it using the operations CloseFile, CloseAllFiles, or CloseFilesLL. The number of 512 byte sectors allocated for the file is not changed.

If, for example, you had written 512 bytes and the file length that you specified to your last ChangeFileLength operation was 1024 bytes, the file length will remain 1024 bytes when you close the file.

Using Byte Streams. When you close the byte stream, the end-of-file pointer is set automatically. The byte stream adjusts the number of allocated file sectors to the minimum required to contain your file data.

If, for example, you closed a file containing 612 bytes of data, the byte stream calls SetFileStatus to set the end-of-file pointer within the second sector. ChangeFileLength then is called to deallocate the unused 28 sectors.

LOCAL FILE SYSTEM

When the operating system intercepts a request to open a file, it routes the request to the local file system. If the volume is not found, it routes the request to the master.

You can route a file access request explicitly to the master by including the special exclamation point character (!) before the volume specification, as in [!Sys]<Sys>Exec.Run, for example.

Any cluster workstation can access files on disks at the master. However, you cannot access files on a local disk from the master or from other cluster workstations. You must copy a local file to the master if it is to be processed by the master, another workstation in the cluster, or another node.

You must copy a local file to the master before it can be processed by any of the following:

- spooler (if the Generic Print System is not in use)
- remote job entry (RJE)

- indexed sequential access method (ISAM)
- any system service executing at the master or another cluster workstation

A cluster workstation bootstrapped from its local file system is a self-contained entity that must access the master only for shared files. If a malfunction occurs at the master, the cluster workstation can continue to operate normally provided all file accesses are to local disks.

LFSTOMASTER

LfsToMaster is a system configuration file option that provides for sharing master run files with cluster workstations. (For details on configuration file options, see the CTOS System Administrator's Guide.)

LfsToMaster results in certain requests for opening a file that fail locally to be retried at the master. The request is retried if all of the following conditions are TRUE:

- The request is an OpenFile, OpenFileLL, or ReOpenFile operation opened in read or peek (shared) mode.
- The status code returned is 203 ("No such file").
- The request originated at a cluster workstation with a local file system.
- The file specification is of the form [Sys]<Sys>Filename.

To specify the local file system (and thereby override the default of routing the request to the master), use [+Sys]<Sys>Filename as the file string.

VOLUME CONTROL STRUCTURES

A disk volume contains volume control structures. Volume control structures allow the file management system to manage (allocate, deallocate, locate, avoid duplication of) the space on the volume not already allocated to the volume control structures themselves.

Volume control structures are created when the disk is first initialized. Initialization must be performed using the **IVolume** command. (For details, see the CTOS System Administrator's Guide.)

The volume control structures include the

- Volume Home Block
- File Header Blocks
- Master File Directory
- directories
- Allocation Bit Map

The primary and secondary copies of the File Header Block are located on different cylinders and at different rotational positions and are accessed (except for floppy disks) by different read/write heads. These duplicates ensure that damage to one copy does not cause a data loss. The **IVolume** command permits suppression of duplicate File Header Blocks. However, this reduces reliability and is not recommended.

The initial copy, unlike the working copy, of the Volume Home Block, is not modified after it is created. The primary and secondary copies of the File Header Block, however, are always true duplicates.

VOLUME HOME BLOCK

Each volume is assigned a Volume Home Block. The Volume Home Block (VHB) is the root structure (that is, the starting point for the tree structure) of the information on the disk volume.

For example, the VHB contains the volume name and the date it was created. The VHB also contains the memory addresses of the Allocation Bit Map, the Bad Sector File, the File Header Blocks, the Master File Directory, the System Image, the Crash Dump Area, and the Log File. The VHB is one sector in size.

(The VHB structure is shown in Table 4-33 in the CTOS/VM Reference Manual.)

ALLOCATION BIT MAP AND BAD SECTOR FILE

The Allocation Bit Map controls the assignment of disk sectors. It has 1 bit for every sector on the disk, and the bit is set if the sector is available. The size of the Allocation Bit Map depends on the size of the volume.

The operating system places an entry for each unusable disk sector in the Bad Sector File. The Bad Sector File is one or more sector(s) in size.

FILE HEADER BLOCK

Each file is assigned a File Header Block (FHB). The FHB contains information about the file such as its name, password, protection level, the date/time it was created, the date/time it was last modified, and the disk address and size of each of its disk extents. The FHB is one sector in size.

(The FHB structure is shown in Table 4-12 in the CTOS/VM Reference Manual.)

DISK EXTENT

A Disk Extent is one or more contiguous disk sectors that compose all or part of a file. The entry for a Disk Extent in the FHB is 8 bytes: 4 bytes specify its location, and 4 bytes specify its size.

The operating system allocates a File Area Block (FAB) for each Disk Extent of an open file.

EXTENSION FILE HEADER BLOCK

A FHB can accommodate 32 Disk Extents. A file that contains more requires extension File Header Blocks (extension FHBs). Extension FHBs are seldom necessary unless you place an unusually heavy burden on the file management system. Your file may require extension FHBs, for example, if you expand the same file many times or fragment the available disk space by deleting and creating files frequently on a nearly full volume you seldomly refresh. (You can refresh a volume by using the **Backup Volume**, **IVolume**, and **Restore** commands. See the CTOS System Administrator's Guide for details on these commands.)

MASTER FILE DIRECTORY AND DIRECTORIES

Each directory on a volume, including the Sys directory (see below), has an entry in the Master File Directory (MFD). The entry's position within the MFD is determined by randomization (hashing) techniques. The entry contains the directory's name, password, location, and size.

(See Table 4-7 in the CTOS/VM Reference Manual for the format of a directory entry in the MFD.)

Each directory on the volume consists of one or more directory sectors. Randomization (hashing) techniques determine the directory sector in which a file is entered. The entry contains the file's name and a pointer to the FHB.

(See Table 4-8 in the CTOS/VM Reference Manual for the format of a file entry in a directory sector.)

The MFD and the directories provide fast access to the File Header Block of a specific file. They do not, however, contain any information about the file that is not also contained in its FHB. (The most recently used file and directory information is retained in memory.)

SYSTEM DIRECTORY

The Sys Directory is different from other directories in two ways. First, when a volume is initialized, its MFD contains only one entry, which is for the Sys Directory. (You can create other directories by using the CreateDir operation.) Second, the Sys Directory contains entries for all system files. You must not delete, rename, or overwrite these files.

These file entries are always present in the Sys Directory of each volume:

- the MFD (Mfd.Sys)
- the FHB (FileHeaders.Sys)

SYSTEM DATA STRUCTURES

System data structures are data areas contained within the operating system and are necessary for its operation. They are often configuration-dependent. The six system data structures related to the file management system are the

- User Control Block (UCB)
- File Control Block (FCB)
- File Area Block (FAB)
- Device Control Block (DCB)
- I/O Block (IOB)
- Volume Home Block (VHB)

The UCB and the DCB are user-accessible and are described below.

USER CONTROL BLOCK

A user number is associated with the resources allocated to an application partition.

Each user number is assigned a UCB. The UCB contains the default node, default volume, default directory, default password, and default file prefix set by the last SetPath and SetPrefix operations.

(The UCB structure is shown in Table 4-30 in the CTOS/VM Reference Manual.)

An incomplete file specification is expanded by the

- Net Agent at the master, before the Net Agent routes the request to a remote node in the net
- Cluster Agent before the Cluster Agent routes the request to the master
- local file system
- Kernel on a sending processor board in an SRP during inter-CPU communication (ICC)

(For details on request routing, see Chapter 29, "Interprocess Communication." For details on ICC, see Chapter 30, "Inter-CPU Communication.")

DEVICE CONTROL BLOCK

Each physical device is assigned a DCB. The DCB contains information, generated at system build, about the device. For a disk, the information includes how many tracks are on a disk, the number of sectors per track, and so forth. The DCB contains the memory address of a chain of I/O Blocks.

(The DCB structure is shown in Table 4-6 in the CTOS/VM Reference Manual.)

WILD CARD OPERATIONS

A wild card is a special character in a file specification. It instructs the Executive program to search for file specifications that match all characters given in the file specification except the wild card character(s).

The Executive recognizes the asterisk (*) and the question mark (?) as wild card characters. (For details on wild card characters, see the Executive Manual.)

Two wild card operations can be used with files. These are

- WildCardInit
- WildCardMatch

You can use these operations to build a list of files that match a wild card specification.

To do this, call WildCardInit with a wild card specification. Then build a loop, calling WildCardNext. Each time WildCardNext returns to your program, it returns to the next file name that matches the wild card specification.

The Executive, for example, uses these operations to expand wild cards that the Executive user types into a form.

$ DIRECTORY

The <$> directory is a disk directory in which programs can create temporary files. A <$> directory is required by all application programs and is needed for the maximum number of users connected to the master.

When a request with the directory name of <$> is given as part of a file specification, the operating system expands the directory name to the form

<$000>nnnnn>

where nnnnn is the user number associated with the application partition. This expansion occurs only if the directory name is <$>.

If, for example, user number 3 requests access to the foo file on the [Sys] volume using the directory name <$>, the file specification is expanded as follows:

[Sys]<$>foo   to [Sys]<$000>00003>foo

Since the user number(s) of a cluster workstation are reassigned whenever the system is boot-strapped, you should not use the <$> directory for permanent files.

OPERATIONS

The file management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

BASIC

OpenFile	Opens an already existing file, and returns a file handle.
Read	Transfers an integral number of 512 byte sectors from disk to memory.
Write	Transfers an integral number of 512 byte sectors from memory to disk.
CloseFile	Closes an open file.
CloseAllFiles	Closes all files that are currently open for the user, except those marked long-lived.

BASIC UTILITY OPERATIONS

CreateFile	Creates a file of the specified name in the specified directory on the specified volume.
DeleteFile	Deletes an open file.
RenameFile	Changes the file name and/or the directory name of an existing file. A file can be renamed to another directory on the same volume.

FILE ATTRIBUTES

ChangeFileLength	Expands or contracts an open file to a new length.
GetFileStatus	Copies the requested status information to the specified area.
SetFileStatus	Copies the specified status information from the specified memory area to the FHB.

DEFAULT PATH

ClearPath	Clears the defaults established by the SetPath and SetPrefix operations.
SetPath	Establishes a default volume, a default directory, and a default password.
SetPrefix	Establishes a default file prefix that begins the file name part of a file specification if that file specification does not have an explicit volume name or directory name.
SetNode	Allows the specification of a node name to be used as part of the default path whenever a file specification is given that does not contain a node name or volume name.
GetUCB	Copies the UCB for the current user number to the specified area.

DIRECTORIES

WildCardInit	Establishes a wild-carded file specification to be used by successive calls to the related WildCardNext operation.
WildCardNext	Returns the next file name that matches a wild-carded file specification supplied previously by a call to WildCardInit.
CreateDir	Creates a directory of the specified name on the specified volume.
DeleteDir	Deletes an empty directory.
ReadDirSector	Reads a 512 byte sector of the specified directory.
GetDirStatus	Determines information about a directory.
SetDirStatus	Changes a directory password or default file protection level.

LONG-LIVED FILES

OpenFileLL	Opens an already existing file and returns a file handle marked long-lived.
SetFhLongevity	Sets how long a file handle is to survive.
GetFhLongevity	Copies the requested information on the longevity of the file handle to the specified area.

CloseAllFilesLL
Closes all files that are currently open for the user, including those marked long-lived.

FILE HANDLE OPERATIONS

ChangeOpenMode — Changes the access mode of a file that is already open.

RemakeFh — When given an existing file handle, creates a new file handle to be associated with the user number of the process issuing this request.

ReopenFile — Is similar to OpenFile except that, if a file handle already exists for that file for the issuing user number, that handle rather than a new one is returned.

ASYNCHRONOUS FILE I/O

ReadAsync — Initiates the transfer of an integral number of 512 byte sectors from disk to memory. The procedure CheckReadAsync must be called to check the completion status of the transfer.

WriteAsync — Initiates the transfer of an integral number of 512 byte sectors from memory to disk. The procedure CheckWriteAsync must be called to check the completion status of the transfer.

CheckReadAsync

Waits for input completion, checks the status code, and obtains the byte count of data read after a ReadAsync procedure.

CheckWriteAsync

Waits for output completion, checks the status code, and obtains the byte count of data written after a WriteAsync procedure.

VOLUME DATA STRUCTURES

GetVHB

Copies the VHB of the specified device to the specified memory area.

12 DISK MANAGEMENT

Disk management operations provide device-level access to disk devices, in contrast to the file-level access provided by file management operations. Access to a disk device at such a level is necessary to read a floppy disk written on a non-Convergent system or to format an uninitialized disk.

Device-level access is provided to the following media:

- single or dual sided, 5 1/4 inch floppy disks written in double density
- all varieties of hard disks

The sector size and density of a floppy disk, if other than 512-byte double density, must be specified with the SetDevParams operation. (For a complete description of SetDevParams, see the CTOS/VM Reference Manual, Chapter 3, "Operations.")

ACCESSING A DISK DEVICE

A device can be accessed by using an OpenFile operation with a device or volume specification. The Read, Write, ReadAsync and CheckRead Async, WriteAsync and CheckWriteAsync, and CloseFile operations all accept a file handle returned by such an OpenFile operation. (For details on file handles, see Chapter 11, "File Management," and Chapter 29, "Interprocess Communication.")

Device-level access to disks bypasses the concurrency control of the file management system. Thus extreme care is required if device-level access is used in a cluster configuration.

DEVICE SPECIFICATION AND DEVICE PASSWORD

A disk device is a physical hardware entity. Access to a device requires presentation of a device specification and a password. A device specification can take either of two forms, depending on whether the medium of the disk device contains a valid file system.

If a volume contains a valid file system, the device specification has the form

{node}[volname]

In this case, the volume password must be specified. Volume passwords are described in Chapter 11, "File Management."

If, however, the medium does not contain a valid file system (either because the medium was never initialized to contain one or because the file system has become malformed), the device specification has the form

{node}[devname]

In this case, the device password must be specified. A device password protects a device. It can have a maximum of 12 characters, consisting of all alphanumeric characters plus the period (.) and the hyphen (-).

A volname (volume name) or a devname (device name) is a string of characters. A volname or devname can have a maximum of 12 characters, consisting of all alphanumeric characters, plus the period (.) and the hyphen (-).

OPERATIONS

The disk management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

SetDevParams	Allows the characteristics of the floppy disk controller to be modified.
QueryDCB	Copies the Device Control Block (DCB) of the specified device to the specified memory area.
Format	Initializes the surface of a floppy disk or other disk media to accommodate fixed-size data sectors. Format is used by the **IVolume** command.
MountVolume	Mounts the volume on the specified disk drive.
DismountVolume	Dismounts the specified volume,

13 PRINTING MANAGEMENT

The Printing Management facility provides a Generic Print System (GPS) to route output to the printer. If GPS is installed, it takes precedence over pre-GPS printing. (For details on pre-GPS printing, see Appendix A, "Spooler Management.")

COMPONENTS

GPS consists of the following dynamically installed system services:

- Routing Switch
- Device Driver
- Spooler
- Font Service

The above services and the Queue Manager (described in Chapter 35, "Queues and Queue Management") work together to control printing and to handle communication between the application program, the operating system, and the installed printing devices.

GPS is a separate program from CTOS/VM and, as such, is covered comprehensively in separate manuals. (For installation details, see the Printing Guide; for programming information, see the Generic Print System Programmer's Guide.)

INTERFACE CONSIDERATIONS

You can choose to request output to a GPS printing device through any of the following interfaces:

- Sequential Access Method (SAM). In accessing SAM directly, you sidestep GPAM's controls but retain device-independence. From a programmer's view-point, SAM is the simplest way to print a document. You specify the GPS printer name, and the GPS system services handle all aspects of printing for you. (For details, see Chapter 7, "Sequential Access Method.")

- Generic Print Access Method (GPAM). GPAM is a device-independent means of including complex text formatting, such as boldface, text, or graphics, to your output with a minimum of programming effort. GPAM sends its control information to the printing device through SAM. (For details, see Chapter 19, "Generic Print Access Method.")

- Direct GPS request. At the direct interface level, your program becomes GPS-dependent. This is not the recommended method.

14 COMMUNICATIONS PROGRAMMING

This chapter describes communications programming at the device-dependent and the device-independent interface levels.

- At the device-dependent level, communications byte streams (SamC) consists of the device-dependent interfaces of the Sequential Access Method (SAM). These interfaces provide greater control through a variety of operations specific to communications needs. SamC is the standard way to access RS-23 2-C ports in asynchronous mode.

- At the device-independent level, SAM allows your program to send I/O to a variety of devices. Using communication byte streams at this level is described briefly in this chapter for comparison purposes. (For details, see Chapter 7, "Sequential Access Method.")

WHAT SAMC IS USED FOR

SamC is the RS-232-C device-dependent portion of SAM. It is the standard operating system driver for RS-232-C ports (in asynchronous mode). This includes the use of ports for terminals, modems, and serial printers, as well as direct inter-CPU connection.

Using the standard RS-23 2-C driver frees the applications programmer from having to write interrupt handlers (described in Chapter 36, "Interrupt Handlers"), buffer management procedures, serial controller chip initialization sequences, and other low-level software.

SamC is intended to be flexible enough to do anything you might need to do with a serial port, except synchronous RS-232-C communication, which is not supported. You can use SamC indirectly, as part of SAM, which preserves device independence (the ability to perform I/O on SamC or a disk file interchangeably, for example). Alternatively, for special needs, you can call SamC directly using its device-dependent interfaces. (SAM does not provide access to all of these interfaces.)

WHAT PROGRAMS USE SAMC

Programs that accept or internally generate operating system file specifications beginning with [COMM] or [PTR] use SamC. SamC is linked with the program's run file.

Clients of SamC include the Executive **Copy** command and the spooler (for serial printers).

WHAT PROGRAMS CANNOT USE SAMC

Programs based on a synchronous RS-232-C communications protocol cannot use SamC. Such programs must interface directly with the operating system at a lower level. (For details, see Chapter 15, "Serial Port Management.")

USING SAMC AT THE DEVICE-INDEPENDENT INTERFACE LEVEL

SAM allows you to access communication ports from your program at the level of OpenByteStream, ReadBsRecord, WriteBsRecord, and the other device-independent byte stream operations described in Chapter 7, "Sequential Access Method." To use the device-independent SAM operations, you must specify a device in your OpenByteStream call. (For a list of the device specifications, see Chapter 7, "Sequential Access Method.")

SAM can be configured to include or exclude support for particular devices. Each device type has a corresponding byte stream. You can choose your own subset of the byte stream types, depending upon your needs and memory requirements.

To use SamC through SAM, it is necessary to have a configuration file for each communications channel. The configuration file specifies options for devices attached to the channel. As an example, separate transmission/receive baud rates may be required. You can use the default configuration file, or you can use the **Create Configuration File** utility to create or edit configuration files. (For details, see the **Create Configuration File** utility in the CTOS System Administrator's Guide.)

The configuration file supports parallel printer, serial printer, and communications configurations. SamC handles serial printer ([PTR]) and communications ([COMM]) configurations. You can open both kinds of configuration files with [COMM] or [PTR] device specifications.

(See "Communications Programming" in the CTOS Programmer's Guide for details. Also see "Building a Customized SAM" in the same manual for information on how to customize SAM.)

USING SAMC AT THE DEVICE-DEPENDENT INTERFACE LEVEL

The device-dependent interfaces of SamC itself (as distinct from SAM of which it is a part) provide a more powerful and flexible set of services than those available at the level of SAM.

Programs that are distinctly communications oriented (as opposed to programs such as the Executive, which merely use SamC through SAM as it would any other type of byte stream) can take advantage of the SamC services.

SamC also supports operations that are not appropriate for other byte stream types. Programs may supplement SAM by occasionally using SamC interfaces.

Although more complex to use than SAM, SamC comprises a complete set of services and can act as a replacement for SAM (provided communications byte streams and no other device types need be supported). Used in this fashion, SamC is a general-purpose device driver for asynchronous RS-232-C communications. It can form the heart of virtually any communications product except those that use synchronous communications protocols. Both half- and full-duplex communications are supported efficiently with a variety of line control and data editing options. Among other conveniences, using SamC frees you from writing interrupt handlers. (Writing interrupt handlers is described in Chapter 36, "Interrupt Handlers.")

THE SamC OPERATIONS

Asynchronous Interface

Because SamC is a subroutine package, you cannot issue asynchronous requests to it as you can with disk or keyboard byte streams, for example. (Asynchronous requests allow the caller to continue executing rather than wait at an exchange.) For this reason, asynchronous variants of the synchronous interfaces are provided, as follows:

Asynchronous	**Synchronous**
FillBufferAsyncC	FillBufferC
FlushBufferAsyncC	FlushBufferC
CheckPointBsAsyncC	CheckPointBsC,

Some applications require using asynchronous interfaces. MS-DOS, for example, must be able to initiate FillBufferC (communications input) and FlushBufferC (communications output) operations without the possibility of waiting as a side-effect.

The asynchronous operations include additional parameter options that allow the caller to specify what SamC should do if it needs to wait before the operation can be completed. As an example, one option provides using the PSend Kernel primitive to send a message to a caller-specified exchange when completion becomes possible. (PSend and other Kernel primitives for sending messages are described in detail in Chapter 29, "Interprocess Communication.")

FillBufferAsyncC provides a way to check the Byte Stream Work Area (BSWA) contents for input without waiting, if no input is there. In the past, SamC users often peeked into the BSWA to see if input characters were waiting. Doing so required knowledge of the BSWA, communications byte stream's private control structure. This is not recommended, however, because the BSWA contents change from release to release.

The AcquireByteStreamC Operation (Low-Level Open)

The OpenByteStream and OpenByteStreamC operations require a configuration file containing the communications line configuration parameters (baud rate and so on). AcquireByteStreamC is a lower-level interface that accepts an in-memory structure corresponding to the configuration file contents. Applications, such as CT-MAIL, use this interface to open SamC channels, thus avoiding an actual configuration file on disk. (For details, see the discussion on "Avoiding Configuration Files" in "Communications Programming" in the CTOS Programmer's Guide.)

AcquireByteStreamC also provides for greater control over the buffer sizes chosen for the receive and transmit queues. Under OpenByteStreamC, the caller supplies a single memory area of a chosen size, which OpenByteStreamC divides up between receive and transmit queues, according to its needs.

Dynamically Changing Parameters

SamC provides a way to query or change configuration parameters without closing and reopening the byte stream. CT-MAIL uses this feature to change the baud rate and other parameters without closing the byte stream (and thereby disconnecting an attached modem).

Querying and Setting Status Lines

The RS-232-C standard defines additional status lines that are not used by SamC but may be significant when dealing with modems or special hardware. Communications byte streams provide an interface to access or, where appropriate, to change the state of these lines.

The CheckForOperatorRestartC Operation

The Spooler periodically can call the CheckForOperatorRestartC operation to support auto restart on printers. This feature makes it possible for the spooler to restart output in response to

- an operator pressing the Break switch on the printer
- an operator opening and then closing the printer cover (on a printer with no Break switch)

OPERATIONS

The SamC operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

OpenByteStreamC*
: Opens a ([COMM] or [PTR]) byte stream device-specific to an RS-232-C serial port.

AcquireByteStreamC
: Is a substitute for OpenByteStreamC that does not require a configuration file on disk and offers more flexibility.

FillBufferC*
: Reads characters from the receive queue that have been received at the serial port.

FillBufferAsyncC
: Is the asynchronous form of FillBufferC.

FlushBufferC*
: Writes characters to the transmit queue, where they will be output to the serial port.

FlushBufferAsyncC
: Is the asynchronous form of FlushBufferC.

---

*This operation is the communications byte stream variant of a device-dependent SAM operation. (See Chapter 8, "Device-Dependent SAM," for details.)

DiscardInputBsC
: Discards any characters in the receive queue.

DiscardOutputBsC
: Discards any characters in the transmit queue.

SetImageModeC*
: Sets normal, image, or binary mode for [Comm] and [Ptr] byte streams device-specific to RS-232-C serial ports.

ReadByteStreamParameterC
: Reports the current value of the specified communications line parameter.

WriteByteStreamParameterC
: Modifies the value of the specified communications parameter.

ReadStatusC
: Reads the values of the specified status bits.

WriteStatusC
: Writes to the specified communications lines status bits, changing the condition of the corresponding status lines.

CheckForOperatorRestartC
: Checks for an operator signal to restart the printer.

---

*This operation is the communications byte stream variant of a device-dependent SAM operation. (See Chapter 8, "Device-Dependent SAM," for details.)

SendBreakC — Sends a break signal on the communications line previously opened under the Sequential Access Method.

CheckPointBsC* — Waits until all characters previously written to the byte stream have been physically output from the serial port.

CheckPointBsAsyncC — Asynchronous form of CheckPointBsC that can be used to perform the CheckPointBsC function when the caller does not want its process to wait.

ReleaseByteStreamC* — Stops all receive and transmit operations on a serial byte stream, making the serial port available for use by other users again.

---

*This operation is the communications byte stream variant of a device-dependent SAM operation. (See Chapter 8, "Device-Dependent SAM," for details.)

15 SERIAL PORT MANAGEMENT

This chapter describes communications programming at the serial port interface level. This is a level below SamC, which is described in Chapter 14.

ACCESS BELOW THE BYTE STREAM LEVEL (CommLine)

SamC does not support the serial controller in synchronous mode. To write a program that uses a synchronous communication protocol, it is necessary to interface directly with the operating system at a level below SamC.

The following operations are part of the operating system's support for serial ports:

- InitCommLine
- ResetCommLine
- ChangeCommLineBaudRate
- TerminateCommLine
- ReadCommLineStatus
- WriteCommLineStatus

These operations are used by SamC itself. They are not to be used by clients of SamC.

The serial port operations accomplish three objectives:

- Workstation-independent programs do not require relinking for each new hardware type.

- Raw interrupt handlers are compatible in protected mode. (See Chapter 36, "Interrupt Handlers," for details on raw interrupt handlers.)

- The operations are compatible with the SRP.

(For details on how to use these operations to write programs and to convert old programs to use the InitCommLine interface, see "Communications Programming" in the CTOS Programmer's Guide.)

All Convergent synchronous RS-232-C communications products that do not use the SamC level of interface use InitCommLine. These products do not incorporate into their software any specific knowledge of different port addresses, clock frequencies, and so on that are peculiar to different machines. A single run file for each of these products runs on all types of hardware, including all workstations and the SRP CP and TP boards.

SERIAL PORT OPERATIONS

SERIAL PORT REQUESTS

InitCommLine

InitCommLine assigns the caller to a physical channel on any RS-232-C serial port communications controller. InitCommLine does this by parsing the device specification passed to it. (For a list of device specifications, see Chapter 7, "Sequential Access Method.")

Your program should treat this specification as an uninterpreted string, so that your program continues to work when new hardware modules (with new forms of file specifications) are introduced. InitCommLine returns two port addresses, a control port and a data port, for the channel.

Note that InitCommLine does not tell the caller which half of the communications controller it is on. (Each controller has two channels, A and B.) This distinction is not necessary to a program. The controller does have certain operations that are always performed on channel A or channel B but affect both channels. InitCommLine performs these functions for you. For example, the serial communications controller is reset after an interrupt. (For details on interrupts, see Chapter 36, "Interrupt Handlers.")

The user still must perform some operations directly on the channel, using the two port addresses. InitCommLine does not even fully initialize the channel (although it does reset it), since it is not provided all of the initialization parameters. Note that the only parameters supplied to InitCommLine are those dealing with external hardware (outside the serial controller). This hardware (baud rate timers and external control registers) is InitCommLine's responsibility because it varies from machine to machine. The controller, however, is invariant: All Convergent machines use the same (or software-equivalent) serial controller-type chips.

The SRP TP has 8274 and 8251 serial controllers. Only the two 8274 devices (four serial ports in all) are supported by InitCommLine. The 8251 devices are not supported.

ResetCommLine

You cannot issue ResetCommLine, or any other operation, until you have successfully completed an InitCommLine operation for that channel. The argument to ResetCommLine is a handle returned by InitCommLine.

InitCommLine acquires the channel for you (and resets it so you have a chance to initialize it to your specifications before you start taking interrupts). ResetCommLine gives the channel back to the operating system, making it available for other users and freeing you from the responsibility for servicing interrupts from it. Thus, InitCommLine and ResetCommLine are logical parentheses, like OpenFile and CloseFile, for a serial port.

ChangeCommLineBaudRate

ChangeCommLineBaudRate is used to change InitCommLine's baud rate parameters dynamically.

SamC uses this interface to implement its WriteByteStreamParameterC call. SamC clients should use WriteByteStreamParameterC to modify the baud rate(s) dynamically. They should not use ChangeCommLineBaudRate directly.

The serial controller is not affected by ChangeCommLineBaudRate.

SERIAL PORT SYSTEM-COMMON PROCEDURES

These operations are procedures rather than requests so that it is possible to call them from inside an interrupt handler. (For details on interrupt handlers, see Chapter 36, "Interrupt Handlers.")

ReadCommLineStatus

This procedure allows certain RS-232-C signals, whose function is not defined by the serial controller, to be queried by the application program in machine-independent fashion.

SamC uses this interface to implement its ReadStatusC call. ReadStatusC is the way communications byte stream clients should query the status lines. They should not use ReadCommLineStatus directly.

WriteCommLineStatus

This procedure allows certain RS-232-C signals, whose function is not defined by the serial controller, to be raised or lowered by the application program in machine-independent fashion.

SamC uses this interface to implement its WriteStatusC call. WriteStatusC is the way communications byte stream clients should set or clear the status lines. They should not use WriteCommLineStatus directly.

OPERATIONS

The serial port operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

InitCommLine	Allocates a serial port to the user and specifies how interrupts from the port will be serviced.
ReadCommLineStatus	Reads values to the specified status bits.
WriteCommLineStatus	Writes to the specified status bits, changing the condition of the corresponding status lines.
ChangeCommLineBaudRate	Reinitializes the specified baud rate timer(s)
ResetCommLine	Makes the specified serial port available for use again.
LockIn	Allows a program to read from the serial I/O port. LockIn is essential on certain types of workstation hardware because of the timing functions it performs.
Lockout	Allows a program to write to the serial I/O port. Lockout is essential on certain types of workstation hardware because of the timing functions it performs.

16 PARALLEL PORT MANAGEMENT

This chapter describes the interfaces to a Centronics-compatible device that connects to a parallel port.

The parallel port operations below are variants of device-dependent SAM operations. (See Chapter 8, "Device-Dependent SAM," for details.) Lp, the name of the parallel port device, is appended to the generic prefix in each operation name:

- OpenByteStreamLp
- FlushBufferLp
- CheckPointBsLp
- ReleaseByteStreamLp

You can use these operations directly. They allow you to open a parallel port printer byte stream and to perform I/O to that byte stream at the level closest to the hardware.

If, however, you open a byte stream using the device-independent OpenByteStream operation, and you specify [LPT] as your device string, OpenByteStream automatically maps to OpenByteStreamLp. As another example, to send output to an open byte stream, you can call the device-independent operation, WriteBsRecord, which, in turn, maps to FlushBufferLp.

Chapter 8, "Device-Dependent SAM," lists the device-independent operations that map to each of the parallel port operations.

OPERATIONS

The parallel port operations described below are categorized by function. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

I/O

OpenByteStreamLp
: Opens a parallel port byte stream.

FlushBufferLp
: Writes output to the parallel port.

CheckPointBsLp
: Waits until the byte stream has been physically output from the parallel port.

ReleaseByteStreamLp
: Stops all receive and transmit operations on a parallel port byte stream, making the port available for use by other users again.

INTERRUPT HANDLING

SetLpISR
: Establishes the printer interrupt service routine (PISR) to process interrupts generated by the parallel printer interface. (See Chapter 36, "Interrupt Handlers," for details.)

17 SRP TERMINAL MANAGEMENT

The SRP terminal management operations are programming interfaces to Shared Resource Processor (SRP) terminals attached to all ports. However, there are certain ports that can be accessed only by these interfaces. Figure 17-1 shows the relationships of ports to access methods for the SRP and for workstations.

The communications programming operations described in Chapter 14 are at the same interface level as the SRP terminal management operations.

Figure 17-1 indicates that you can access the 8274 ports using either of these operation groups. You would most likely use the SRP terminal operations if you need to access the 8251 ports.

At a level farther away from the hardware, you can use the device-independent Sequential Access Method (SAM) operations to access all ports.

For details on the other interfaces illustrated, see

- Chapter 7, "Sequential Access Method"
- Chapter 14, "Communications Programming"
- Chapter 15, "Serial Port Management"

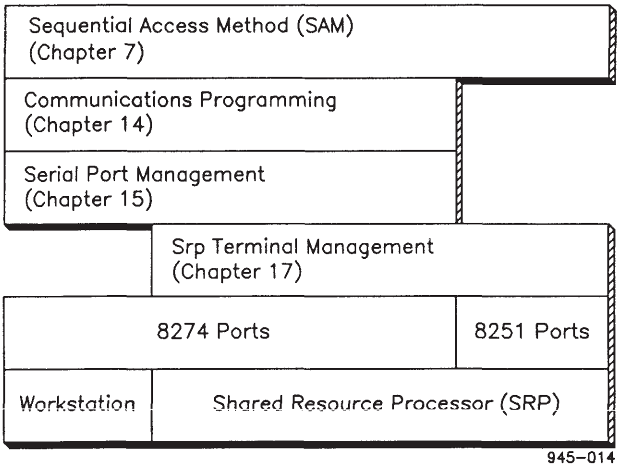


Figure 17-1. Ports/Access Methods Relationship

OPERATIONS

The SRP terminal management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

OpenTerminal	Initiates the use of a specified port on either a Cluster Processor (CP) or a Terminal Processor (TP), or initiates asynchronous RS-422 communications with a Programmable Terminal (PT) connected to a CP.
ReadTerminal	Reads data from a PT or from one of the asynchronous ports on a CP or a TP.
CloseTerminal	Indicates that the requesting process (client) is finished with a port.
SetTerminal	Performs out-of-band functions on a port.
WhereTerminalBuffer	Locates the terminal output buffer.
DrainTerminalOutput	Ensures an empty output buffer.

18 TAPE MANAGEMENT

Tape management provides you with the information you need if you are writing programs for quarter-inch cartridge (QIC) tape or half-inch tape.

You use the tape medium for storing data. If you are aware of the underlying software (and, in the case of QIC tape, hardware) principles of how tape works, tape can be much faster and more efficient than floppy disks for storage purposes. The tape utilities, such as **Tape Backup Volume**, **Tape Restore**, and **Tape Copy**, described in the CTOS System Administrator's Guide allow you to use tape through the Executive in an optimal way. They are sufficient for most users.

In some cases, you may want to write your own programs. This chapter describes the tape software available to you and provides you with the principles you need to know to write tape programs.

SOFTWARE REQUIREMENTS/INSTALLATION

All tape software is part of standard software. Tape software includes

- half-inch tape server for the Shared Resource Processor (SRP)
- QIC tape server for the SRP
- QIC tape server for workstations
- tape versions of various Executive utilities (described in the CTOS System Administrator's Guide)

INTERFACE LEVELS

You can write programs to a tape device at different interface levels.

BYTE STREAM LEVEL

At the byte stream level, you can use the Sequential Access Method (SAM) operations to send I/O to a tape byte stream. (See Chapter 7, "Sequential Access Method.")

You must link tape byte streams with your program by means of a special version of SamGen. (For details, see "Building a Customized SAM" in the CTOS Programmer's Guide.)

REQUEST LEVEL

At the request level, you can use the operations described in "Operations" at the end of this chapter. These operations provide greater program control over the tape hardware.

To request tape services, you can use the request procedural interface, or you can use the Request and Wait or Check Kernel primitives. (For details, see Chapter 29, "Interprocess Communication.")

In certain cases, such as when you are reading a foreign tape or a multicartridge QIC tape file, you must use Request and Check. (For details, see "Tape Byte Streams" and "Multicartridge QIC Tape File," later in this chapter.)

TAPE BYTE STREAMS

A tape byte stream is a set of procedures that reads a tape as a purely sequential sequence of bytes. It looks for the pattern of file marks that designate the beginning and end of a file. Within the limits specified by the tape configuration file, tape byte streams for half-inch tape ignore exact record and block sizes when reading.

The general concept of byte streams and their relation to the SAM is discussed in Chapter 7, "Sequential Access Method." The tape software is an example of the user-written, device-specific SAM object modules described in "Customizing the Sequential Access Method" in that chapter.

A half-inch tape byte stream interprets file mark pairs as meaning end of tape (EOT). For this reason, your program must use requests to the tape server to read a tape that uses file mark pairs in any other way.

TAPE FILES AND TAPE NAMING

A tape contains tape files. A tape file can span multiple QIC tape cartridges or half-inch tape reels.

Tape files differ from disk files in that they do not have file names and are not grouped into directories. They are identified by sequential numbers: 0, 1, 2, and so on. Also, a tape file can contain many disk files. For example, when you archive an entire hard disk using the **Tape Backup Volume** utility, all the files from that disk are placed in one long tape file.

As a result, tape names include numbers to indicate the QIC or half-inch tape drive and the tape file.

TAPE NAMES

Tape names are of the following formats:

Tape Type	Format
QIC tape	[QICm]n
Half-inch tape	[TAPEsd]n

where

n Is a number that indicates the position of the tape.

where

0 Is the beginning of the first file, which is always at the beginning of the tape.

1 Is the beginning of the second file, and so on.

+ Is the position after the last existing tape file; + is valid for opening a tape for writing only.

For QIC tape, if n is left blank, the first position on the tape is assumed.

For half-inch tape, if n is left blank, the current position on the tape is assumed; this is a convenient way to indicate "start at the next file."

NOTE: Be sure to make the distinction between multiple tape drives (which is discussed below) and multiple QIC tape cartridges or half-inch tape reels. The tape drive name does not indicate which of a series of tapes is referred to.

m Indicates the drive number for QIC tape (where m is 0 or 1).

On a workstation, the default is drive 0, which is the leftmost drive.

On the SRP, only one drive is available, so the drive number can be omitted.

s Indicates the number of the SRP board [Storage Processor (SP) or Data Processor (DP)] that controls the half-inch tape drive (0 for the first SP or DP board, 1 for the second SP or DP, 2 for the third, and so on to 7). The default is 0. (For details on board numbering, see the CTOS System Administrator's Guide.)

d Indicates the drive number for half-inch tape on the SRP (where d is in the range of 0 to 5.)

If you do not include the drive number, the drive directly connected to the SP board is assumed (that is, the first drive in the daisy chain).

EXAMPLES

Following are a few examples of tape names:

[QIC1]2 Is the third QIC tape file on the tape in the second (rightmost) QIC drive.

[QIC]0	Is the first QIC tape file on the tape in the first (leftmost) QIC drive.
[TAPE1]2	Is the third half-inch tape file on the tape in the second drive.
[TAPE]+	Is the position after the last tape file on the tape in the first drive.

(Tape names also are discussed in the CTOS System Administrator's Guide.)

QIC TAPE

FORMAT

Figure 18-1 shows the general format of a QIC tape. A QIC tape file is the data between tape marks. A tape mark can indicate either the logical end of a file or the logical end of tape (EOT).

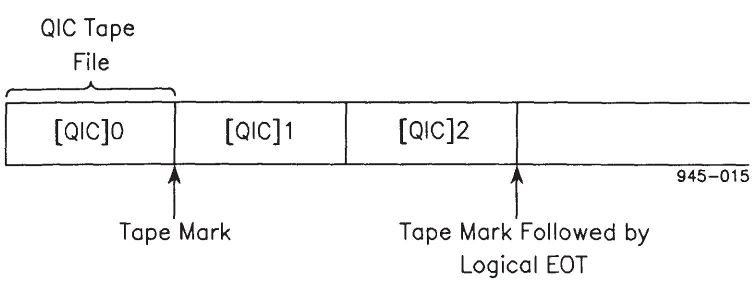


Figure 18-1. General QIC Tape Format

You can write to QIC tape at either of two positions: at the beginning of the tape or at the logical EOT (to append data).

Figure 18-2 shows the detail of a QIC tape file.

A QIC tape file contains a sequence of fixed-sized, 512-byte physical records or data blocks.

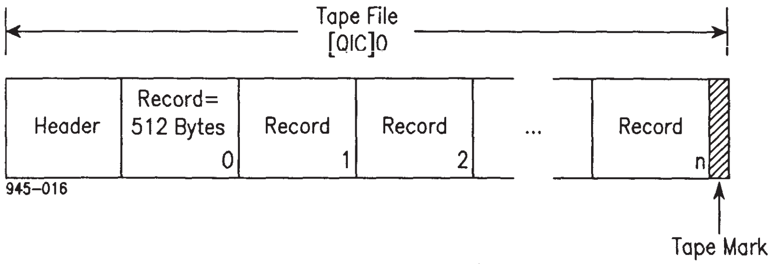


Figure 18-2. Detail of a QIC Tape File

The tape header has no fixed format. Its contents can vary with each application.

You can write approximately 2000 bytes in one WriteTapeRecords operation. (Note that the number of bytes can be changed by a SysGen. For details, see the CTOS System Administrator's Guide and the Release Notice for your version of the operating system.) Your data is written to the tape in fixed-sized, 512 byte records. If the data does not completely fill a record, the record is padded with 0s.

QIC tape provides no space between records; through data compression, it increases storage efficiency.

OPERATION

The QIC tape drive is a streaming-mode device. In streaming mode, the tape can move rapidly, without stopping between blocks. This mode makes QIC tape highly suitable for archiving, for example.

The QIC tape server maintains an internal buffer in which it houses buffered data that you supply when you call the WriteTapeRecords operation. When the QIC server's buffer is full, it writes the data out to tape in 512-byte records. If the server's buffer empties sooner than it is filled, the tape's movement becomes less efficient.

To maintain tape movement, the QIC server rewrites the last record again in anticipation of more data. If it does not receive another buffer of data, the hardware stops the tape.

Tape stopping and starting takes approximately 1 1/2 seconds. If this occurs frequently, QIC tape can be less efficient than other methods of data storage.

To maintain constant tape movement, your application program can use I/O buffers in the following ways:

- Use single, large buffers (up to 64K bytes) for I/O. If you are performing I/O to a multicartridge QIC tape file, however, you are restricted to a maximum buffer size of 1536 bytes. (See "Multicartridge QIC Tape File," later in this chapter, for details.)

- Use multiple buffers. To do this you must issue the Request and Wait or Check Kernel primitives. By doing so, your program provides a greater degree of overlap between multiple I/O operations and computation. Using multiple buffers ensures that the server's internal buffer is used to its maximum efficiency. (See Chapter 29, "Interprocess Communication," for details on the Request, Wait, and Check Kernel primitives.)

  A request, for example, can be issued with a 1536 byte buffer. This allows the client to issue a second request without waiting for the response from the first. When the response for the first returns, another request can be issued.

READING AND WRITING TO QIC TAPE

The QIC tape server can read from QIC tape at any valid position. Because data cannot be overwritten, however, you are not allowed to specify the name of an existing tape file to the OpenTape operation. You can specify either of two positions :

- the beginning of the tape (that is, [QIC]0), to erase the tape in advance of writing
- the logical EOT (that is, [QIC]+), to append data to the end of the tape

For details on tape naming, see "Tape Files and Tape Naming," earlier in this chapter and the CTOS System Administrator's Manual.

SINGLE-VOLUME QIC TAPE FILE

If your program is going to perform I/O to a single-volume tape file, you can use a single large buffer. You do not need to use any of the special tape operations described in "Multicartridge QIC Tape File," below.

MULTICARTRIDGE QIC TAPE FILE

Writing to Tape

If your program writes a quantity of data that will not fit on a single volume, you will need to check for the EOT. You must plan the sequence of operations carefully.

Each time the QIC server receives a buffer of information from a WriteTapeRecords operation, the server responds with a 0 status code (ercOK). The response, however, does not necessarily mean that the server has written the data to the tape; the server may have housed the information in its own internal buffer for a future write to the tape.

If your program provides no means of handling the contents of the QIC server's buffer at the EOT, the server will flush its buffer. As a result, neither you nor your program will have any way of accounting for lost data.

Reading from Tape

If your program reads a quantity of data spanning volumes, you will need to provide a means of verifying and reading the next tape(s).

Figure 18-3 shows the sequence of operations for performing I/O to a multicartridge QIC tape file.

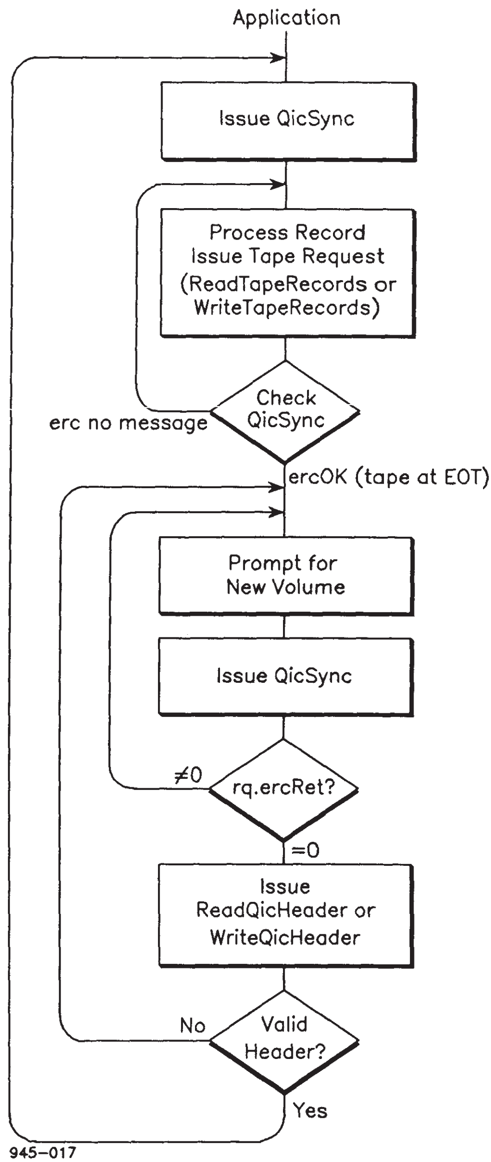


Figure 18-3. Multicartridge QIC Tape Operation Sequence

The following describes the sequence summarized in Figure 18-3:

1. The application program issues the QICSync operation immediately after it calls OpenTape.

2. The application processes a record and issues either a WriteTapeRecords or ReadTapeRecords request. It checks periodically for a response from the server.

3. The QIC server responds to QICSync at EOT.

4. The application prompts the user to insert the next tape.

5. The application calls QICSync. The server responds that the user has either inserted the tape (zero value in the ercRet field of the request block) or has not inserted the tape (nonzero value in ercRet).

   If ercRet is nonzero, steps 4 and 5 are repeated.

6. If the application is writing, the application calls the WriteQICHeader request. WriteQICHeader bypasses the server's internal buffer and writes the header information to the newly inserted tape. Upon completing WriteQICHeader, the server responds with a zero value in ercRet.

   - If the application determines that it wrote the header to the correct tape, the application calls QICSync again with a zero value in ercRet to inform the server that the tape header is valid. When the server responds, the application calls QICSync again to prepare the server for the next EOT (back to step 1). This completes one cycle of the sequence.

   - If, however, the application determines that it wrote the header to the correct tape, (user inserted the wrong tape), the application calls QICSync with a nonzero value in ercRet. The sequence is repeated from step 4.

7. If the application is reading, the application calls the ReadQICHeader request. Upon completing ReadQICHeader, the server responds with a zero value in ercRet.

   - If the application determines that the header it read (user inserted correct tape) is valid, the application calls QICSync again with a zero value in ercRet to inform the server. When the server responds, the application calls QICSync again to prepare the server for the next EOT (back to step 1). This completes one cycle of the sequence.

- If, however, the application determines that the header it read (user inserted the wrong tape) is invalid, the application calls QICSync with a nonzero value in ercRet. The sequence is repeated from step 4.

SPECIAL CARE FOR QIC TAPE

New tapes should always be retensioned before use. To retension the tape, use the **QicRetension** utility through the Executive. This utility winds the entire tape in one direction and then rewinds it. (See the CTOS System Administrator's Guide for details.)

The tape cartridge should be retensioned every 8 hours of normal use. When the tape drive is used extensively in start/stop mode, the cartridge should be retensioned once every 2 hours.

A tape cartridge that has been exposed to low temperatures (below 41°F or 5°C) or high temperatures (above 113°F or 45°C) for any length of time or a tape that has been stored unused for a long time should be retensioned before you try to read it or write to it again.

HALF-INCH TAPE

FORMAT

Figure 18-4 shows the general format of a half-inch tape. Each of the tape files represents data written at a different tape session. For example, one file could be the result of a Tape Backup Volume. A second file could be data from a user-written program.

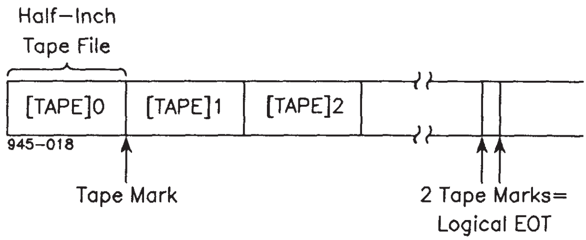


Figure 18-4. General Half-Inch Tape Format

A half-inch tape file consists of data between tape marks. A tape mark indicates either of two positions:

- the logical end of a file
- the logical EOT if the tape mark is followed immediately by a second mark

TapeOperation is used to write file marks at the end of a taping session. If your program appends data to the tape in a later write, the appended data overwrites (erases) one of the two file marks before the new data is written.

Figure 18-5 shows the details of a half-inch tape file.

A half-inch tape file contains a sequence of records or data blocks. The system leaves a space between records called the interrecord gap.

The tape header has no fixed format. Its contents can vary with each application.

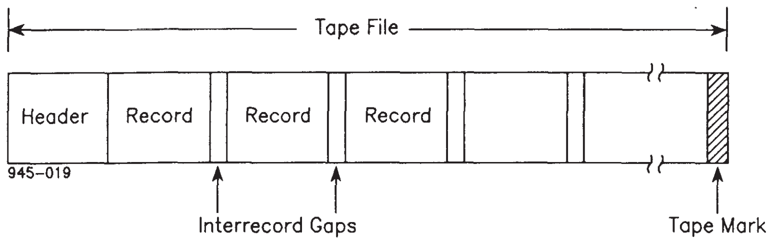


Figure 18-5. Detail of a Half-Inch Tape File

You can write records of any length within the minimum and maximum limits. (The maximum limit is determined by the buffer size specified during installation of the tape server.) You can, however, get more data on a single tape if you make the records large. (Also, it is faster for the server to write one large record than to write a series of smaller ones.) On the other hand, it is safer if you make records very small because, if any part of a record is damaged, all of the data on that record is lost. You must reach a compromise between these factors in deciding the size of a record.

OPERATION

The half-inch tape drive can operate in either start/stop or streaming mode.

In start/stop mode, the drive writes a record and then stops within the interrecord gap. This mode runs slowly to avoid damage to the tape.

In streaming mode, the drive runs the tape much more quickly and ramps slowly until it stops. It then backs up to a point considerably before the interrecord gap at the end of the file just written. When it is called upon to write again, it ramps up to speed and starts writing as it passes the end of the previously written file.

Considering the ramping and backing time, the effective tape speed is not actually streaming mode. Speed is significantly reduced if you are running a program that uses streaming mode from a cluster workstation. Start/stop mode can be more efficient in this case.

READING AND WRITING TO HALF-INCH TAPE

The tape server can read from or write to half-inch tape at any valid tape position. The server can overwrite a previous file. However, all data on the tape is lost beyond the position that writing begins. Single records on half-inch tape cannot be updated.

You are not required to use a special EOT sequence when performing I/O to a multivolume half-inch tape file. The half-inch tape server does not use an internal buffer to implement streaming.

When the server reaches the EOT, a status code is returned to the program. Your program can then prompt the user to insert a new tape.

OPERATIONS

The tape management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

QUARTER-INCH AND HALF-INCH TAPE

The following operations are used for programming to QIC tape and to half-inch tape.

OpenTape — Gives a user exclusive access to the tape drive and positions the tape.

CloseTape — Removes a user's exclusive access to a tape drive, thereby making the drive available to other users. For half-inch tape, CloseTape rewinds the tape. The tape is not rewound for QIC tape.

ReadTapeRecords — Reads n fixed-length records from the tape into a user buffer.

WriteTapeRecords — Writes n fixed-length records.

TapeStatus — Allows users to determine the status of the tape drive.

TapeOperation — Allows the user to issue such non-data transfer commands as **Rewind Tape**, **Erase**, and **Skip Records** to a tape drive.

QUARTER-INCH TAPE

The following operations are used for reading and writing a multivolume QIC tape file.

QICSync	Functions as part of the dialogue between the client and the QIC server in handling EOT.
ReadQICHeader	Reads the header information on a newly inserted tape. ReadQICHeader bypasses the QIC tape server's internal buffer, which may contain data from a partially read record.
WriteQICHeader	Writes the header information on a newly inserted tape. This operation bypasses the QIC server's internal buffer, which may contain data to be written to the new tape.

19 GENERIC PRINT ACCESS METHOD

The Generic Print Access Method (GPAM) is a library of object module procedures, which send text formatting commands to an output device. GPAM is a high-level, device-independent I/O programmer interface. (See Chapter 6, "Input/Output.")

You would typically use GPAM if you wish to add a variety of formatting characteristics to text you output to a printing device.

GPAM's formatting commands communicate with the output device through the Sequential Access Method (SAM). (For details on GPAM, see the Generic Print System Programmer's Guide.)

20 STRUCTURED FILE ACCESS METHODS

The file management system described in Chapter 11 provides access to disk file data as randomly addressable, 512 byte sectors. Up to 127 sectors can be read or written in a single request. Data is transferred directly between disk and the buffer specified in the read/write request (that is, it is not buffered by the file system). Asynchronous operation (concurrent I/O and computation on behalf of the same process) is a standard feature of the file management system.

Several structured file access methods (described in detail in the following three chapters in this manual) augment the capabilities of the file management system. The file access methods are object module procedures that can be linked to application programs as required. (See the Linker/Librarian Manual.) These object module procedures provide buffering and use the asynchronous I/O capabilities of the file management system to automatically overlap I/O and computation.

In contrast to the file management system, which organizes disk file data as unstructured 512 byte sectors, the structured file access methods organize disk file data as one of the following:

- a sequence of variable-length records
- a sequence of fixed-length records

Files are organized as a contiguous sequence of records. They are both blocked (as many records as possible are stored in each physical sector) and spanned (logical records are permitted to cross physical sector boundaries).

Generally, a file is created and accessed by

- the Indexed Sequential Access Method (ISAM) or the Direct Access Method (DAM), if the file is a sequence of fixed-length records
- the Record Sequential Access Method (RSAM), if the file is a sequence of variable-length records

Note that SAM described in Chapter 7 is an unstructured file access method. SAM is used to create and to subsequently access a file consisting of an unstructured sequence of bytes called a byte stream. (See Chapter 7, "Sequential Access Method," for details.)

STRUCTURED FILE ACCESS METHOD CHARACTERISTICS

The structured file access methods and their general characteristics are the following.

Indexed Sequential Access Method (ISAM) provides random and sequential, nonoverlapped I/O. Nonoverlapped means that a call to an ISAM operation does not return to the application program until an associated I/O is complete.

ISAM is a multikey, multiuser access method. Each ISAM data set is composed of one type of data record of a fixed format. Therefore, all data records in a given ISAM data set have the same size.

The size of the data records, the number of keys, the type of each key, and the method of ordering keys are specified when an ISAM data set is created.

An ISAM data set consists of two files: an index file and a data store file.

ISAM consists of object module procedures in the library, ISAM.lib. ISAM is a separately purchasable software product. (See the ISAM Manual for details.)

Record Sequential Access Method (RSAM) provides sequential, overlapped I/O. Overlapped means that although the application program makes a call to an RSAM operation and that operation returns, I/O can continue concurrently (overlapped) with the computations of the application program.

An RSAM file is accessed as a sequence of fixed- or variable-length records. Files can be opened for read, write (which replaces any prior file content), and append. In addition to pure sequential access, there are operations for scanning forward to the next well-formed record following detection of a malformed record.

RSAM consists of object module procedures in the standard operating system library, CTOS.lib.

Direct Access Method (DAM) provides random, non-overlapped I/O.

A DAM file is accessed as a sequence of numbered, fixed-length records. Random access is by record number; the implementation is such that reading or writing records with sequential record numbers provides good sequential performance. Files can be opened for read or modify (permitting selective modification for prior file content).

DAM consists of object module procedures in the standard operating system library, CTOS.lib.

HYBRID ACCESS PATTERNS

In the following chapters, the terms ISAM data store file, RSAM file, and DAM file are used to denote the primary means by which the file is accessed.

This usage, while convenient, is oversimplified: any file created with ISAM, RSAM, or DAM can be physically viewed as unstructured and accessed using SAM. Similarly, any file of records created with DAM or ISAM can be physically accessed using RSAM (that is, treating fixed-length records as a special case of variable-length records). Finally, an ISAM data store file contains fixed-length records and therefore can be accessed using DAM.

Although all these hybrid access patterns are possible, they are not all advisable. For example, reading a DAM file with SAM fetches control bytes along with the DAM record bytes; interpreting these requires special knowledge. Also, the file header for ISAM data store files, RSAM files, and DAM files contains a byte used to identify the file type. Accessing the file with a different access method can alter this byte. For example, if an ISAM data store file is accessed with DAM, it is marked as a DAM file and cannot be accessed by ISAM operations unless an ISAM Reorganize is done. (See the ISAM Manual for details.)

An ISAM data store file has an associated index file that must be updated in a complex way when the data store file is modified. If the data store file is modified using ISAM, this is done automatically. If the data store file is updated otherwise, the integrity of the ISAM data set can easily be destroyed. (See the ISAM Manual for details.)

The hybrid access patterns listed below are both useful and safe:

- Use of RSAM or DAM to read an ISAM-created file as though it were an unkeyed DAM file, that is, with the records accessed according to their physical ordering.

- Use of RSAM to read, write, or append to a DAM-created file. (However, if, following a write or append to such a file, there are records of different lengths, the file is subsequently accessible only with RSAM, not with DAM.)

- Use of DAM to read or modify an RSAM-created file in which all records have the same length.

MODIFYING AND READING DATA FILES

The **Maintain File** command can modify and/or read RSAM and DAM data files. **Maintain File** can do all of the following:

- verify the file structure

- remove malformed records

- remove deleted records

- optionally write a log of the verification of the file structure to a video display

Maintain File is described in the Executive Manual.

Maintain File also is used with the ISAM **Reorganize** command. (See the ISAM Manual for details.)

ISAM data store files, RSAM files, and DAM files are standard access method files. As such, they contain standard record headers, record trailers, and file headers.

A physical record consists of the record header, the record data, and the record trailer stored in contiguous bytes.

A standard file header is located at the beginning of the first sector at the start of the file. The header consists of information common to all standard access methods followed by information unique to the particular access method. The first physical record is located at the beginning of the second file sector.

The structure of a standard file header, a standard record header, and a standard record trailer are given in the CTOS/VM Reference Manual, Chapter 4, "System Structures." (See Tables 4-22, 4-23, and 4-24, respectively.)

OPERATIONS

The file access methods provide the operation listed below. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description.)

GetStamFileHeader
: Copies the file header of an RSAM, DAM, or ISAM file into the specified area.

21 INDEXED SEQUENTIAL ACCESS METHOD

The Indexed Sequential Access Method (ISAM) provides efficient, yet flexible, random access to fixed-length records identified by multiple keys stored in disk files.

Each ISAM data set holds one type of data record. The size of the data records, the number of keys, and the type of each key are specified when an ISAM data set is created.

ISAM is described more fully in the ISAM Manual.

22 RECORD SEQUENTIAL ACCESS METHOD

The Record Sequential Access Method (RSAM) provides efficient sequential access to fixed- and variable-length records. Records are read and written using sequential, overlapped I/O. Records are both blocked (as many records as possible are stored in each physical sector) and spanned (logical records are permitted to cross physical sector boundaries). There is also an operation to scan forward to the next well-formed record following detection of a malformed record. Files can be opened for read, write (which replaces any prior file content), and append.

RSAM can be called directly from any Convergent programming language. RSAM consists of object module procedures contained in the standard operating system library, CTOS.lib.

RSAM FILES AND RECORDS

The RSAM provides efficient sequential access to fixed- and variable-length records in a file. An RSAM file is a sequence of these records.

A record can be as large as 65,527 bytes or as small as 1 byte. To provide efficient disk storage, records are blocked and spanned.

If a sector cannot be read or a record is malformed, the remainder of the file can be read after the ScanToGoodRsRecord operation is used to locate the next well-formed record.

WORKING AREA

RSAM uses a work area supplied by the application program. A Record Sequential Work Area (RSWA) is a 150 byte memory work area for the exclusive use of the RSAM procedures. Multiple RSAM files can be open simultaneously using separate RSWAs.

BUFFER

RSAM also uses a word-aligned buffer supplied by the application program. The buffer must be at least two sectors (1K byte) long. The buffer size is not constrained by the longest record to be read or written, but, in such cases, performance can be improved by using large buffers.

RSAM uses overlapped output. Therefore, data written to an RSAM file can be retained in the buffer and not actually written to the file until some time after the WriteRsRecord operation returns. The CheckpointRsFile operation flushes the buffers of an RSAM file, ensuring that all data was written to disk.

OPERATIONS

The RSAM operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

BASIC

OpenRsFile	Opens or creates an RSAM file.
ReadRsRecord	Reads the next record from an RSAM file.
WriteRsRecord	Writes a record to an RSAM file.
CloseRsFile	Closes an RSAM file (including conclusion of all I/O operations).

ADVANCED

SetRsLfa	Sets the logical file address at which the next I/O operation will occur.
GetRsLfa	Returns the logical file address at which the next I/O operation will occur.
ScanToGoodRsRecord	Scans forward to the next well-formed record in an RSAM file.
CheckpointRsFile	Checkpoints the open output RSAM file.
ReleaseRsFile	Releases all resources associated with an open RSAM file (for example, open files and exchanges).

23 DIRECT ACCESS METHOD

The Direct Access Method (DAM) provides efficient random access to fixed-length records. A record is referred to in DAM by the record number within a file.

DAM can be accessed in COBOL through COBOL Relative I/O. DAM can also be called directly from any of the Convergent programming languages. DAM consists of object module procedures in the standard operating system library, CTOS.lib.

In reading, writing, or deleting, DAM does simple address calculations based on the record size and number to find the required sectors of the DAM file. DAM keeps a cache of recently referenced sectors that are obtained without reference to the disk. Sectors not in the cache are accessed with a single disk access.

DAM FILES, RECORDS, AND RECORD FRAGMENTS

DAM provides efficient random access to records identified by the record number within a file. The record number of the first record in a DAM file is 1.

A DAM file is a sequence of fixed-length records. The length of a record is specific to each DAM file and is specified when the file is first created.

A record can be as large as 63,992 bytes or as small as 0 bytes. To provide efficient disk storage use, records are both blocked (as many records as possible are stored in each physical sector) and spanned (logical records are permitted to cross physical sector boundaries). A record that is blocked and spanned contains the standard 8 bytes of header and trailer in addition to the stored data of the record itself.

A record fragment is a contiguous area of memory within a record. A record fragment is specified using an offset from the beginning of the record and a byte count. The record fragment must be contained within the record.

Record fragments are read from and written to open DAM files using the operations ReadDaFragment and WriteDaFragment, respectively.

WORKING AREA

DAM uses a work area supplied by the application system. A Direct Access Work Area (DAWA) is a 64 byte memory work area for the exclusive use of the DAM procedures. Any number of DAM files can be open simultaneously using separate DAWAs.

BUFFER

DAM also, uses a word-aligned buffer supplied by the application program. The buffer size is specified by the program. The size is subject only to the constraint that it be a multiple of 512, and that it be greater than or equal to the record size plus 519.

This constraint can be relaxed in two cases:

- If 512 is a multiple of the record size plus 8, the minimal size is simply 512.
- If the record size plus 8 is a multiple of 512, the minimal size is the record size plus 8.

BUFFER SIZE AND SEQUENTIAL ACCESS

DAM reads from and writes to the buffer by using a single request to the file management system. This typically requires only a single disk access. Whenever the disk is read, the entire buffer is filled.

If the buffer size is chosen to be larger than the record size (by at least a factor of 2), the buffer acts as a look-ahead cache. If sequentially numbered records are requested, DAM typically finds them in the buffer and does not access the disk. In this way, if the application program makes a suitable buffer size choice, DAM can provide efficient record sequential access.

BUFFER MANAGEMENT MODES: WRITE-THROUGH AND WRITE-BEHIND

DAM provides two modes of buffer management: write-through and write-behind. The mode is initially set to write-through when a DAM file is opened. The mode can be changed using the SetBufferMode operation.

In the write-through mode, DAM immediately writes the changed sectors of the buffer to disk whenever a record is written or deleted. DAM guarantees that the file content on disk is accurate at the completion of a modify operation.

In the write-behind mode, DAM writes changed sectors of the buffer to disk only when new sectors are brought into the buffer, the DAM file is closed, or the mode is changed to write-through. Write-behind mode provides better performance when DAM is used to modify records in sequential order.

OPERATIONS

The DAM operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

BASIC

OpenDaFile	Opens or creates a DAM file.
ReadDaRecord	Reads a record from a DAM file.
WriteDaRecord	Writes a record to a DAM file.
DeleteDaRecord	Deletes a record from a DAM file.
CloseDaFile	Closes a DAM file.

ADVANCED

QueryDaRecordStatus	Copies to the specified area the status of a record in an open DAM file.
QueryDaLastRecord	Copies to the specified area the number of the last record in an open DAM file.
TruncateDaFile	Truncates an open DAM file (that is, it removes all records beyond a specified point).
ReadDaFragment	Reads a record fragment from an open DAM file.

WriteDaFragment
: Writes a record fragment to an open DAM file.

SetDaBufferMode
: Sets the buffer management mode to write-through or write-behind.

24 MEMORY MANAGEMENT

Memory management supports the dynamic allocation and deallocation of memory areas in an application partition for a program's code and data storage.

TYPES OF MEMORY

The two types of memory allocation available to a program are long-lived and short-lived. Within each application partition, long-lived memory expands upward from low memory locations, while short-lived memory expands downward from high memory locations.

The operating system allocates short-lived memory for the program's code and static data when it loads the program. No explicit use of memory management operations by the programmer is necessary to do this.

You can obtain additional long-lived and short-lived memory for your program by making requests of the operating system.

When program execution is terminated, the short-lived memory of its partition is automatically deallocated.

Long-lived memory is deallocated only at the explicit request of each application program. Therefore, long-lived memory is useful for passing information from an application program to a succeeding program in the same partition. Long-lived memory is deallocated, however, by a program that calls the Chain operation and is replaced by the Executive.

ADDRESSING MEMORY

In real mode, you are limited to a 1 megabyte physical address space. This means you can reference each of 1,048,576 bytes by a unique physical address.

The physical address (PA) is the actual location in memory.

In protected mode, the physical address space extends beyond the first megabyte. The amount of physical memory you can address is determined by your system's processor and its hardware limitations. A 80286 processor, for example, is capable of providing a 16 megabyte physical address space. The actual address space, however, is determined by the hardware. (For details on the address space, also see Chapter 3, "Using CTOS/VM Operations.")

SEGMENTS

A segment is a contiguous area of fewer than 64K bytes within the physical address space. The operating system uses segmented addressing. This means every address is relative to a segment.

A paragraph is 16 bytes of memory. In real mode, segments are aligned on paragraph boundaries in physical memory.

It is conventional to address a byte within a segment by using a logical memory address. A logical memory address consists of the following:

- a 16 bit segment address (SA)
- a relative address (RA) (called an offset)

In real mode, the SA is the actual segment base address. The segment base address is the first byte of the segment in physical memory.

In protected mode, the SA is a selector (SN). The SN is the index of a segment descriptor entry in either a Local Descriptor Table (LDT) or a Global Descriptor Table (GDT). (For details on protected mode structures, see Chapter 3, "Using CTOS/VM Operations.")

The segment descriptor contains a segment base address, which may be located anywhere in physical memory. For this reason, if you are writing a program you want to execute in protected mode, your program should not depend upon the value of the SA.

The RA (or offset) is the low-order 16 bits of a logical address. It is the distance, in bytes, of the target location from the beginning of the segment.

A byte of memory does not have a unique logical memory address. The same byte of memory can be referred to by many different combinations of SAs and RAs.

In this manual, the term memory address means logical memory address. (Chapter 30, "Inter-CPU Communication," describes a linear address used for routing requests among processor boards in SRPs. This is the only case in which memory address has another meaning.)

CODE, STATIC DATA, AND DYNAMIC DATA SEGMENTS

The three types of segments are code, static data, and dynamic data. Each segment type can be either shared or exclusive.

- A code segment contains only processor instructions (code) and is never modified once it is loaded into memory. This characteristic permits several processes to execute instructions from the same code segment. It also allows the Virtual Code management facility to reload code segments from the run file as needed without previously saving a copy of the segment in memory. (For details, see Chapter 34, "Virtual Code Management.")

- A data segment contains writable data. There are no restrictions on modifying a data segment's content. If a data segment is shared among processes, concurrency control is the responsibility of those processes.

  A static data segment is automatically loaded into memory when the run file that contains it is loaded. A dynamic data segment is allocated by a program in memory by means of run-time calls to the operating system.

A program on disk is stored in a run file that contains code and/or static data segments. When requested, the operating system loads the program into a memory partition and adjusts any logical memory addresses that exist in either code or data segments to reflect the memory address at which the program is loaded. (See Figure 24-1.)

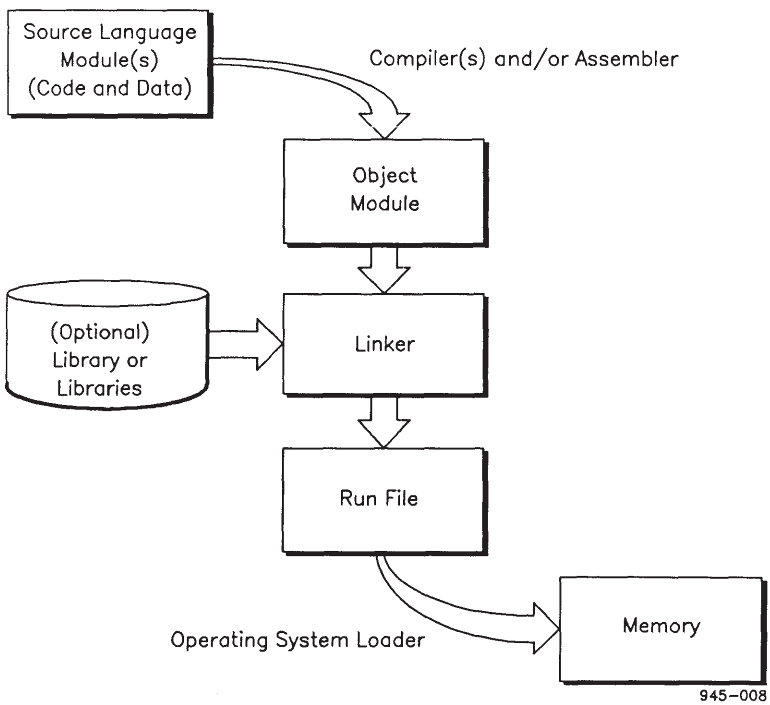


Figure 24-1. From Source Language Modules to Program in Memory

Code and static data segments are created by compiling and/or assembling source language modules into object modules and linking the object modules together into code and data segments.

The Linker reads the object module(s) and combines them according to their segment names, class names, and directives from the user.

(For details, see the Linker/Librarian Manual and the Assembly Language Manual.)

Segments can be combined based on a series of different models of computation (use of segment registers). Most programming languages use the medium model, although the operating system also supports the small and large model. (For details, see the CTOS Programmer's Guide.)

A run file created by linking object modules produced by the Pascal compiler, for example, consists of one code segment for each object module included in the link and a single static data segment. The single static data segment, or DGroup, combines the static data and stack requirements of all the object modules.

A run file of this form is considered standard; assembly language programmers are urged to adopt this standard unless other considerations are overriding. The COBOL compiler and BASIC interpreter do not produce object modules. (For details, see the Linker/Librarian Manual.)

A program can allocate a dynamic data segment of memory by means of run-time calls to the operating system.

The Virtual Code Management facility allows you to run a program that is larger than the available memory in an application partition. If the Virtual Code management facility is in use, all the static data segments and the resident code segment are loaded into memory. The nonresident code segments are loaded into memory only as needed. (For details, see Chapter 34, "Virtual Code Management.")

NOTE: This manual generally describes a logical model of the operating system rather than a particular implementation (such as real mode or protected mode). (For implementation details, see the Release Notice for your version of the operating system.)

Figure 24-2 shows the memory organization of an application partition. Because system services do not allocate or deallocate memory, the memory in a system partition consists only of enough short-lived memory for the system service itself. (For details on system partitions, see Chapter 31, "System Services Management.")

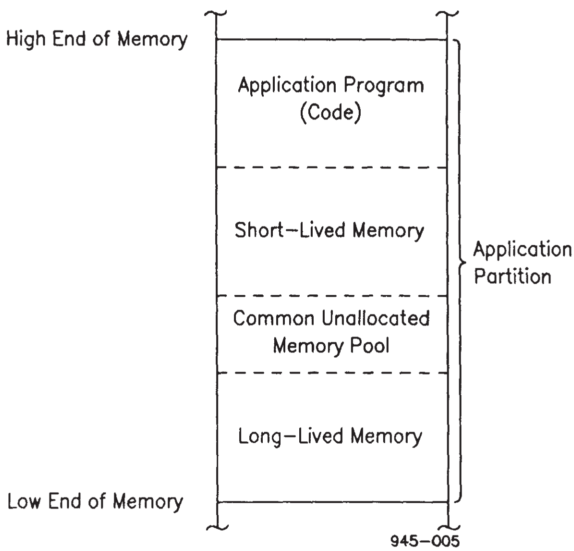


Figure 24-2. Memory Organization of an Application Partition

LONG-LIVED AND SHORT-LIVED MEMORY

All currently unallocated long-lived and short-lived memory in an application partition is in a contiguous area called the <u>common</u> <u>unallocated</u> <u>memory</u> <u>pool</u>. Memory can be allocated from both ends of the pool. There is no restriction on how much can be allocated from either end, other than that the sum of the allocations cannot exceed the amount of memory available in an application partition. The QueryMemAvail or QueryBigMemAvail operation returns the size of all available memory in an application partition.

In real mode, memory is allocated and deallocated only on paragraph boundaries. That is, the physical address of the area is a multiple of 16. Because of this, the areas of memory the operating system allocates can be conveniently referenced by using the segment addressing convention discussed in "Segments," earlier in this chapter.

The AllocMemoryLL, AllocAreaSL, and AllocMemorySL operations allocate long-lived (LL) and short-lived (SL) memory segments in an application partition. The AllocAllMemorySL operation can allocate more than 65,536 bytes, and thus the entire area allocated by this operation is not necessarily addressable as a single segment.

The DeallocMemoryLL and DeallocMemorySL operations deallocate long-lived and short-lived memory segments, respectively, in an application partition. The ResetMemoryLL operation deallocates all long-lived memory in an application partition.

The ExpandAreaLL and ExpandAreaSL operations increase the size of a segment previously allocated using the AllocMemoryLL or AllocAreaSL operations, respectively. (Segments allocated with AllocMemorySL should not be expanded.) If the Linker's DS allocation option is specified, ExpandAreaSL also may be used to increase the size of the static data segment, DGroup. (See the Linker/Librarian Manual for details.)

The ShrinkAreaLL and ShrinkAreaSL operations decrease the size of a segment previously allocated using the AllocMemoryLL or AllocAreaSL operations, respectively. (Segments allocated using AllocMemorySL cannot be decreased in size.)

DEALLOCATIONS

Relative to allocations from one end of the memory of an application partition, deallocations must occur in exactly the opposite sequence. That is, the user must follow a last-allocated, first-deallocated discipline when deallocating either long-lived or short-lived memory. For example, if a program allocates short-lived memory segments A, B, and C, it must deallocate them in the order C, B, A.

Thus the motion of the borders (the dashed lines in Figure 24-2) of the common memory pool in an application partition resembles the playing of an accordion: the borders converge when memory is allocated and diverge when memory is deallocated. This scheme is efficient because all unallocated memory is in a common pool and because the operating system has to remember only the addresses of the next (long-lived and short-lived) segments to allocate, not the addresses of all allocated segments.

LONG-LIVED MEMORY USES

The long-lived memory in an application partition is used for VLPB parameters passed from one program to a succeeding program in the same partition. A program cannot place 32 bit logical memory addresses in long-lived memory. This is because the long-lived memory of a variable partition can be relocated when a program terminates and is replaced by a succeeding program with different memory requirements.

Long-lived memory allocations are returned to the common pool of unallocated memory in an application partition upon explicit request of the program or if the program calls the Chain operation and is replaced by the Executive.

SHORT-LIVED MEMORY USES

The short-lived memory in an application partition is used by the operating system to contain the code and static data segments of each application program. If code is shared, however, code can be located anywhere in memory. (For details, see Chapter 32, "Program and Partition Management.") Short-lived memory also is allocated by application programs for use as dynamic data segments for data that is to be processed only by the current program. Other common uses of short-lived memory are I/O buffers and the Pascal heap.

Short-lived memory allocations are returned to the common memory pool whenever the program is replaced (in any application partition by the Chain, ErrorExit, or Exit operations, or if a single application partition is in memory, by the key combination **Action-Finish**). (See Chapter 4, "Program Management.")

OPERATIONS

The memory management operations are described below. Operations in the first two categories are arranged in a most to least frequent use order. Operations in the remaining categories are alphabetized. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

SHORT-LIVED MEMORY

AllocAreaSL — Creates a short-lived segment and allocates memory for it of the specified size. AllocAreaSL returns a 32 bit logical address of the base of the segment.

AllocMemorySL — Similar to AllocAreaSL, except that the segment may not subsequently be increased or decreased in size. However, the offset portion of the 32 bit address returned is guaranteed to be 0, enabling the segment to be addressed using only the 16 bit segment base address portion.

AllocAllMemorySL — Allocates the largest possible short-lived memory segment in an application partition.

DeallocMemorySL — Deallocates a short-lived memory segment in an application partition.

ExpandAreaSL Allocates additional memory of the specified size within the specified short-lived segment. The specified segment must have been created by a prior call to AllocAreaSL (except when specifying the Linker's DS allocation option and ExpandAreaSL to expand the static data segment, DGroup).

ShrinkAreaSL Deallocates memory of the specified size within the specified short-lived segment. The segment must have been created by a prior call to AllocAreaSL (except when specifying the Linker's DS allocation option and ShrinkAreaSL to decrease the size of the static data segment, DGroup).

AllocMemoryFramesSL
Creates a short-lived segment and allocates cFrames*4096 bytes of short-lived memory at the beginning of the segment. The beginning of the segment is aligned on a 4K byte boundary in physical memory.

LONG-LIVED MEMORY

AllocMemoryLL Allocates a long-lived memory segment in an application partition.

DeallocMemoryLL
Deallocates a long-lived memory segment in an application partition.

ExpandAreaLL Allocates additional memory of the specified size within the specified long-lived segment.

ShrinkAreaLL — Deallocates memory of the specified size within the specified long-lived segment.

ResetMemoryLL — Deallocates all long-lived memory in an application partition.

SHORT-LIVED AND LONG-LIVED MEMORY

CreateAlias — Returns an alias selector for the specified source memory address. The alias selector combined with a 0 offset references the same linear address as the specified source memory address.

DefineInterlevelStack — Initializes the stack segment (SS) and the stack pointer (SP) fields of the caller's task state segment for the specified protection level. DefineInterlevelStack is supported in protected mode only.

DefineLocalPageMap — Defines an address mapping between the linear address specified by saLocal and the physical address referenced by pFrames. This operation is supported only by 80386 microprocessor-based operating systems executing in protected mode.

InitLocalPageMap
Initializes the segment addressed by pLocalPageMap for use as a local page map. The segment must be at least 8K bytes in size and must be aligned on a 4K byte boundary in physical memory. This operation is supported by 80386 microprocessor-based operating systems executing in protected mode only.

QueryBigMemAvail
Returns the size in bytes of all available free memory in an application partition.

QueryMemAvail Returns the size in paragraphs of all available free memory in an application partition.

ADDRESS TRANSLATION

PaFromP Returns the 32 bit PA referenced by the logical memory address. PaFromP supports software that interfaces with hardware using physical addresses.

PaFromSn Returns the PA referenced by the protected mode selector (SN). PaFromSn is supported by operating systems executing in protected mode only and is invoked by PaFromP.

SgFromSa Returns an alias GDT selector (SG) that references the same memory location as the specified SA.

SnFromSr — Returns the protected mode SN that references the same memory location as the specified real mode segment address (SR). SnFromSr is supported only by operating systems executing in protected mode.

SrFromSn — Returns the real mode SR that references the same memory location as the specified protected mode selector (SN). SrFromSn is supported only by operating systems executing in protected mode.

ALIAS MANAGEMENT

CreateAlias — Returns an alias selector for the specified source memory address. The alias selector combined with a 0 offset references the same linear address as the specified source memory address.

ReuseAlias — Rewrites the base address and limit fields of an alias selector initially allocated by CreateAlias. ReuseAlias is supported in protected mode only.

ReuseAliasLarge — Is the same as ReuseAlias except that it provides a more general interface. ReuseAliasLarge is supported in protected mode only.

OTHER

AllocMoverSegment
: Allocates a variable-length segment of memory with an address greater than 1 megabyte.

DeallocMoverSegment
: Frees a variable-length segment previously allocated with the AllocMoverSegment operation.

SetSegmentAccess
: Sets the access mode of a code or a data segment in protected mode. SetSegmentAccess performs no function in real mode.

25 UTILITY OPERATIONS

The standard operating system library, CTOS.lib, provides a number of utility operations that can be used to maximize the efficiency of writing programs.

DATE/TIME MANAGEMENT

SYSTEM DATE/TIME STRUCTURE

If a program executing on a master or standalone workstation needs to know the time to greater precision than 1 second, it can access the system date/time structure by calling the GetpStructure operation with a structCode parameter of 240. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a description of GetpStructure.)

The system date/time structure is shown in Table 4-26 in the CTOS/VM Reference Manual)

SYSTEM DATE/TIME FORMAT

The system date/time format provides a compact representation of the date and the time of day that precludes invalid dates and allows simple subtraction to compute the interval between two dates. The system date/time format consists of the seconds and the dayTimes2 fields of the system date/time structure.

The system date/time format is represented in 32 bits to an accuracy of 1 second. The high-order 15 bits of the high-order word contain the count of days since March 1, 1952. The use of a 15 bit field allows dates up to the year 2042 to be represented. The low-order bit of the high-order word is 0 to represent AM and 1 to represent PM. The low-order word contains the count of seconds since midnight/noon. Valid values are 0 to 43199.

The current system date/time is maintained in the master (for all the workstations of a cluster configuration) or in the standalone workstation.

You can access and modify the current system date/time by calling the GetDateTime and SetDateTime operations.

EXPANDED DATE/TIME FORMAT

The ExpandDateTime and CompactDateTime operations convert between the system date/time format and an expanded date/time format in which year, month, day of month, and so forth, are represented as discrete fields.

(See Table 4-9 in the CTOS/VM Reference Manual for the expanded date/time format.)

STRING COMPARING

String comparing operations inform you of string equalities.

StringsEqual states whether two strings contain exactly the same data. StringsEqual does no translation.

ULCMPB compares two strings, using uppercase and lowercase translations. Unlike StringsEqual, ULCMPB can take nationalized character sets into account. (For details on nationalization, see Chapter 40, "Native Language Support.") The Executive program uses this operation for interpreting the field entries in a command form or for file matching.

NlsYesOrNo and NlsYesNoOrBlank are two other string comparing operations that handle nationalized characters. (For details, see Chapter 40, "Native Language Support.")

NlsYesOrNo uses uppercase and lowercase translations to compare a string against nationalized words meaning **yes** and **no**. The string passed can match any portion of a **yes** or **no** word. For example, **y**, **ye**, and **yes** match **yes**.

NlsYesNoOrBlank performs the same function as NlsYesOrNo, except that NlsYesNoOrBlank, in addition, checks for a null string.

It is recommended that the operations NlsYesOrNo or NlsYesNoOrBlank be used in conjunction with the RgParam operation for parsing answers to yes/no options of Executive parameters. (For details on RgParam, see Chapter 5, "Parameter Management.")

FComparePointer compares two logical addresses for equality. FComparePointer typically is used to compare the binary values of the logical addresses. However, it also can be used to compare the byte locations of the addresses in the linear memory address space. (For details on memory addresses, see Chapter 3, "Using CTOS/VM Operations," and Chapter 24, "Memory Management.")

OUTPUT ROUTINES

Output routines allow you to direct information to any byte stream (including the video device) in a way that is compatible with the operating system. The default output device is [VID]0 (video frame 0).

These operations are replacements for language run time operations. They provide a convenient and efficient way of coding strings in a language such as PL/M, which has no run-time support for displaying strings.

NPrint
OutputBytesWithWrap
OutputQuad
OutputWord
PutByte
PutChar
PutPointer
PutQuad
PutWord
SbPrint
ZPrint

All of the output operations use NPrint and PutChar for output, allowing you to provide your own versions of NPrint and PutChar.

CONFIGURATION FILE PARSING

The configuration file parsing operations are used for parsing standard configuration files, which contain human readable entries of the form

:fieldname:value

Examples of these files are .user files, Context Manager configuration files, and Document Designer configuration files. (The CTOS System Administrator's Guide illustrates a user configuration file, which is a typical example.)

The file parsing operations are

LookUpField
LookUpNumber
LookUpReset
LookUpString
ReadToNextField

TEXT EDITING

The TextEdit operation is a useful operation for building a single-line text editor. You can call the TextEdit operation if you want to write a program that allows the user to do either of the following:

- enter data into a field
- edit the data entered by means of the normal keystrokes of **Backspace**, **Left Arrow**, **Right Arrow**, and **Code-Left Arrow**, such as those used by the Executive

INFORMING USER OF WAITING MAIL

The QueryMail operation can be used by any program to display that new mail is waiting for the user. The Executive, for example, uses this operation to display the mail message on the video status line.

OPERATIONS

The utility operations are described below. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

DATE/TIME MANAGEMENT

CompactDateTime	Converts from an expanded date/time format to system date/time format.
ExpandDateTime	Converts from the system date/time format to an expanded date/time format in which year, month, day of month, and so on, are represented as discrete fields.
FormatDateTime	Is the same as NlsFormatDateTime. FormatDateTime is documented for historic reasons only.
FormatTime	Converts an expanded date/time structure into an ASCII string containing the day, date, and time.
FormatTimeDt	Converts an expanded date/time structure into an ASCII string containing the date.
FormatTimeTm	Converts an expanded date/time structure into an ASCII string containing the time.
GetDateTime	Returns the current date/time in the system date/time format.

NlsFormatDateTime
: Converts from date/time format to textual string format. This operation is used if you are creating your own NLS tables to be linked with your program. (For details on NLS, see Chapter 40, "Native Language Support.")

NlsParseTime
: Converts a string into an expanded date/time structure.

NlsStdFormatDateTime
: Obtains the memory address of the Date and Time Formats table. This operation is recommended over either NlsFormatDateTime or FormatDateTime for ease in nationalization. (For details, see Chapter 40, "Native Language Support.")

ParseTime
: Is the same as NlsParseTime. NlsParseTime should be used for ease in nationalization.

SetDateTime
: Sets the date/time of the operating system.

STRING COMPARING

FComparePointer
: Compares two logical addresses. FComparePointer returns TRUE if the addresses have the same binary value.

FsCanon
: Translates a byte string into the file system canonical form with respect to case.

NlsULCMPB — Is the same as ULCMPB. NlsULCMPB is documented for historic reasons.

NlsYesNoOrBlank — Is similar to NlsYesOrNo, except that, in addition, NlsYesNoOrBlank checks for a null string.

NlsYesOrNo — Performs a case-insensitive string comparison against nationalized words meaning yes and no.

StringsEqual — Compares two strings using a Boolean function that returns TRUE (0FFh) if the two strings are the same.

ULCMPB — Compares two strings, using the lowercase to uppercase conversion table, if present, to carry out the case-insensitive comparison. NlsULCMPB returns 0FFFFh if the two strings are equal/ otherwise, it returns a word containing the index of the first two characters in the strings that are different. ULCMPB should be used over NlsULCMPB for ease in nationalization.

WildCardMatch — Checks a string against a wild card specification.

OUTPUT ROUTINES

NPrint	Prints a string to the video or other device.
OutputBytesWithWrap	Outputs a string to the video byte stream. If the string does not fit on the current line, a Carriage Return and Tab are inserted to continue the string output on the next line.
OutputQuad	Prints a quad (32-bit unsigned integer) to the video or other device as specified by the NPrint and PutChar operations.
OutputWord	Prints a word (16-bit unsigned integer) to the video or other device as specified by the NPrint and PutChar operations.
PutByte	Prints a byte (8-bit unsigned integer) to the video or other device as specified by the NPrint and PutChar operations.
PutChar	Prints a character to the video or other device.
PutPointer	Prints a memory address to the video or other device as specified by the NPrint and PutChar operations.
PutQuad	Prints a quad to the video or other device as specified by the NPrint and PutChar operations.

PutWord	Prints a word to the video or other device as specified by the NPrint and PutChar operations.
SbPrint	Prints a string in which the first byte is the size of the string to the video or other device as specified by the NPrint and PutChar operations.
ZPrint	Prints a null-terminated string to the video or other device as specified by the NPrint and PutChar operations.

CONFIGURATION FILE PARSING

LookUpField	Reads from a file and searches for a :FieldName: string, beginning at the current location in the file. The operation returns the string :FieldName: and the count of bytes in the string.
LookUpNumber	Reads from a file and searches for a :FieldName: string, beginning at the current location in the file. The operation returns the string length to the caller.
LookUpReset	Resets the point from which a scan begins to the beginning of the current file.
LookUpString	Reads from a file and searches for a :FieldName: string, beginning at the current location in the file. The operation returns the count of bytes in the string.

ReadToNextField

Reads from a file and searches for a :FieldName: string. The operation stores the text strings between the current location and the :FieldName: string, setting the beginning of :FieldName: to be the current location.

TEXT EDITING

TextEdit — Edits a line of text. The operation takes a character and a text descriptor and returns the descriptor with appropriate changes.

OTHER

QueryMail — Displays to the video status line the fact that mail is waiting for the user. QueryMail can be called by any program.

WriteLog — Writes a variable-length record to the system log file.

26 SYSTEM DEFINITIONS

This chapter presents the system structures and other kinds of system-related information. This chapter also recommends methods you can use to obtain system information.

Table 26-1 presents the system structures and provides a brief description of each. (See the CTOS/VM Reference Manual, Chapter 4, "System Structures," for detailed descriptions of each of these structures.)

Table 26-1

SYSTEM STRUCTURES
(Page 1 of 4)

System Structure	Definition
Application System Control Block	Passes parameters and other information between programs within a partition.
Boot Block	Contains the information passed to the operating system by the bootstrap ROM.
Communications Status Buffer	Contains usage statistics for the master workstation and the workstations attached to it.
Device Control Block	Describes the type, characteristics, and status of a disk.
Expanded Date/Time Format	Contains discrete fields for the date/time, including the year, month, day of month, and so forth.
Extended Partition Descriptor*	Contains specifications for the current application program file and exit run file.

*This structure is for internal use only.

Table 26-1

SYSTEM STRUCTURES
(Page 2 of 4)

System Structure	Definition
File Header Block	Contains information about a file, its disk address, and its disk extents.
Frame Descriptor	Contains all information about a video frame.
Real Mode Interrupt Vectoring	Contains hardware and software interrupt and trap vectors.
Port Structure	Contains hardware port addresses of various devices whose memory addresses differ in various configurations.
Process Control Block*	Contains the combined hardware and software context of a process.
Queue Status Block	Contains a queue entry's server user number, priority, and the buffers in which the queue entry handles for the queue entry and the logically following queue entry are stored.

*This structure is for internal use only.

Table 26-1

SYSTEM STRUCTURES
(Page 3 of 4)

System Structure	Definition
Standard File Header Format	Contains file header information, such as the file signature, file type, and the minimum and maximum file record sizes.
Standard Record Header Format	Contains record information, such as the unique record identifier, the physical record size, and the record status.
Standard Record Trailer Format	Indicates whether the record is malformed.
System Configuration Block	Contains detailed information about the System Image.
System Date/Time Structure	Contains information about the system date/time to a greater precision than 1 second.

Table 26-1

SYSTEM STRUCTURES
(Page 4 of 4)

System Structure	Definition
Terminal Output Buffer	Used by the Shared Resource Processor (SRP) Terminal Management operations. (See Chapter 17, "SRP Terminal Management," for details on these operations.)
Timer Pseudointerrupt Block	Used by the SetTimerInt and ResetTimerInt operations. (See Chapter 33, "Timer Management," for details on these operations.)
Timer Request Block Format	Controls the Realtime Clock (RTC) services.
Variable Length Parameter Block	Communicates parameters when a program chains to another program.
Video Control Block	Contains all information known to the operating system about the video display.

METHODS OF OBTAINING SYSTEM INFORMATION

Certain operations provide access to particular system structures. These operations and the system structures you can access are as follows:

Operation	System Structure
GetpASCB	Application System Control Block
GetpStructure	Returns the memory address of various system structures. (For details, see the CTOS/VM Reference Manual, Chapter 3, "Operations.")
GetUCB	User Control Block
GetVHB	Volume Home Block
QueryDCB	Device Control Block

You should use the GetpStructure operation to access the system structures not included in the list above. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a description of GetpStructure.)

Programs that access a system structure directly are not compatible with operating systems executing in protected mode.

As an example, historically, the Video Control Block (VCB) could be accessed directly by its memory address (244h) in low memory. This required a segment address of 0. The resulting logical address thus was

0:244

In protected mode, this address implies a selector (SN) of 0. An SN with a value of 0, however, is invalid. (For guidelines on writing protected mode programs, see the Engineering Update for 2.0 CTOS/VM.)

GetpStructure provides you with a valid memory address compatible in real mode and in protected mode.

OPERATIONS

The system information operations are described below. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

CLUSTER MANAGEMENT

GetClusterStatus
: Returns usage statistics for each communications line and for the workstations attached to it.

DISK MANAGEMENT

QueryDCB
: Copies the Device Control Block (DCB) of the specified device to the specified memory area.

FILE MANAGEMENT

GetUCB
: Copies the User Control Block (UCB) for the current user number to the specified area.

GetVHB
: Copies the VHB of the specified device to the specified memory area.

OPERATING SYSTEM

CurrentOSVersion
: Determines the version of an operating system. CurrentOSVersion should be used instead of OSversion for programs that run on earlier versions of the operating system.

EnterBootrom
: Transfers control to the beginning of the boot ROM.

FilterDebugFInterrupts
: Directs the Debugger to pass through Debugger interrupts (single step, breakpoint) on a per process basis by sending messages to an exchange.

FProcessorSupportsProtectedMode
: Returns TRUE on an 80286 or subsequent microprocessor (a processor capable of protected mode execution). It returns FALSE on an 8086 or 80186 microprocessor.

FProtectedMode
: Returns TRUE if the calling program is executing in protected mode. It returns FALSE if the program is executing in real mode.

FRmos
: Returns TRUE if the calling program is executing in real mode and the operating system is executing in protected mode.

FRmosUser
: Is the same as FRmos, except FRmosUser allows the specification of a user number.

GetCoProcessorStatus
: Reports if either a math coprocessor or a software floating-point emulator, such as the Math Server, is present (to execute floating-point instructions).

GetFRmosUser
: Is used to determine a client's execution mode. GetFRmosUser sets the fRmos flag to TRUE if the specified user number is executing a real mode program. Otherwise, the flag is set to FALSE.

GetNodeName
: Obtains the node name of the local node where this request is issued.

GetPartitionStatus
: Returns status information about the specified application partition and the program currently executing in it.

GetpASCB
: Returns the address of the ASCB of the application partition in which the program is executing.

GetpStructure
: Returns the memory address of an operating system structure.

OsVersion
: Is the same as CurrentOSVersion. CurrentOSVersion, however, should be used for programs running on earlier versions of the operating system.

QueryCoprocessor
: Reports if a coprocessor, such as the Math Server, is present (to execute floating-point instructions).

QueryLdtr	Returns the GDT selector that identifies the specified user number's (LDT). If the user number does not have an LDT, QueryLdtr returns the null selector.
SetpStructure	Provides controlled write access to selected fields of certain system data structures that may legitimately be modified by user programs running in protected mode.

USER NAME MANAGEMENT (name entered at signon)

GetWsUserName	Returns the user name that is signed on at the specified cluster workstation.
GetUserStatus	Copies user status information to the specified memory area.
SetWsUserName	Stores the user SignOn name of the workstation.

VIDEO

QueryVideo	Performs the same function as QueryVidHdw, except QueryVideo fills in all fields in the specified memory area.
QueryVidHdw	Places information describing the level of video capability of a workstation in the specified memory area.

27 MULTIPROGRAMMING

This chapter serves as an introduction to information that will become more important to you as you gain familiarity with the more immediate and practical operating system concepts described in previous chapters. The chapters that follow describe the operating system's multiprogramming capabilities.

Multiprogramming allows several programs to be in memory at once. In addition to independent execution scheduling, these programs are provided the ability to communicate with each other.

For example, it is possible for a program to communicate with

- other run files within the same partition
- programs in other partitions
- programs at other workstations within a cluster
- other processors of the Shared Resource Processor (SRP)
- programs at remote nodes

The multiprogramming chapters describe those events, transparent to you, that allow multiprogramming to take place. As an example, your program can request to write to a file. By using the appropriate write operation, you can have your output written to the file you specify. A status code is returned to your program, indicating the success or failure of this operation.

In a multiprogramming environment, the following are just a few of the events that take place as a result of your program request:

- The request is communicated to the file system by means of interprocess communication (IPC) and Kernel primitives.

- The file system manages access to the specified file, performs the requested service (sends output to the file), and responds to your program by means of IPC and Kernel primitives.

- Underlying these events, process management is at work, scheduling the execution activities of your program, the file system, and all other system processes that are competing with each other for processing time.

The multiprogramming subjects and a reference to the chapter in which each is described are as follows:

- Processes: A process is an independent thread of execution along with the hardware context (that is, the processor registers) necessary to that thread. One or more processes are created each time a program is executed. Certain operations manipulate processes, allowing you to create, prioritize, and obtain information about them for future programming reference. (See Chapter 28, "Process Management.")

- IPC: IPC is the core to communication among processes. Chapter 29 describes the IPC concepts of messages and exchanges and the relationships of these concepts to processes. (See Chapter 29, "Interprocess Communication.")

- Inter-CPU Communication (ICC): ICC is the method by which messages are passed between processor boards on the SRP. (See Chapter 30, "Inter-CPU Communication.")

- System Services: System services act as managers of resources that can be accessed by application programs or other system services. Chapter 31 describes how system services work and includes guidelines for writing system services. (See Chapter 31, "System Services Management.")

- Program and Partition Management: Chapter 32 describes how the operating system manages its memory resource. It describes how programs are loaded into memory and how they exit. It further describes the operations that can be used by a partition managing program to create and remove partitions under its management. (See Chapter 32, "Program and Partition Management.")

- Timer Management: Timer management describes the Realtime Clock (RTC) and the Programmable Interval Timer (PIT). (See Chapter 33, "Timer Management.")

28 PROCESS MANAGEMENT

PROCESS

A process is a single thread of program execution. It can be perceived from three points of view:

- The end user sees two processes on the Executive screen.
- The programmer creates additional processes to perform separate functions within a single program by making the appropriate operating system calls.
- The operating system schedules processes to use the processor.

END USER

As an end user, you can actually see two processes at work in the Executive. When you type into an Executive form, the main program process accepts your keystrokes. At the same time, a second process updates the clock. Whether or not you type, the clock continues to be updated.

PROGRAMMER

As a programmer, you perceive a process in terms of how you create an additional process in a multiprocess program like the Executive.

When the Executive run file is loaded into memory, the main program starts executing. At some point, it calls CreateProcess, which starts up the clock process. Each process executes as a separate thread. Global variables allow the main program and the clock to share the same data.

Typically, processes share the same code but have separate stacks. The degree and method of data sharing are program-specific.

OPERATING SYSTEM

The operating system Kernel views the two Executive processes as units competing for processor time. It is the operating system's responsibility to manage use of the processor among all existing processes.

PROCESS MANAGEMENT

The operating system's process management facility always allocates the processor to the highest priority process currently requesting it. In the Executive for example, the clock process runs at a higher priority than the process accepting user keystrokes to ensure that the clock gets updated.

Scheduling is driven by system events. Whenever an event, such as the completion of an I/O operation, makes a higher priority process eligible for execution, rescheduling occurs immediately.

This scheduling technique is called event-driven priority scheduling. It reduces overhead and provides for a more responsive system than techniques that are entirely time-based.

To give multiple programs with the same priority a fair share of system resources, processes with priorities in a predefined range are optionally subject to time slicing.

CONTEXT OF A PROCESS

The context of a process is the collection of all information about a process. The context has both hardware and software components.

The hardware context of a process consists of values to be loaded into processor registers when the process is scheduled for execution. This includes the registers that control the location of the process's stack.

The software context of a process consists of its default response exchange, the priority at which it is scheduled for execution, and the interrupt vectors pointing to software interrupt handlers that the program uses.

The root structure containing the combined hardware and software context of a process is a system data structure called the Process Control Block (PCB).

When a process preempts another process of lower priority, the operating system performs a context switch by saving the hardware context of the preempted process in that process's PCB. When the process is rescheduled for execution, the operating system restores the content of the registers, thus permitting the process to continue as though it were never interrupted.

PROCESS PRIORITIES AND PROCESS SCHEDULING

The priority of a process indicates the process's importance relative to other processes and is assigned at process creation. Priorities range from a high of 0 to a low of 254. Priorities and their normal (and recommended) uses are as follows:

Priority	Use
0-9	Operating system
10-64	System services
65-254	Application programs
255	Null process (see below)

The scheduler maintains a queue of the processes that are eligible to execute on a priority basis.

Rescheduling occurs when a system event makes a process executable because it has a higher priority than the one currently executing. Examples of system events include an interrupt from a device controller, X-Bus module, timer, or real-time clock, or a message sent from another process.

In most cases, the time interval between events is determined by the duration of a typical I/O operation. A process can lose control involuntarily only to a process of higher priority.

If a system event causes a message to be sent to an exchange at which a higher priority process is waiting, the operating system performs a context switch and reallocates the processor to execute the higher priority process.

When a system event occurs that makes a process eligible to execute, that process receives control of the processor until one of the following occurs:

- Another process with a higher priority preempts its execution.
- It voluntarily relinquishes control of the processor usually by calling the Kernel primitive, Wait.

If no other process has work to perform, the null process is given control of the processor. The null process, which is always ready-to-run, executes at priority 255, lower than any real process.

In real mode, the null process examines the checksum value the operating system creates for its code segment when it is bootstrapped. The null process ensures that the checksum is valid. A variance in checksums indicates that a program has modified code and results in the operating system crashing with status code 91 ("Operating system checksum error").

In protected mode, the null process exists only to simplify the algorithm of the operating system scheduler.

To give multiple programs with the same priority a fair share of system resources, processes with priorities in a predefined range are subject to time slicing. Processes within this range that have the same priority are executed in turn for intervals of 100 milliseconds each in repeating cycles. The priority range is a system build parameter.

PROCESS STATES

A process can exist in one of four states: running, ready, waiting, and suspended.

A process is in the running state when the processor is actually executing its instructions. Only one process can be in the running state at a time. Any other ready-to-run processes are in the ready state. Any number of processes can be in the ready state at the same time.

A process is in the waiting state when it waits at an exchange for a message to synchronize execution with other processes. A process enters the waiting state when it voluntarily issues the Kernel primitive, Wait, which specifies an exchange at which no messages are currently queued. Any number of processes can be waiting at a time. (See Chapter 29, "Interprocess Communication," for details.)

As soon as the running process waits, the highest priority process in the ready state is placed in the running state.

If a process is suspended, it is also placed in either the ready or waiting states, but it is never placed in the running state. A process is suspended, for example, if it is subject to time slicing and its time slice runs out before it has completed executing.

The relationship among process states is shown in Figure 28-1. Table 28-1 describes the transitions between process states and the events causing the transitions.

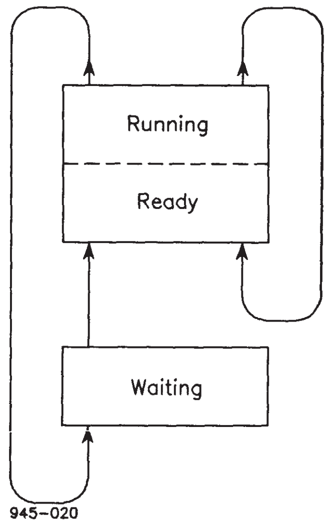


Figure 28-1. Process States

Table 28-1
PROCESS STATE TRANSITION

Transition From	To	Event
Running	Waiting	A process executes a Wait but no messages are at the exchange.
Waiting	Ready/	Another process sends a message to the exchange at which a process is waiting.
Running	Ready	A higher priority process leaves the waiting state and preempts this process.
Ready	Running	All higher priority processes enter the waiting state.

OPERATIONS

The process management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

CreateProcess — Creates a new process and schedules it for execution.

ChangePriority — Changes the priority of the calling process.

ChangeProcessPriority — Changes the priority of a process specified by the process ID.

SetDeltaPriority — Allows the dynamic changing of a process priority based on the memory partition it is executing in. SetDeltaPriority is used only by partition managing programs, such as the Context Manager.

SetDispMsw287 — Directs the Kernel to set the machine status word on every process context switch. (SetDispMsw287 is used by software that manages or emulates the Math Coprocessor only on 80286 or 80386 microprocessor-based operating systems.)

RescheduleProcess — Moves a process in front of all other processes of the same priority on the run queue.

QueryProcessNumber — Allows a process to determine its own process ID.

29 INTERPROCESS COMMUNICATION

The Interprocess Communication (IPC) facility synchronizes process execution and information transmission between processes through the use of messages and exchanges. A process can communicate with another process in its own partition or in another partition.

AN IPC EXAMPLE

When you write a statement in your program, like

```
erc := OpenFile(ADS fh, ADS lsFileSpec[1],
         lsfileSpec.len, NULL, 0, 'mr');
```

your purpose might be to use the file handle (fh) returned to refer to the open file in a subsequent read or write statement. You assume that the statement will just work.

What you are actually doing is using the request procedural interface, one of the most common applications of IPC. A request procedural interface is an operating system routine that uses IPC to communicate the requested information in the statement you wrote to the file system service. IPC is used again to return the response information (file handle) from the file system to your program.

Your program is the client. Any program, including another system service, can be a client if it makes a request of a system service.

IPC provides the means by which a client and a system service communicate with each other. The communication is in the form of a special IPC message called a request block. The messenger facilitating the communication is the operating system Kernel.

WHAT REALLY HAPPENS

When your program calls OpenFile, your program enters the operating system's request procedural interface routine. (See Figure 29-1.)

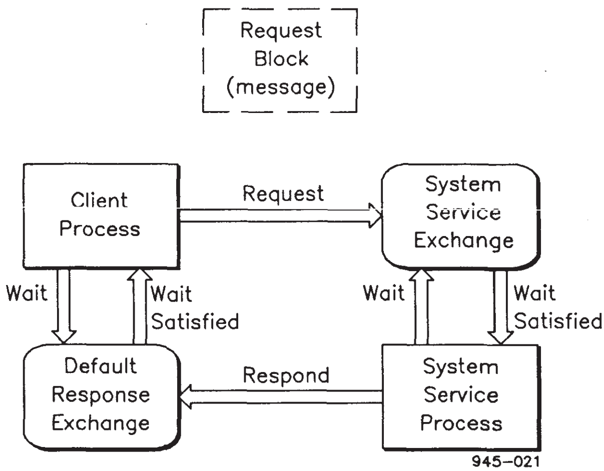


Figure 29-1. Interaction of Client and System Service Processes

REQUEST PROCEDURAL INTERFACE

The request procedural interface

1. builds the request block message for OpenFile in the client process's memory, copying information provided in the OpenFile statement

2. calls Request to route the request block to the system service exchange

3. places the client in a wait state at its default response exchange

(For details, see "Using the Request Procedural Interface," later in this chapter.)

SYSTEM SERVICE

The system service waits at its service exchange for request blocks requesting its service. (See Figure 29-1.) Upon receipt of the request block, the system service verifies the information in the request block message.

If the information is valid, the system service performs its service and answers the client's request by filling in the request block with its response and a normal status code (erc). If the request is invalid, however, it places a status code indicating an error in the request block.

Upon completion Of these functions, the system service calls Respond to route the request block back to the client's exchange. The client's wait is satisfied, and it is ready to execute its next instruction.

Figure 29-2 compares the processing flow of the client process to the system service process.

SUMMARY

The request procedural interface uses IPC to send your request to the system service. You assume the information you requested will be available to use in your next program instruction.

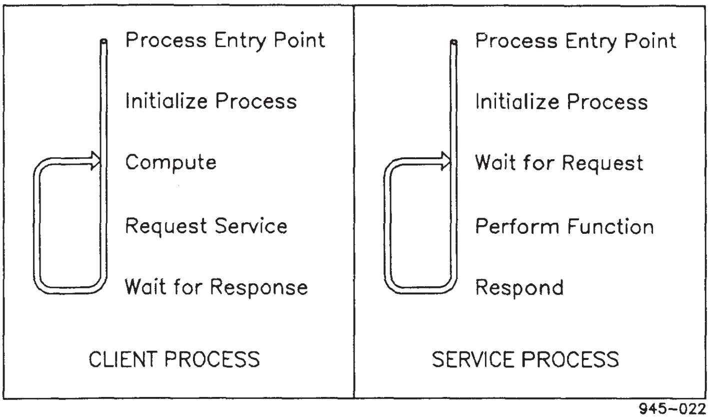


Figure 29-2. Processing Flow of Client and System Service Processes

The request procedural interface is a convenient way to access system services. It is compatible with BASIC, COBOL, FORTRAN-86, PL/M, C, and Pascal, as well as assembly language. For this reason, it is a common IPC application. IPC has other applications as well.

OTHER IPC APPLICATIONS

To a great extent, the power of the operating system results from its IPC facility. IPC supports three multiprogramming capabilities:

- communication
- synchronization
- resource management

COMMUNICATION WITHIN AN APPLICATION PARTITION

Communication, the most elementary interaction between processes, is the transmission of data from one process to another by means of an exchange. Figure 29-3 shows an example of communication. Process A and Process B are executing within the same application partition. Process A sends a message to Exchange X, and Process B waits for a message at that exchange.

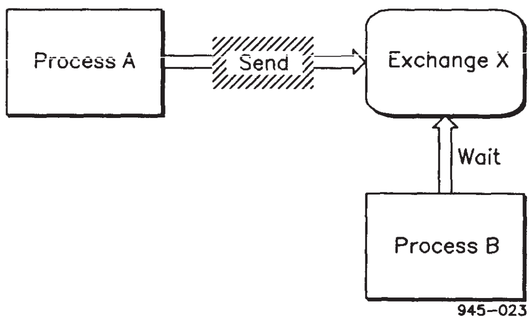


Figure 29-3. Communication Between Processes

COMMUNICATION BETWEEN APPLICATION PARTITIONS

IPC is used in a special way by application programs that want to communicate with programs in other application partitions. This is done using the Intercontext Message Server (ICMS).

The requesting application program sends an IPC message to ICMS. ICMS, in turn, uses IPC to forward the message to the receiving program. If the receiving program is swapped out of memory, ICMS holds the message until the receiving program is resident again to accept it.

Figure 29-4 shows how IPC is used with ICMS. (For details on ICMS, see the Context Manager/VM Manual.

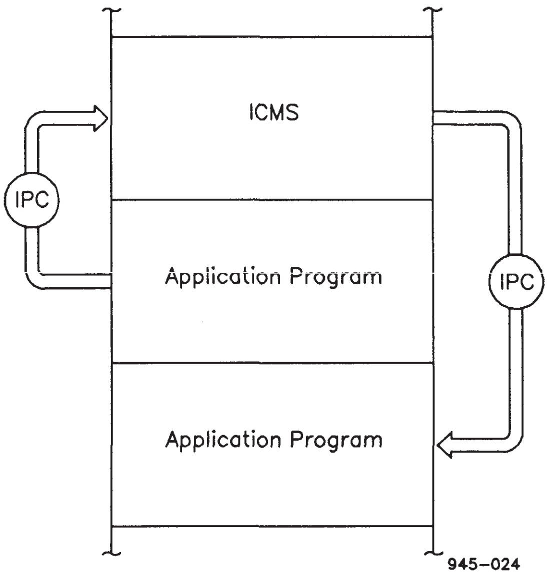


Figure 29-4. How IPC Is Used with ICMS

SYNCHRONIZATION

Synchronization is the means by which a process ensures that a second process has completed a particular item of work before the first process continues execution. Synchronization between processes and the transmission of data between processes usually occur simultaneously.

As shown in Figure 29-5, Process A sends a message to Exchange Y, requesting that Process B perform an item of work. Process A then waits at Exchange Z until Process B has completed the work. This synchronizes the continued execution of Process A with the completion of an item of work by Process B.

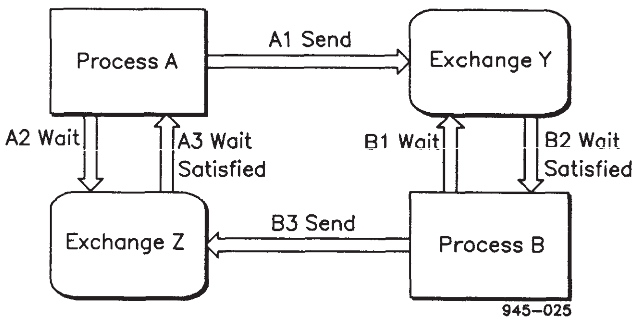


Figure 29-5. Synchronization

RESOURCE MANAGEMENT

In a multiprogramming environment, resource management is the means of sharing resources among processes in a controlled way. For example, several processes may need to use the printer; however, only one process can use the printer at a particular time.

One way to control a resource is to establish a process to manage it. Then, only the managing process accesses the resource directly. Other processes access the resource indirectly by sending messages to the process that performs the chosen function. System service processes, which manage resources such as files, devices, and memory, are implemented by means of an analogous mechanism.

As an example of resource management, a pool of buffers may be available in a partition to be shared by processes performing I/O. When a buffer is not in use, a message indicating the number of the available buffer can be queued at an exchange set up to manage allocation of the buffers.

As shown in Figure 29-6, Process A waits at the exchange, picks up a message indicating that buffer 1 is available, and proceeds to use buffer 1. Process B waits at the exchange and is allocated buffer 2. Process C waits and is allocated buffer 3, the last available buffer. Then, when Process D waits, no buffer is available. Process D, therefore, must wait at the exchange until one of the processes using a buffer completes and returns a buffer available message back to the exchange.

WHY UNDERSTAND IPC?

You can write statements like the OpenFile example at the beginning of this chapter, and IPC will work for you automatically.

At some point, however, you may want use some more advanced programming techniques to increase the efficiency of your program or to write your own system service. In these cases, you need to understand the mechanism behind IPC.

REQUEST CODES

A request code is a 16 bit value that uniquely identifies a chosen system service operation. For example, the request code for the OpenFile operation is 4.

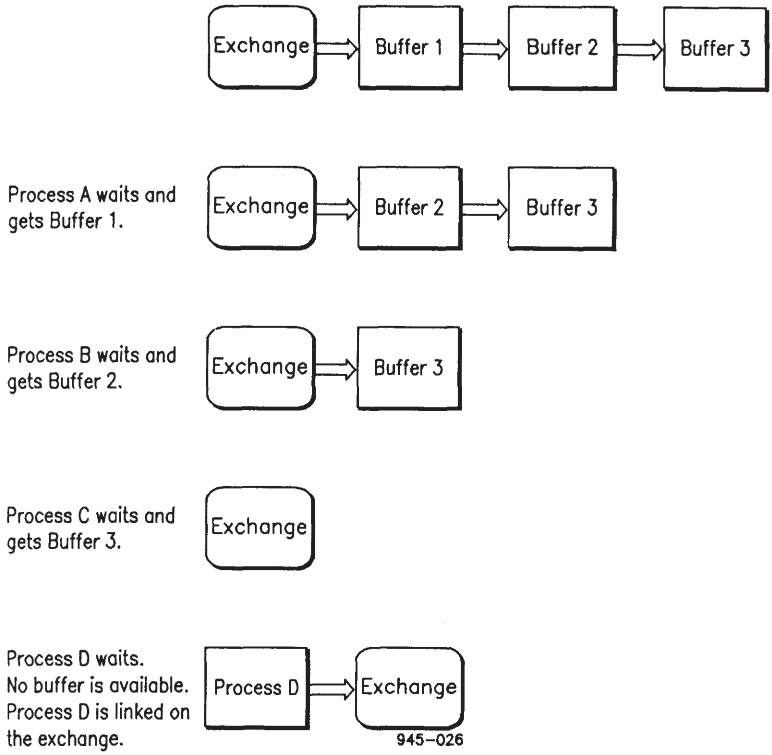


Figure 29-6. Buffer Management

The request code is used in IPC both to route a request to the exchange of the appropriate system service and to specify which of its several functions the request refers to.

If you are writing a system service, you need to assign request codes to the requests you define to be performed by that service.

The operating system has a number of built-in system services of its own, such as the file system and keyboard. Request codes for operating system requests are listed in Appendix D of the CTOS/VM Reference Manual.

There are 64K possible request codes divided into 16 byte request levels of 4K bytes each. To use the request procedural interface and validity checking structures, a request must be defined by a request code in an even-numbered level.

The request levels are shown in Table 29-1. Levels 0 through B are reserved for internal use.

NOTE: You can reserve request codes in Level A or B by contacting Convergent Technologies, Technical Support.

Level 0 must always be linked into the operating system, and it must contain definitions for all requests used by the operating system up to the point at which system initialization is complete. Levels 1 through F can be loaded at system boot time from one or more files contained on the system disk.

Initialization request structures also include tables for the system requests used for termination, abort, and swapping. System requests are defined by odd-level request codes. (For details on system requests, see Chapter 31, "System Services Management.")

Table 29-1

REQUEST CODE LEVELS

Level	Hexadecimal Values
0	0000 to 0FDF and FFE0 to FFFF
1	1000 to 1FFF
2	2000 to 2FFF
3	3000 to 3FFF
4	4000 to 4FFF
5	5000 to 5FFF
6	6000 to 6FFF
7	7000 to 7FFF
8	8000 to 8FFF
9	9000 to 9FFF
A	A000 to AFFF
B	B000 to BFFF
C	C000 to CFFF
D	D000 to DFFF
E	E000 to EFFF
F	F000 to FFDF

INTERPROCESS COMMUNICATION (IPC) COMPONENTS

The basic components of IPC are the

- Kernel primitives
- exchanges
- message (usually a request block)

The Kernel primitives are used to send and receive messages.

Actually, the Kernel primitives are inherently part of the Kernel's code: calling a primitive wakes up the Kernel to perform some action.

The Kernel acts as a messenger by sending messages to their appropriate destinations. When a system service waits to receive a message at its designated exchange, the Kernel checks to see if any messages are there to be serviced.

The message conveys information and provides synchronization between processes. A 4 byte data item is communicated between processes. This data item is usually the memory address of a larger data structure that is called the message.

A message is actually sent to an exchange rather than directly to a process. An exchange can be thought of as serving the function of a post office where postal patrons (processes) go to mail (send) letters (messages) or pick up (wait/check for) letters.

In the same way that the postal patron drops a letter in the mail box and then walks away trusting that the letter will be delivered, a process sends a message and then continues executing without further regard.

A postal patron who is expecting an important letter can periodically go to the post office to check whether it has arrived. If the letter is especially important, the patron can wait in the post office for the letter to arrive.

A process has analogous mechanisms* available when it expects to receive a message. It can check periodically whether a message is posted at (queued on) an exchange, or it can wait at the exchange for the arrival of a message. Because computers are much faster than the postal service, it is usually more appropriate to wait for a message than to check for its arrival.

A request block is a special type of message formatted according to specific conventions and used by all interprocess communications to the operating system.

THE KERNEL PRIMITIVES

KERNEL PRIMITIVES FOR SENDING A MESSAGE

The Kernel primitives for sending a message include

- Request
- Respond
- Send
- ForwardRequest
- RequestDirect

(Note that RequestRemote is an additional Kernel primitive that is used to send a message to a remote processor on the SRP. This operation is described in Chapter 30, "Inter-CPU Communication.")

Request and Respond

Request is used by a client to direct a request to a system service. Respond is used by system services to respond back to the client. To provide a meaningful environment, each Request must be answered by a matching Respond.

The Request primitive directs a request for a system service from a client process to the service exchange of the system service process. (See Figure 29-7.) Before the Request is issued, the data required for the system service must be arranged in a request block in the client's memory.

A request block is a special type of message formatted according to specific conventions and used by all IPCs to the operating system.

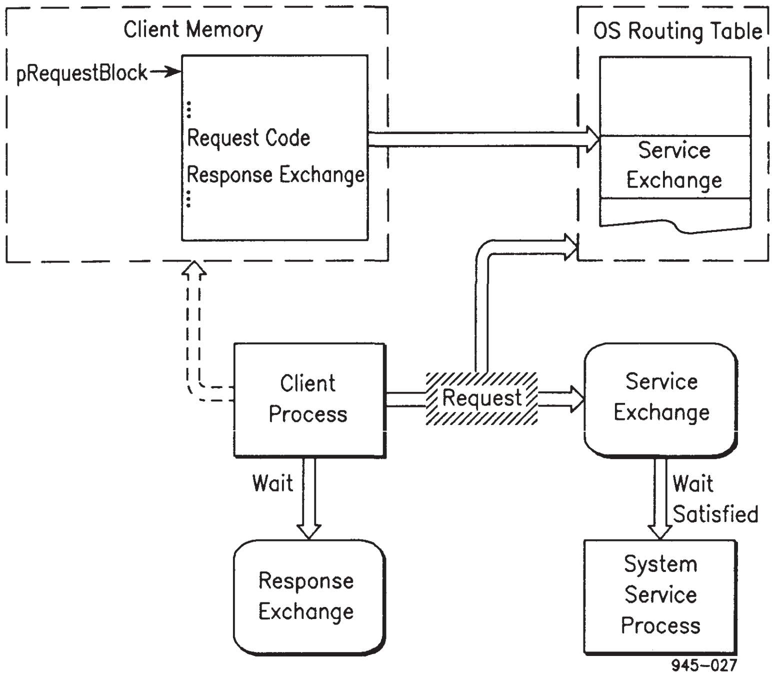


Figure 29-7. Request Primitive

Request does not accept an identification of an exchange as a parameter. It uses the request code of the request block as an index into an operating system request routing table to determine the appropriate service exchange. Request routing tables reside in the System Image and translate request codes to service exchanges.

The Respond primitive is used by a system service process to send an answer to a request back to the response exchange of a client process. (See Figure 29-8.)

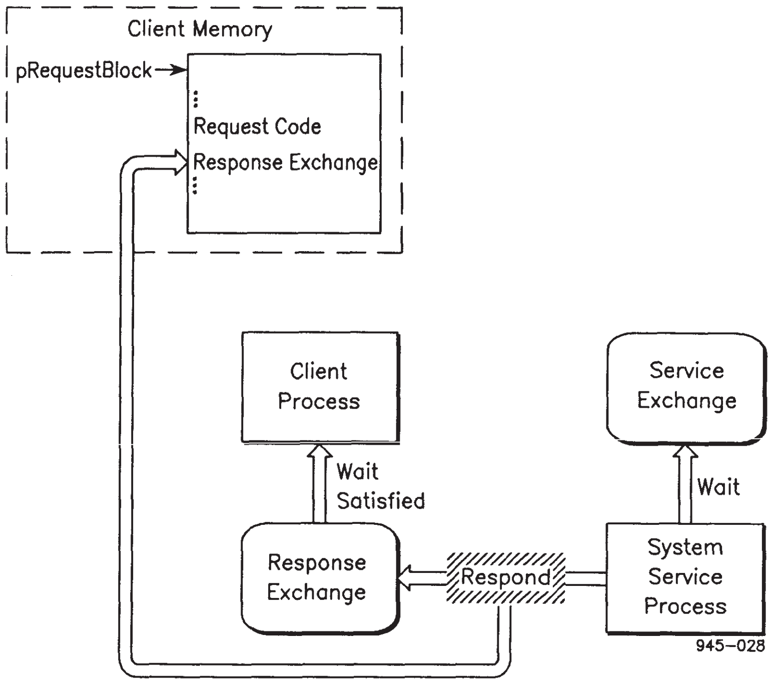


Figure 29-8. Respond Primitive

The only parameter to the Respond primitive is the memory address of the client's request block. That is, the system service must use (as a parameter to Respond) the same memory address that the client used as a parameter to Request. The exchange to which the response is directed is determined by the response exchange field of the request block.

Send

The Send primitive, unlike Request or Respond, accepts any 4 byte field as a parameter. This is usually, but not necessarily, the address of a message. Send does not require a formalized request block message, nor does it require a matching response.

Send should be used by processes within the same partition (user number).

NOTE: Send should not be used to send messages between programs in different application partitions. This is done in a special way by ICMS. For details, see "Communication Between Application Partitions," earlier in this chapter.

Figure 29-9 shows how Send works in the communication between Process A and Process B in the same partition. Process A sends a message to Exchange X, and Process B waits for a message at that Exchange.

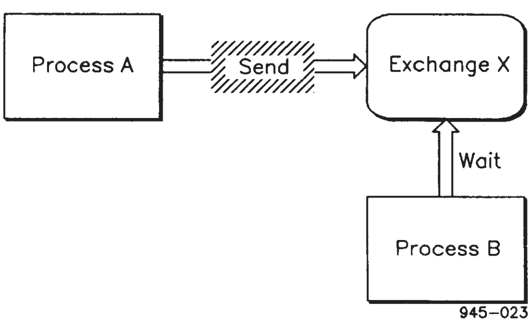


Figure 29-9. Send Primitive

ForwardRequest and RequestDirect

The ForwardRequest and the RequestDirect primitives are used by special types of system services called filters, which intercept messages destined to other system services. (See "Filters," later in this chapter. For details on how ForwardRequest and RequestDirect are used by filters, see Chapter 31, "System Services Management.")

KERNEL PRIMITIVES FOR RECEIVING A MESSAGE

The Kernel primitives for receiving a message are wait and Check.

Wait

The Wait primitive checks whether a message is queued at the specified exchange. System services wait at system service exchanges until their services are requested. Clients wait at exchanges to synchronize their execution with the completion of a system service they request. (See Figure 29-10.)

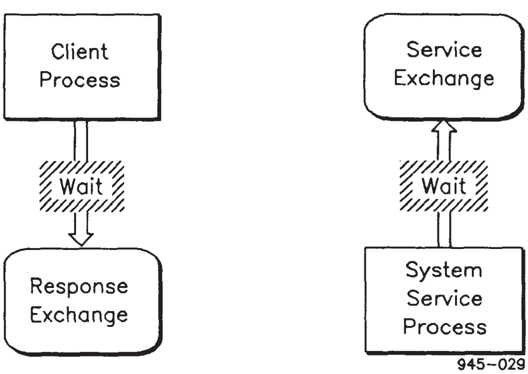


Figure 29-10. Wait Primitive

In the context of all Kernel primitives for sending messages except Send [that is, Request, Respond, ForwardRequest, RequestDirect, and RequestRemote (described in Chapter 30)], the message queued at the exchange is always a request block.

The details on how a process waits at an exchange to receive messages are described and illustrated in detail in the following sections, later in this chapter:

- "Sending a Message to an Exchange"
- "Waiting for a Message at an Exchange"

The request procedural interface uses the Wait primitive rather than Send.

Check

The Check primitive checks whether a message is queued at the specified exchange. If one or more messages are queued, the message that was queued first is removed from the queue and its memory address is returned to the calling process. If no messages are queued, status code 14 ("No message available") is returned.

Unlike the Wait primitive, the Check primitive never causes the calling process to wait.

(For details and examples of how to use the Check primitive, see "Accessing System Services," later in this chapter.)

THE EXCHANGE

A message is actually sent to an exchange rather than directly to a process. An exchange functions as a message center.

An exchange is referred to by a unique, 16 bit integer. An exchange consists of two first-in, first-out queues. One is a queue of processes waiting for a message,- the other is a queue of messages that are ready for processes to poll.

Note that either messages or processes (not both) can be queued at an exchange at any given time.

Only the address of the message, not the message itself, is sent to an exchange. This minimizes overhead. Therefore queueing a number of messages at the same exchange requires very little execution time or memory. IPC places no restriction on the size and content of the message. The receiving process must be programmed to use IPC to wait or to poll (check) for the availability of a message.

TYPES OF EXCHANGES

A response exchange is the exchange at which the client waits for the system service's response. The response exchange field in the request block directs the response to the correct exchange.

The default response exchange is a special case of response exchange. This exchange is automatically used as the response exchange whenever a client process uses the request procedural interface to a system service. A run file is assigned a default response exchange when it is first loaded into memory. Each new process created in a program must be allocated a unique default response exchange.

Direct use of the default response exchange (that is, using it when you are not using the request procedural interface) is not recommended. The use of the default response exchange is limited to requests of a synchronous nature. That is, the client, after specifying the exchange in a Request, must wait for a response before specifying the exchange again in another Request.

A service exchange is an exchange that is assigned to a system service at system build or when the system service is dynamically installed. The system service waits for requests for its services at its service exchange.

EXCHANGE ALLOCATION

Exchanges are allocated in three ways:

- Exchanges for certain built-in system services are allocated at system build.
- Exchanges can be dynamically allocated and deallocated using the AllocExch and DeallocExch operations.
- A unique default response exchange must be allocated for each new process created in a program that will use the request procedural interface. A process can determine the identification of its own default response exchange using the QueryDefaultRespExch operation.

Upon termination, the exchanges allocated to the terminating program are deallocated.

SENDING A MESSAGE TO AN EXCHANGE

When a process sends a message to an exchange, one of two actions results at the exchange (see Figure 29-11):

- If no processes are waiting, the message is queued.
- If one or more processes are waiting, the process that was queued first is given the message and is put in the ready state. If this ready process has a higher priority than the sending process, a context switch occurs, and the ready process becomes the running process. The sending process is placed in the ready state and loses control until it once again becomes the ready process with the highest priority. [The process states (ready, running, and waiting) are described in Chapter 28, "Process Management."]

After a message is queued at an exchange, the sending process must not modify it. A system service, for example, may have temporarily replaced the response exchange in a waiting client's request block with its own service exchange to request a resource from another system service.

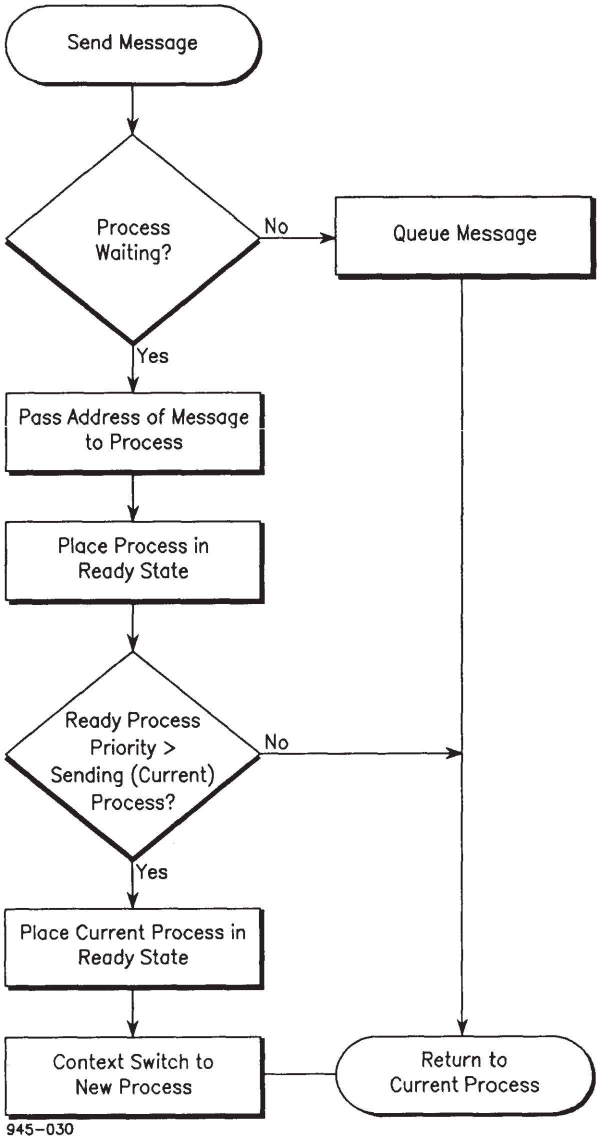


Figure 29-11. Sending a Message to an Exchange

WAITING FOR A MESSAGE AT AN EXCHANGE

When a process calls Wait and waits for a message at an exchange, one of two actions results at the exchange (see Figure 29-12):

- If no messages are queued, the calling process is placed in the waiting state, and its Process Control Block (PCB) is queued at the exchange until a message is sent.

- If one or more messages are queued, the message that was queued first is removed from the queue and its memory address is returned to the process, which then resumes execution.

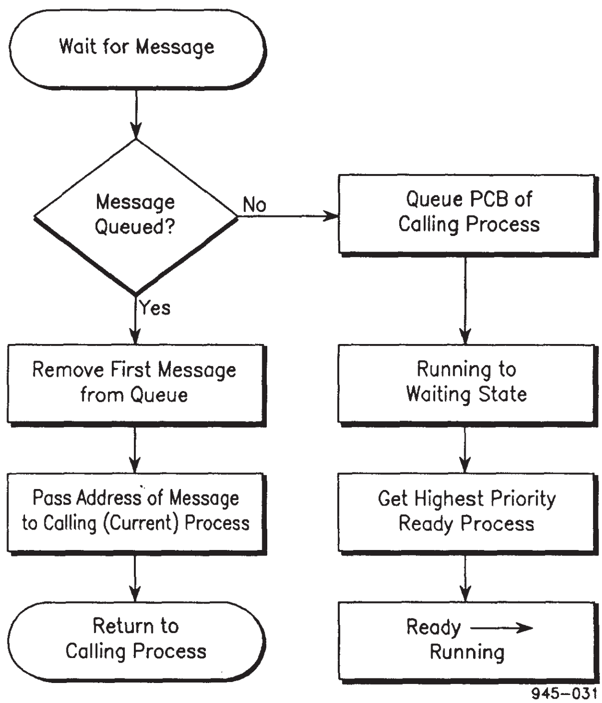


Figure 29-12. Waiting for a Message at an Exchange

EXCHANGE QUEUES

Either processes or messages, but not both, can be added to a queue at an exchange at any given time.

Messages are queued using link blocks. A link block is a 6 byte structure containing the address of the message (or the message itself) in the first 4 bytes and the address of the next link block (if any) in the last 2 bytes.

Figure 29-13 shows how messages are queued at an exchange.

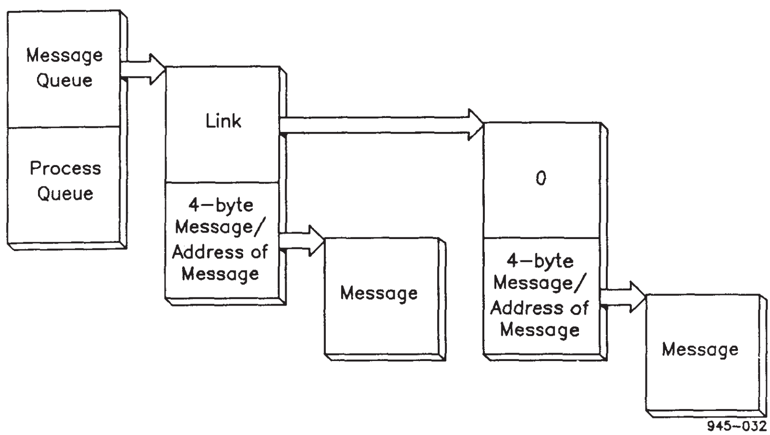


Figure 29-13. Messages Queued at an Exchange

Processes are queued at an exchange by linking through a field that is reserved for this purpose in each PCB.

Request blocks are self-describing and consist of

- a standard header
- control information specific to the request
- a routing code
- descriptions of the request data items
- descriptions of the response data items

STANDARD HEADER

The format of the standard request block header is shown in Table 29-2.

Table 29-2

FORMAT OF A REQUEST BLOCK HEADER

Offset	Field	Size (bytes)
0	sCntInfo	1
1	RtCode	1
2	nReqPbCb	1
3	nRespPbCb	1
4	userNum	2
6	exchResp	2
8	ercRet	2
10	rqCode	2

wnere

sCntInfo — Is the number of bytes of control information. Control information is the data after the request block header and before the first request address/size (pb/cb) pair.

RtCode — Is a routing code placed in the request block by the operating system for routing requests. The default value of this field is 0.

nReqPbCb — Is the number of request address/size pb/cb) pairs,

nRespPbCb — Is the number of response address/maximum size (pb/cbMax) pairs.

userNum — Is a 16 bit integer that uniquely identifies the client's partition and the resources with which it is associated.

Each application partition has a unique user number. The processes in an application partition share the same user number. A process can obtain its user number by means of the GetUserNumber operation. (GetUserNumber is described in Chapter 3, "Operations," in the CTOS/VM Reference Manual.)

If the userNum field contains 0, the operating system substitutes the user number of the client initiating the request.

Figure 29-14 shows how processes are queued at an exchange.

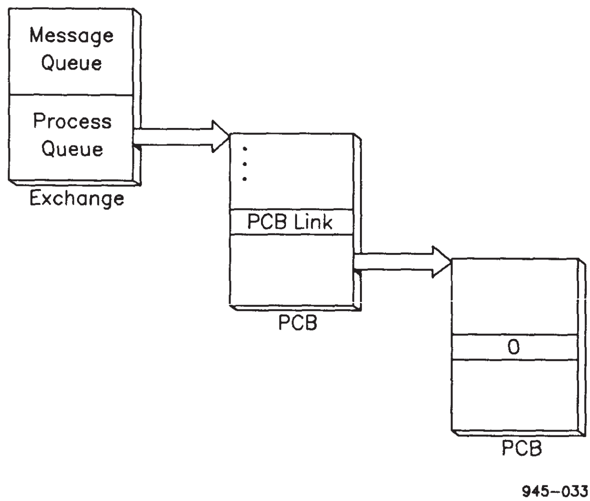


Figure 29-14. Processes Queued at an Exchange

THE MESSAGE

A message conveys information and provides synchronization between processes. A 4 byte data item is communicated between processes. This data item is usually the memory address of a larger data structure, which is called the message. The interpretation of the 4 byte field is by agreement of the sending and receiving processes. Typically this field is the memory address of a request block.

The message can be in any part of memory that is under the control of the sending process. By convention, control of the memory that contains the message is passed along with the message.

A request block is a special type of message formatted according to specific conventions and used by all interprocess communications to the operating system. It is a data structure provided by the client, containing the specification and the parameters of the desired system service. A request block contains a request code field, a response exchange field, and several other fields; IPC is used most commonly with messages in this format.

This format is described in "Request Block Format," which follows.

REQUEST BLOCK FORMAT

The Request primitive initiates the request for a system service; the Respond primitive initiates the response. This structure provides

- guaranteed matching of Requests and Responds
- opportunity to redirect requests for system services to other system services
- opportunity to redirect requests for system services to the master of a cluster configuration or over CT-Net

The format of a request block is designed to pass information between a client and a system service. It provides for the transparent migration of system services between standalone, cluster, and network configurations.

exchResp	Is the response exchange of the client. A special exchange called the default response exchange is allocated when a run file is loaded into memory. It is used by the request procedural interface and should not be used explicitly. The AllocExch operation should be used to allocate exchanges.
ercRet	Is the status code (erc) returned by the system service.
rqCode	Is a request code, a 16 bit value that uniquely identifies the selected system service. The request code is used both to route a request to the appropriate system service exchange and to specify to that service which of its several functions is being requested.

CONTROL INFORMATION

Control information is specific to each request. The sCntInfo field contains the number of bytes of control information transmitted from the client to the system service.

ROUTING CODE

The routing code (RtCode) field consists of 1 byte that allows the Kernel and agents to route a request from a program anywhere in the network, even if the request is undefined in the client process's workstation operating system. The default value of this field is 0.

This field is important to you if you are defining requests for a system service. (See "Routing Requests," later in this chapter, for more information about the RtCode field. Also see Chapter 31, "System Services Management," for details on defining requests for user-written system services.)

REQUEST DATA ITEM

Each request data item descriptor consists of the following:

- the 4 byte memory address of the request data item
- the 2 byte size of the item

The total size (in bytes) of the request data item descriptors is 6 times nReqPbCb. Request data items are transmitted from client to system service. As an example, a request data item can be a name of a file to be opened.

RESPONSE DATA ITEM

Each response data item descriptor consists of the following:

1. the 4 byte memory address of the area into which the response data item is to be moved by the system service
2. the 2 byte maximum allowable byte count of the response data item

The total byte size of the response data item descriptors is 6 times nRespPbCbMax. Response data items are transmitted from system service to client.

A response data item is information that the system service wants the client to know about, such as the file handle (fh) returned by the OpenFile operation or the number of bytes it wrote to the client's buffer in a Write operation.

EXAMPLE REQUEST BLOCK

Figure 29-15 shows the request block for the Write operation.

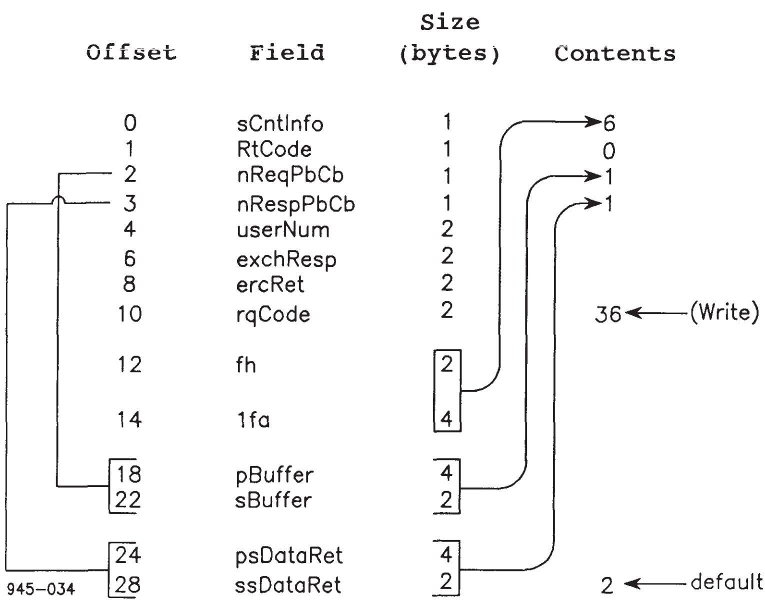


Figure 29-15. Request Block for the Write Operation

Note that the request block header is the standard format described in "Request Block Format," earlier in this chapter. The Contents column shows values for some of the request block fields, for example:

- The request code (rqCode) for Write is 36.

- The 6 bytes of control information (sCntInfo) consist of

  - the file handle (fh) returned from a previous OpenFile operation (2 bytes)

  - the logical file address (lfa) of the sector into which the data is to be written (4 bytes)

- The routing code (RtCode) field contains 0 until the request is issued. File handle or file specification information is defaulted to this field. (See "Routing by File Handle" and "Routing by File Specification," later in this chapter, for more information about this field.)

- A single request data item (nReqPbCb) is described by pBuffer/sBuffer.

  - pBuffer is the memory address of a buffer containing the data to be written.

  - sBuffer is the count of bytes to be written.

- A single response data item (nRespPbCb) is described by psDataRet/ssDataRet.
  - psDataRet is the memory address of a word into which the count of bytes successfully written is returned.
  - ssDataRet indicates the size of the word (2 bytes) into which the count of bytes written is returned. If the request procedural interface is used, it automatically supplies this value.

ACCESSING SYSTEM SERVICES

System services can be accessed

- indirectly, by the request procedural interface
- directly, by the Kernel primitives, Request and Wait (or Check)

USING THE REQUEST PROCEDURAL INTERFACE

Using the request procedural interface is convenient because it automatically constructs a request block and issues the Request and Wait primitives for you.

Except for the ReadAsync and WriteAsync procedures, the request block is constructed on the stack of the client process.

Most request procedural interfaces to system services do not provide any overlap between computation by the client process and execution of the system service. Because Read and Write are the system services for which the overlap of computation and execution of the system service is most ideal, however, the operating system provides the ReadAsync, CheckReadAsync, WriteAsync, and CheckWriteAsync operations.

These operations allow the client to initiate an I/O operation and then to compute and/or initiate other I/O operations before checking for the successful completion of the original one.

A special form of request procedural interface called the alternate request procedural interface provides a further convenience to clients that want to request a service on behalf of a different user number. The very nature of a system service may require that it issue the same request repeatedly for different user numbers.

The alternate request procedural interface requires only that you add the letters **Alt** and one extra parameter (userNum) to the parameters in the request statement. To write to a file, for example, you would write a statement of the form

```
erc=AltWrite(userNum, fh, pBuffer, sBuffer,
    lfa, psDataRet);
```

where userNum is the user number on behalf of which the request is being issued.

USING THE KERNEL PRIMITIVES DIRECTLY

Using the Request and Wait (or Check) primitives is more powerful than using the request procedural interface: it allows a greater degree of overlap between multiple I/O operations and computation by the client process.

For example, if you use the Check primitive instead of Wait, your program can continue executing some other function, such as updating the video to reflect current statistics. Execution is asynchronous with the return of the request.

To use the Kernel primitives directly, you must

- Call AllocExch to allocate a response exchange for the request block. You must not use the default response exchange.

- Build a request block in your program. Static information, such as the request block header, can be defined during program initialization. (See "Request Block Format," earlier in this chapter.) Parameters, such as buffers that change during program execution, must be updated each time the block is reused.

- Call the Kernel primitives, Request and Wait (or Check).

NOTE: Save the request block response exchange in a variable. Do not pass the response exchange in the request block as a parameter to a Kernel primitive. Modification of an outstanding request block can result in conflict if, for example, the request block is redirected to a filter.

If you need to synchronize program execution with the return of the request block information, you can call Request and then issue a Wait for the response. Wait suspends process execution until the request block returns.

If your program does not depend on the information being returned immediately, you can issue Check periodically. Check tells whether a request block has returned without suspending program execution.

You may reach a point at which you must synchronize execution with the return of the request block. Your program, for example, may be performing a heavy computation, occasionally needing to write output to a disk file. When it is time to write, you can call Wait, specifying the response exchange in your request.

When the request block returns, you can safely use it in another Write request. This may require adjusting the addresses and sizes of the request block buffers.

If more than one request block is outstanding, you must ensure that it is the correct one. To do this, your program can verify the request code or the address of the request block. (The request block address can be verified using the FComparePointer operation.)

CLUSTER/NETWORK COMMUNICATION

The operating system provides for cluster workstation as well as CT-Net configurations.

A cluster configuration consists of cluster workstations and a master. A cluster provides communication and resource sharing within a localized area, such as a building.

A CT-Net configuration consists of nodes connected by communications lines over long distances. A node is a junction in a network (such as a workstation or a processor board on the SRP), where communication lines originate and/or terminate. CT-Net, thus, provides for communication and resource sharing over a wide area network.

CLUSTER CONFIGURATION

The master of a cluster configuration can be a master workstation or an SRP. The master provides resources, such as file system management and queue management, for all workstations in the cluster. In addition, it concurrently supports its own program processing as well as user-written, multiuser system services.

Essentially the same operating system executes in each cluster workstation as in a master workstation or in the combined processors of the SRP. A cluster workstation can have its own local file system, or it can use the file management system of the master.

In the cluster configuration, IPC is extended to provide transparent access to system services that execute at the master. While some services (such as queue management, 3270 Terminal Emulator, and database management) migrate to the master, others (such as video management and keyboard management) remain at the cluster workstation.

CLUSTER WORKSTATION AGENT

In a cluster workstation, however, if the function is to be performed at the master, the request block is queued at the exchange of the Cluster Workstation Agent. The Cluster Workstation Agent converts interprocess requests to interstation messages for transmission to the master. The Cluster Workstation Agent is included at system build in a System Image that is to be used on a cluster workstation.

MASTER AGENT

The System Image used at the master is built to include a corresponding service process: the Master Agent. The Master Agent reconverts the interstation message to an interprocess request that it queues at the exchange of the system service on the master that actually performs the intended function. Note that the operating system request code routing table that translates request codes to service exchanges at the master is necessarily different from the table at the cluster workstation. When the system service at the master responds, the response is routed through the Master Agent, the high-speed data link, and the Cluster Workstation Agent before being queued at the client's response exchange in the cluster workstation that was specified in the request block.

The format of request blocks is designed to allow the Cluster Workstation Agents and Master Agents to convert between interprocess requests and interstation messages very efficiently and with no external information. Because request blocks are completely self-describing, the agents can transfer requests and responses between the master and cluster workstations without any knowledge of what function is requested or how it is to be performed.

CT-NET

CT-Net extends the CTOS/VM resource sharing capability to permit sharing of system resources among nodes in a network.

A system service, for example, can be installed at a master and accessed by remote nodes over CT-Net or by workstations in a localized cluster.

CT-Net consists of two system services that issue and execute requests on behalf of clients and system services at local or remote network nodes. These services are the Net Agent and the Net Server.

The Net Agent receives requests destined for system services located at remote nodes and forwards these requests to the remote nodes.

The Net Server responds to requests from remote Net Agents. The Net Server receives a request block from the Net Agent, executes the request on behalf of the remote client, and returns the response to the originating Net Agent.

(For more information on the CT-Net environment and network configurations, see the CT-Net Reference Manual.)

ROUTING BY FILE HANDLE

A request can be routed by handle. A handle is a 16 bit identifier assigned by a system service and used to reference a resource. A file handle is returned by the file system to refer to a file opened by OpenFile. Future requests, such as Read or Write, identify the open file by passing the file handle back to the file system.

The file system sets all of the bits except the high-order bit in the file handle.

A file handle returned over CT-Net from a remote node has the high-order bit set by a Net Agent to indicate that the file system returning it is remote. Any request that references a file handle with the high-order bit set can, thus, be routed to the Net Agent.

RULES FOR ROUTING BY HANDLE

A request routed by file handle must adhere to the following rules:

- If a client issues a request by file handle, the request must be defined to include the handle in the first word of the request block control information. (See "Request Block Format," earlier in this chapter.)

- If a system service issues a response to return a file handle, the request must be defined to return the handle in the first address/size (pb/cb) pair of the request block. (See "Request Block Format," earlier in this chapter.) OpenFile, for example, returns a handle.

THE FILE HANDLE

The bits in the file handle mean the following:

Bits	Meaning
15	The high-order bit identifies a CT-Net handle if it is set. Any request that uses a CT-Net handle is routed to the Net Agent.
14 to 10	The next 5 bits identify the File Processor on the SRP and the verify code. The verify code is the number of times that the master has been rebooted.

9 to 0	The remaining bits identify the File Control Block for the file.

The Net Agent translates these bits to a unique number. The Net Agent uses this number to associate the handle with a session and a remote handle,

A session is a connection between two nodes initiated by the Net Agent. When the Net Agent receives a request routed by handle, it uses the number to find the session and the remote handle.

ROUTING BY FILE SPECIFICATION

Requests can be routed by file specification. File specifications are described by address/size (pb/cb) pairs in the request block. (See "Request Block Format," earlier in this chapter.)

RULES FOR ROUTING BY SPECIFICATION

A request routed by file specification must adhere to the following rules:

- Node names are from 1 to 12 characters long and can be any combination of alphanumeric characters. Each node must be given a unique name and address.

  Two node names are reserved:

Name	Meaning
local	Is ignored. Other routing information is used.
master	Route this request to the master.

- A request can have a maximum of two file specifications. The first file specification must be in the first request pb/cb pair; the second (if any), in the third pb/cb pair.
- If a file specification has a password associated with it, the password must be specified by the pb/cb pair immediately following the file specification. A second instance of the file specification must also have the password.

EXPANDING FILE SPECIFICATIONS

The Cluster Workstation Agent expands any incomplete specifications before sending a request to the master. (The master does not have a copy of the User Control Block and therefore cannot expand the specifications itself.)

Expanding a specification involves adding default path information from the User Control Block. The information that must be added depends on the type of the specification.

File specifications are expanded as follows:

Specification Type	Method of Expansion
FileSpec	Expands everything to the left of the file name, that is, the de-fault file name prefix, the de-fault directory, the default volume, and the default node, for example:

{node}[volname]<dirname>fileName

Specification Type	Method of Expansion
DevSpec	Expands everything to the left of the volume name, that is, the default node, for example:

{node}[volname]

Specification Type	Method of Expansion
DirSpec	Expands everything to the left of the directory name, that is, the default volume and the default node, for example:

{node}[volname]<dirname>

FileSpec2	The same as FileSpec, but the request contains two specifications to expand.
FileSpecP2S2	The same as FileSpec, but the specification occurs in the third request pb/cb pair, instead of the first.

THE ROUTING CODE

The routing code (RtCode) field is a 1 byte field of the request block used by the Kernel and agents to route requests. It determines

- whether the request is to be routed by specification or by handle
- for requests routed by specification, the location of the specification in the request block
- for requests routed by specification, the method of expansion

When the request is issued, the routing code (RtCode) field defaults to the Net Routing information. Net routing information is used to define a file system request.

Table 29-3 describes the Net Routing information

Table 29-3

NET ROUTING INFORMATION

Value (Decimal)	Token	Description
1	RW	This request is a read or write and may have to be broken up into small requests.
2	OpenFh	This request opens a resource. The first response pb/cb pair of this request returns a handle that is used later by other requests to refer to this resource.
4		(Reserved)
8	SpecPW	All file specification pb/cb pairs are followed by password pb/cb pairs. If SpecPW is set and there is no specification to expand (rSpec = 0 or rSpec > 5), the first pb/cb pair is a password to expand (for example, ChangeOpenMode).
16	rFh	Route this request by handle. This handle was returned by a request defined as OpenFh.
32 through 224		(See Table 29-4.) Combinations of bits 5 through 7 of the 1 byte RtCode field indicate the methods of file specification expansion. One combination indicates closing of a resource.

Table 29-4 describes each bit combination of bits 5 through 7 of the RtCode.

Table 29-4

BIT COMBINATIONS FOR BITS 5 THROUGH 7 OF RTCODE

Value (Decimal)	Token	Description
0		No specification routing.
32	DevSpec	Route by DevSpec.
64	DirSpec	Route by DirSpec.
96	FileSpec	Route by file specification.
128	FileSpec2	Route by file specification. (The request contains two of them.).
160	FileSpecP2S2	Route by file specification in P2/S2.
192	CloseFh	This request is closing a resource that was opened by a request defined OpenFh.
224		(Reserved)

ROUTING REQUESTS

A client's request can be routed from anywhere in a cluster or network, even if the request is undefined in the client process's workstation operating system.

In a standalone workstation, the request block is queued at the exchange of the system service that actually performs the desired function.

Figure 29-17 shows the various paths over which a request can be routed.

If the request is issued on the SRP the Kernel on the processor board of the SRP first determines which of seven SRP routing types is defined for the request. (SRP routing types are described in Chapter 30, "Inter-CPU Communication.") The routing type determines whether the request is local or if it must be routed to a remote processor board. See the first decision (branch) at the top of Figure 29-17 (page 1 of 2).

If the request is issued on a workstation, the first decision the Kernel makes is based on the RtCode field in the request block. In Figure 29-17 (page 1 of 2), this is represented by the decision just beneath the top (SRP routing) decision. (For details, see "Routing Code" and "Routing Requests," earlier in this chapter. Also see Chapter 31, "System Services Management," for details on defining requests for user-written system services.)

FILTERS

A *filter process* is a system service that is interposed between a client and a system service process so that they appear to be communicating directly with each other. The filter does this by substituting its exchange for that of the original system service in the operating system request routing table.

Filters can be used in many ways. A filter, for example, might be used between the file management system and its client process to perform special password validation on all or some requests. Filters are commonly used by the keyboard service to filter keystrokes for various accounting purposes.

The interaction of a filter process with a client and system service process is shown in Figure 29-16.

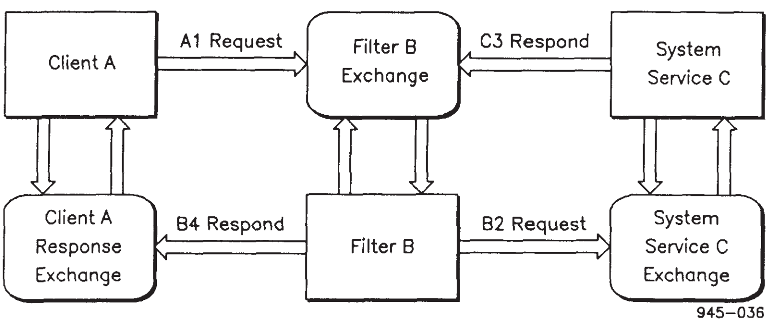


Figure 29-16. Interaction of Filter Process with Client and System Service Process

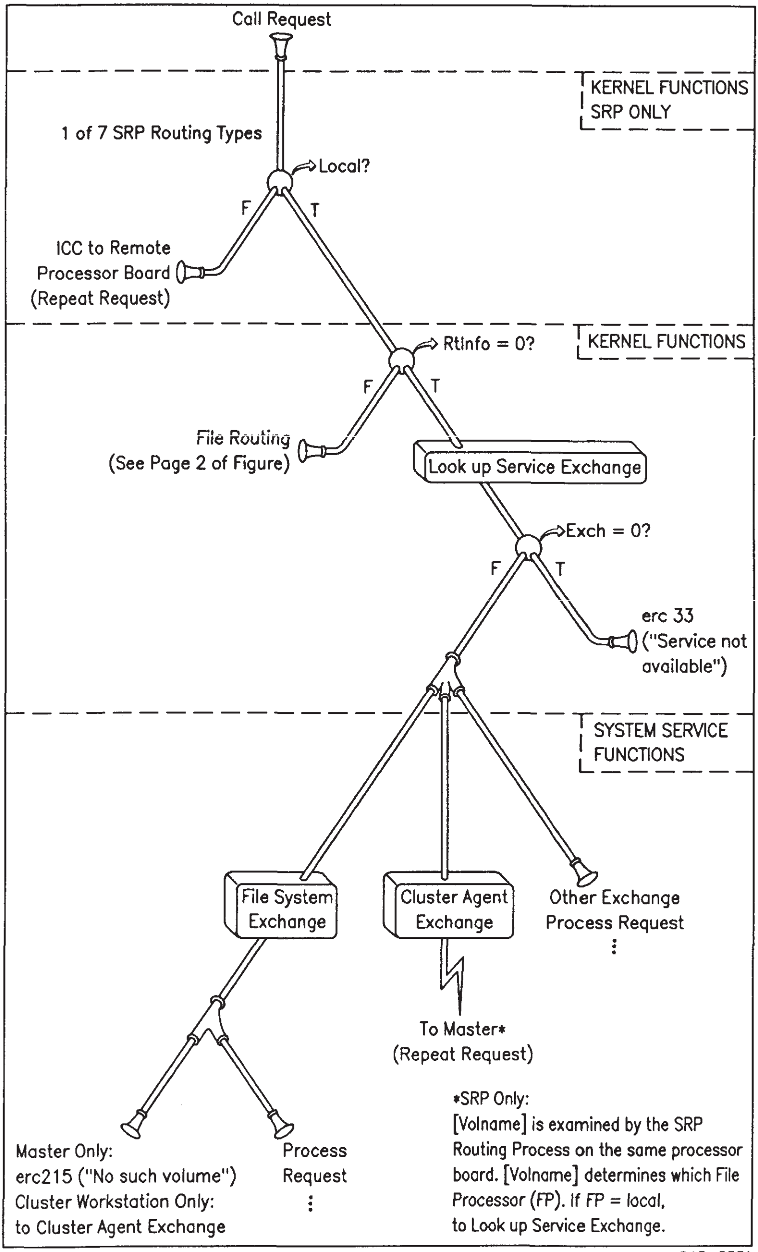


Figure 29-17. Request Routing (page 1 of 2)

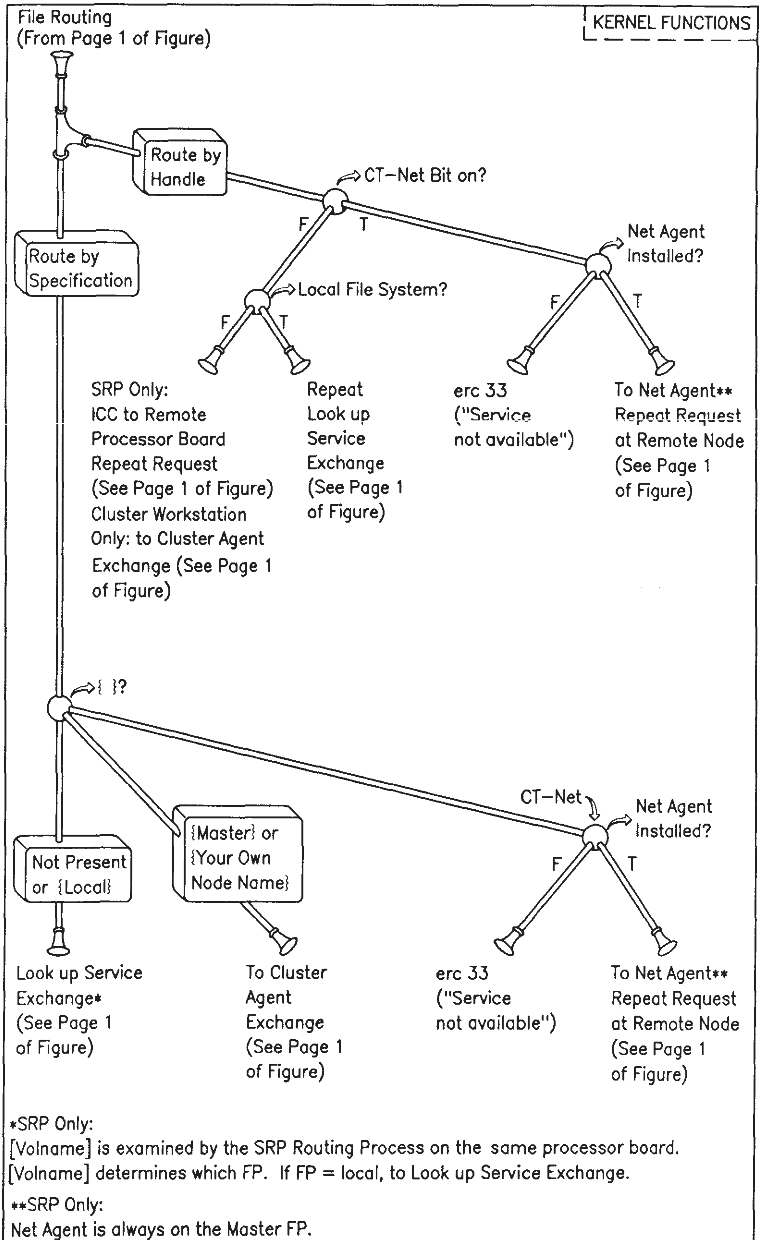


Figure 29-17. Request Routing (page 2 of 2)

Cluster Agents and Net Agents act as filters in directing IPC messages to other destinations for further IPC processing. Configurations involving network routing require that a filter intercept messages branching to local services as well as those that are routed over the network.

(For details on filters, see Chapter 31, "System Services Management.")

INTERPROCESS COMMUNICATION SUMMARY

Figure 29-18 summarizes interprocess communication concepts presented in this chapter.

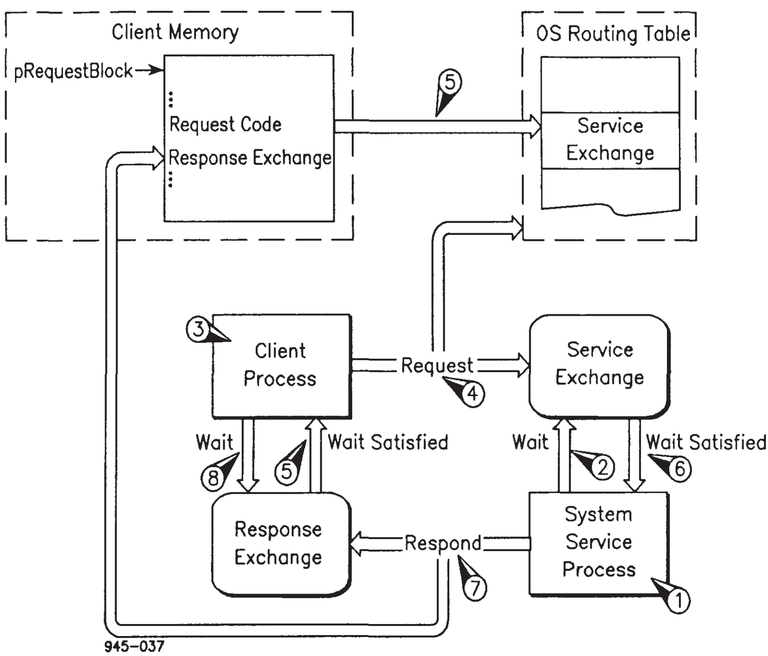


Figure 29-18. Interprocess Communication Summary

1. The system service does a ServeRq to serve the request code(s) at its exchange. (ServeRq is described in Chapter 31, "System Services Management." This causes the Kernel to place the service exchange in the operating system request code routing table at the offset of the request code.

2. The system service process waits at its service exchange. (The Kernel takes the system service process off the run queue and places it in the ready state.)

3. The client process builds a request block in its memory. (Note that the request procedural interface automatically does this step and the next two steps.

4. The client calls Request. (The Kernel looks up the service exchange in the operating system request routing table and queues the request block address on the service exchange message queue. The request can be routed over various paths as described in "Routing Requests," earlier in this chapter.)

5. The client issues a Wait. (The Kernel takes the client process off the run queue and queues the client at its response exchange. The response exchange is the default response exchange if the request procedural interface is used.)

6. The Kernel removes the request block address from the service exchange message queue and passes it to the system service process. The system service process is placed in the ready state.

7. The system service performs its function and calls Respond. The Kernel looks up the client's response exchange in the request block and routes the request back to the client.

8. The client process is given the request block and is placed in the ready state. If it is the highest priority process, it is given control of the processor, and it continues execution.

OPERATIONS

The IPC operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

Request — Requests a system service by sending a request block to the exchange of the system service process.

QueryDefaultResponseExch — Allows a process to determine the identification of its own default response exchange.

Wait — Removes the message (if any) from the queue that was queued first at the specified exchange. Wait causes the calling process to be placed into the waiting state if no messages are queued.

WaitLong — Similar to Wait but is used if the process waiting is expected to be waiting for a long time (more than 30 seconds).

AllocExch — Allocates an exchange.

Respond — Notifies a client process that the requested system service was performed by sending the request block of the client process back to the response exchange specified in the request block.

Check — Removes the first message queued (if any) first at the specified exchange. Check returns the status code 14 ("No message available") if no messages are queued.

Send	Sends the specified message to the specified exchange.
RequestDirect	Sends a request block to an explicitly specified system service exchange. Sending the request block is done independently of the default routing implied by the request code in the request block.
ForwardRequest	Used by filter processes. This operation forwards a request block to another system service for further processing. It does not require a matching Respond.
PSend	Checks whether processes are queued at the specified exchange. The PSend Kernel primitive functions identically to the Send primitive but is used instead of Send for interrupt handling.
DeallocExch	Deallocates an exchange.

30 INTER-CPU COMMUNICATION

The Inter-CPU Communication (ICC) facility provides for communication between CPUs among the different processor boards on the Shared Resource Processor (SRP). ICC is an extension of Interprocess Communication (IPC). (See Chapter 29, "Interprocess Communication." See the CTOS System Administrator's Guide for details on the types of SRP boards and board naming.)

The SRP is compatible with the workstations at the request level. Messages passed between a client and a system service on the same processor board use IPC. The Kernel routes the request to the system service exchange; the system service performs its function and responds to the client's exchange, acknowledging service completion. Figure 29-1 in Chapter 29 shows the request/response model on a workstation. (This same model is used for requests routed locally on a single SRP processor board.)

When a client requests a system service, the Kernel examines its request routing table to determine, for example,

- if the request block is correctly formed
- to which system service the request is to be sent

These actions are taken in the case of ICC or IPC. However, the destination to which the request is sent determines if the request is handled as a normal IPC message or if it is to be routed by means of ICC.

ICC involves interboard routing or the passing of the request and the response message between processor boards. To accomplish this, ICC uses

- processor boards identified by slot numbers
- SRP routing type information in the operating system's request routing table
- an ICC Server Agent on each processor board, which issues requests on behalf of a client on a different processor board
- communication between processors over a high-speed bus
- linear pointers and linear offsets for interboard addressing
- Y-blocks and Z-blocks for storing copies of request blocks
- a request ring buffer and a response ring buffer in a CPU Description Table (CDT) on each processor board
- a doorbell interrupt

SLOT NUMBER

At the hardware level, each processor in a system is identified for ICC communications by a unique 8 bit slot number. Slot numbers range from a high of 77h to a low of 20h.

The slots in the base enclosure are numbered 70 to 77. As viewed from the back of the enclosure, 70 is the leftmost slot, slot 77 the rightmost. The enclosure closest to the base enclosure has slots 60 through 67, the next enclosure in the line has slots 50 through 57, and so on. (See the CTOS System Administrator's Guide for details on slot numbering conventions.)

The slot number is used by certain operating system operations to identify a particular processor and by the hardware to accomplish interboard addressing.

You can use the GetProcInfo and the GetSlotInfo operations to retrieve such hardware information and, thereby, explicitly control ICC routing. You would use these operations if using one of the SRP routing types defined below is not sufficient.

SRP ROUTING TYPES

Table 30-1 describes each of the SRP routing types used to define requests on the SRP. If you are writing a system service for the SRP, you will need to include an SRP routing type in your system service request definition(s). (For details, see Chapter 31, "System Services Management.")

SRP LINEAR ADDRESSING

SRP linear addressing becomes important if you are writing programs that will run on multiple boards. For example, if a client requires a system service located on a processor board other than the one that the client is on, you cannot use equivalent addresses in your program logic.

Table 30-1

SRP Request Routing Types
Page 1 of 2

Field	Description
rLocal*	The request is to be served on the same board. The service exchange is indicated by the service exchange field in the operating system request routing table. The request is to be routed remotely, however, if a file specification for a remote board is included in the request block. In this case, a file system filter calls RequestRemote and routes the request to the board specified by a slot number in the Master Processor global slot number table.
rRemote*	Same as rLocal if the request is served locally. If the request is not served locally, it is searched for in the Master Processor global slot number table. When a system service calls ServeRq during installation, ServeRq updates the Master Processor global slot number table to reflect the system service's slot number. (For details on system service installation, see Chapter 31, "System Services Management.")

*This type is frequently used.

Table 30-1

SRP Request Routing Types
Page 2 of 2

Field	Description
rMasterFP	The request is to be routed to the Master FP.
rHandle*	The request is to be routed by an indexed field in the file handle.
rFileId	The first byte of control information in the request block contains the slot number of the board to which the request is to be routed.
fMasterCp	(Unused)
rLine#	The request is to be routed to the cluster Processor (CP) that handles this line. This routing type is used by the operation MegaFrameDisableCluster. Each CP has two lines. For example, CP000 has lines 1 and 2; CP001 has lines 3 and 4; and so on. (For details, see Chapter 39, "Cluster Management.")

*This type is frequently used.

LINEAR POINTER

The SRP describes structures to be read by the Intel 80x86 family of processors and by multiple boards using a linear pointer. A linear pointer is a 4 byte quantity in which the most significant byte is at the lowest address. A linear pointer (for example the Motorola or IBM format) is absolute, not segment-based.

LINEAR OFFSET

Like a linear pointer, a linear offset has the most significant byte at the lowest address, but it is a 2 byte quantity. The byte ordering is opposite to the Intel 80x86 processor convention, which puts the most significant byte at the highest address. Linear offsets are often said to be byte-swapped.

Linear offsets are used on the SRP to describe structures that must be read by the Intel processors and by multiple boards. A linear offset within a structure is always taken to be the offset relative to the base of the structure.

BLOCKS

Blocks are areas of memory allocated for ICC and for cluster communication.

Y-blocks and Z-blocks are used for holding ICC messages. A Z-block is used if the message can fit into a small number of bytes; otherwise, a Y-block is used. The size and number of these blocks are determined at system initialization. (See the CTOS System Administrator's Guide.)

CPU DESCRIPTION TABLE

Each processor in an SRP contains a CPU Description Table (CDT). The CDT describes the processor to other processors, contains the offsets of the ring buffers used by other processors to send ICC requests and responses, and contains some routing information.

One processor description table, that of the Master Processor, contains rRemote request code slot number tables and tables used to translate line and terminal numbers into particular slot number-port number pairs. (See Table 4-5 in the CTOS/VM Reference Manual for the format of the CDT.)

DOORBELL INTERRUPT

Each processor in an SRP can send an interrupt, called a doorbell interrupt, to any other processor board in the system.

For example, during inter-CPU communication, the Kernel on a processor board sends a doorbell interrupt to alert the ICC Server Agent on the target processor board that a request or response has been registered in a ring buffer and, thus, needs processing.

INTERBOARD ROUTING

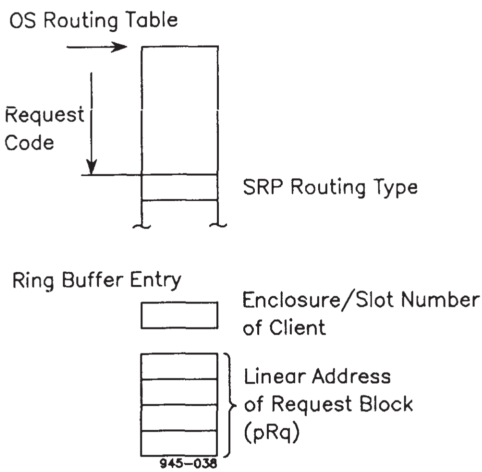
Each processor board provides for sending and receiving messages. In the description of interboard routing that follows, the actions for sending messages and for receiving messages are described separately.

HOW A MESSAGE IS SENT

Sending a message is summarized in Figure 30-1

SENDING REQUESTS

A client process calls Request(pRq). The Kernel calculates SRP routing.



Local routing?

Sends the request to the local service exchange defined by the request code (IPC).

Off-board routing?

Places pRq and slot number into the CDT request ring buffer on the receiving board (ICC).

Sends a doorbell interrupt to the receiving board.

SENDING RESPONSES

Kernel actions in sending a response off board:

Copies the response buffer(s) and a status code to the client's request block on the client's board.

Frees the Z-block holding the request block copy on this board.

Places an entry in the CDT response ring buffer on the client's board.

Sends a doorbell interrupt to the client's board.

Figure 30-1. How a Message Is Sent

Sending Requests

In Figure 30-1, a client calls Request(pRq). The Kernel uses the request code (Rq) as an index into the routing table to determine the SRP routing type. The routing type tells the Kernel where to route the request. (For details, see "SRP Routing Types," earlier in this chapter.)

Local Routing? If request routing indicates that the request is to be served locally and a local server exists, ICC is not used. The request is routed using the normal procedures of IPC. (For details, see Chapter 29, "Interprocess Communication.")

In Figure 30-1, pRq for a request served locally is a logical memory address. (For details on memory addresses, see Chapter 24, "Memory Management.")

Off-board Routing? If request routing indicates that the request is to be served off board, ICC is used to send the request.

To send the request, the Kernel

1. Enters the client's return address into the CDT request ring buffer on the receiving board.

   The <u>ring buffer entry</u> consists of 5 bytes that describe the client's return address: 1 byte defines the client board's enclosure and slot number,- 4 bytes define the client's request block linear address.

2. Sends a doorbell interrupt to the receiving board.

Sending Responses

Figure 30-1 also shows sending responses.

A response to a request originated off-board must be sent back to the client on the requesting board.

The Kernel recognizes a response to be routed off-board by the request block response exchange number.

To return the off-board response, the Kernel takes the following actions:

1. copies the pb/cb response buffers and a status code to the client's request block memory on the client's board

2. frees the Z-block (or Y-block) holding a copy of the client's request block (in the server's processor memory)

3. places the client's return address in the CDT response ring buffer on the client's board

4. sends a doorbell interrupt to the client's board

HOW A MESSAGE IS RECEIVED

Receiving a message is summarized in Figure 30-2.

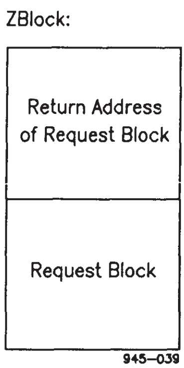
In Figure 30-2, the doorbell interrupt from the sending board alerts the ICC Server Agent on the receiving board that it has received an off-board message in one of its CDT ring buffers.

The ICC Server Agent examines the ring buffer entry to see if it is a request or a response.

The doorbell interrupt from the sending board wakes up the ICC server agent on the receiving board.

RECEIVING REQUESTS

The ICC server agent examines the CDT ring buffer enter.



Request?

Calculates the size of the request.

Copies the request block to an area of memory (Zblock) on this board.

Calls Request (pZblock). (This repeats Sending Requests in Figure 32-1.)

RECEIVING RESPONSES

Respond?

Calls Respond (pRq).

Figure 30-2. How a Message Is Received

Request?

If the ring buffer entry is a request, the ICC Server Agent

1. Calculates the size of the request by examining the size of the client's request block memory. The ICC Server Agent uses the size to reserve a Z-block (or a Y-block) in the ICC board's memory.

2. Copies the request block contents and the client's return address into the Z-block.

3. Calls Request, providing the memory address of the Z-block (or Y-block).

   In Figure 30-2, Request(pZBlock) repeats the sending requests procedure in Figure 30-1.

The Kernel on the receiving board routes the request to the specified service exchange. The message is processed using IPC.

Response?

If the ring buffer entry is a response, the ICC Server Agent calls Respond (pRq) to alert the Kernel on the receiving board to route the response back to the client's local response exchange.

SENDING AND RECEIVING MESSAGES

Figure 30-3 shows the interaction of client A on a Cluster Processor (CP) board and system service B on a File Processor (FP) board.

In Figure 30-3, client A on the CP board requests (Al) a service provided by system service B on the FP board. The Kernel on the CP board places the request block return address in the FP board's CDT request ring buffer and rings the FP's doorbell.

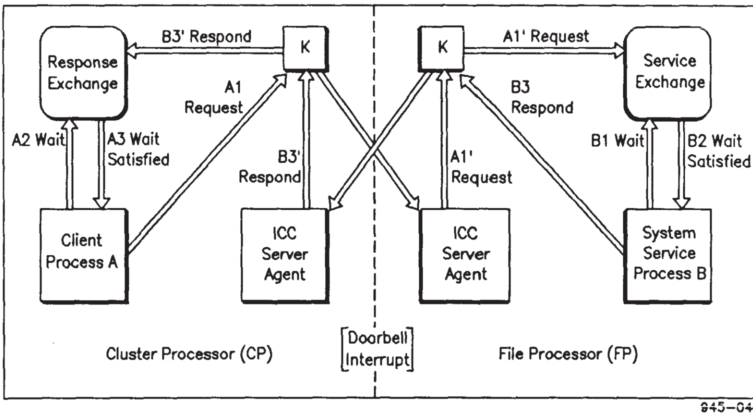


Figure 30-3. Interaction of Client and System Service Using ICC

The ICC Server Agent on the FP board copies the request block contents to a Z-block (or Y-block) in the FP processor and calls Request (A1'). The Kernel on the FP board examines Request (A1'), and sends it to system service B's service exchange, satisfying system service B's Wait (B2). System service B processes the request and responds (B3).

The Kernel on the FP board acts on the Respond (B3) by copying the response back to client A's request block, placing an entry in the CP's CDT response ring buffer, and ringing the CP's doorbell.

The ICC Server Agent on the CP board examines the response ring buffer and calls Respond (B31). The Kernel on the CP board sends Respond (B3') to client A's response exchange, satisfying the client's Wait (A3).

Note that Request and Respond function in two ways in Figure 30-3. One Request and Respond send information to another board; the other Request and Respond are queued at an exchange.

OPERATIONS

The ICC operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

RequestRemote	Requests a system service from a remote processor by sending the request to the ICC Server Agent of that remote processor.
GetProcInfo	Returns the name of the processor on which the caller is running.
GetSlotInfo	Determines the slot numbers of other processors in the SRP system.
RemoteBoot	Causes another dormant processor to be bootstrapped with a specified System Image.

31 SYSTEM SERVICES MANAGEMENT

System services management provides for the management of services to be used by programs requesting them.

A system service is a software program that provides a service to other programs. Examples of services include opening and closing disk files, sending output to a printing device, or accepting input from the keyboard. A service can manage access to a resource, such as a file or a printer.

The program requesting the service is the client. Any program, including another system service, can be a client.

INTERPROCESS COMMUNICATION

A system service does not communicate with a client directly. All correspondence is by means of interprocess communication (IPC). IPC is described in detail in Chapter 29, "Interprocess Communication." In the following description of how a system service functions, some of the IPC concepts are summarized.

A system service receives IPC messages from clients. The message is a special IPC message called a request block.

A request block is a data structure containing the specification and the parameters of the chosen system service. The request block includes fields for the request code and the client's response exchange in addition to other fields that describe the request. (For details, see "Request Block Format" in Chapter 29, "Interprocess Communication.")

The request code is a 16 bit value that uniquely identifies the desired system service. For example, the request code for the OpenFile operation is 4.

A request code is used both to route the request to the exchange of the appropriate system service and to specify which of its several functions the request is for.

The system service waits at its service exchange until it receives a request block from a client. (See Figure 31-1.)

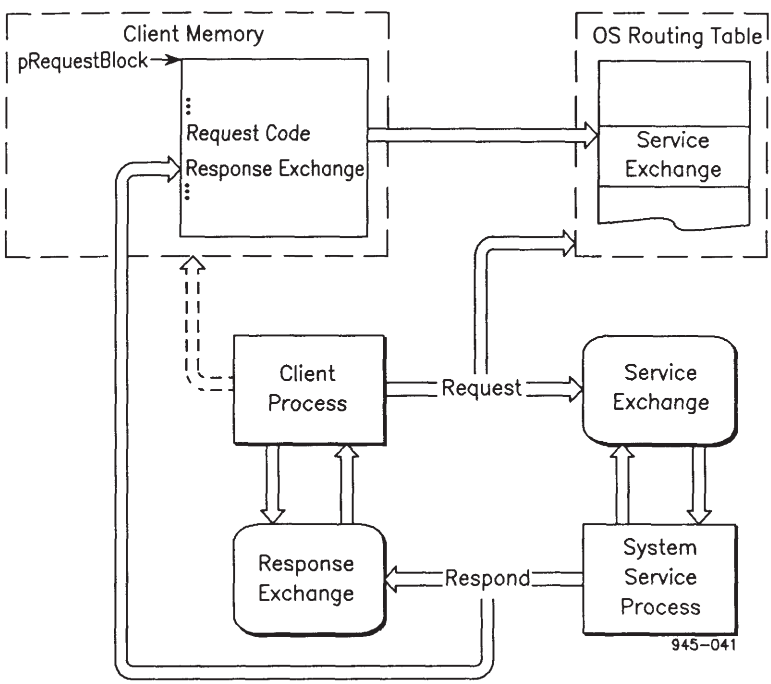


Figure 31-1. Interaction of Client and System Service Processes

The client uses either of two methods to send a request block to the system service's exchange. The client can

- use the request procedural interface, which builds the request block and calls Request
- call Request directly, in which case the client builds its own request block

Request signals the Kernel to examine its request routing table. The Kernel uses the request code as an index into the table to locate the system service's exchange.

Upon receipt of a request block, the system service verifies the information it contains.

If the information is valid, the system service performs its service and answers the client's request by filling in the request block with its response and a 0 status code (ercOK). If the request is invalid, however, it places a status code (in the request block) to indicate an error.

Upon completion of these functions, the system service calls Respond. Respond routes the request block back to the client's exchange as specified in the request block.

Figure 31-2 compares the program model of a system service to that of a client.

In the figure, the system service initializes. Then, it spends its time waiting. Upon receipt of a request block from a client, the system service processes the message and then loops back to its wait.

This is a different model than that of a normal application program. An application spends its time computing, waiting only as required for a service to be performed so it can continue computing.

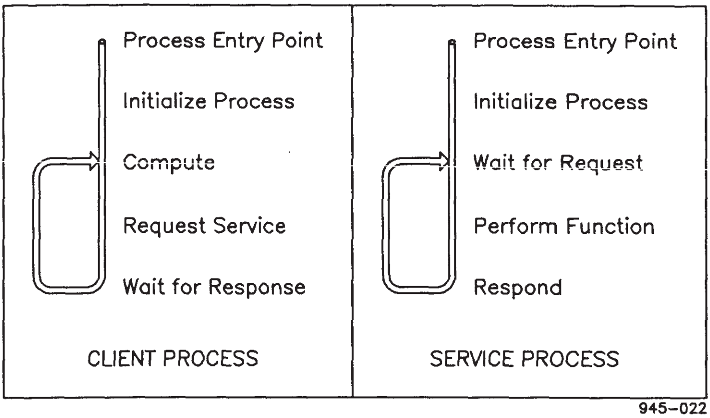


Figure 31-2. Processing Flow of Client and System Service Processes

TYPES OF SYSTEM SERVICES

Some system services can be built into the operating system; others are dynamically installable.

BUILT-IN SYSTEM SERVICES

A built-in system service is one that is linked into the System Image so that it is present when the operating system is bootstrapped. Examples include the file system and the keyboard.

The differences between the various types of operating system are a function of the built-in services each has to offer. A cluster workstation operating system, for example, includes the Cluster Workstation Agent. A cluster workstation with a local file system includes a file management service in addition to the Cluster Agent. (For details, see "Workstation Operating Systems" in Chapter 2, "Overview of Operating System Concepts.")

DYNAMICALLY INSTALLABLE SYSTEM SERVICES

A dynamically installable system service is a service that can be added to the System Image without regenerating the operating system. This type of system service is created as an application program. It becomes become part of the operating system during its initialization.

CT-Net and Mouse Services are examples of installable system services. You also can write your own installable services. (See "Guidelines for Writing a System Service," later in this chapter.)

Dynamically installable services extend operating system functionality. You can install and deinstall them at any time without altering the system in any way. While installed, they function in the same way as built-in system services.

REQUEST ROUTING TABLE

The operating system contains a request routing table for its built-in system services. Request routing tables are used by the Kernel to determine where to send request blocks.

An entry in the request routing table typically includes the following information about a request:

- the request parameters
- the system service exchange of the requested system service

The Kernel uses the request code as an index into the table to locate the system service's exchange. (See Figure 31-3.)

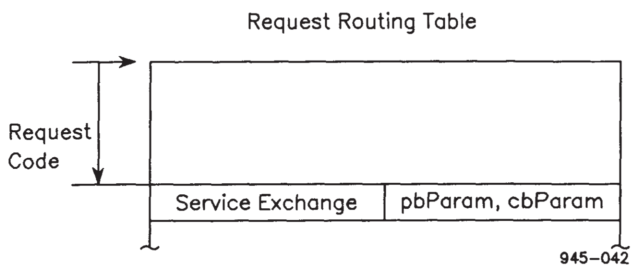


Figure 31-3. Request Routing Table Fields

When a system service is dynamically installed, the request routing table is extended.

You may decide, for example, to install the CT-Mail service at your cluster workstation. The CT-Mail package updates the request routing table to reflect the CT-Mail service exchanges.

WHAT REALLY HAPPENS

In its simplest form, a dynamically installable system service package consists of two software components: the request definitions for the system service and the system service itself.

To allow updating of the request routing table, each of these components is designed in a special way.

REQUESTS

The request definition includes the request code, the request parameters, the system service exchange, and various other fields. (For details, see "Guidelines for Defining System Service Requests," later in this chapter.)

The requests served are defined in a loadable request file. The contents of this loadable request file are merged into already defined loadable requests in a file called Request.sys. The merge occurs during installation of the system service onto the system disk.

When bootstrapped, the operating system reads the Request.sys file, loads it into memory, and adds the new requests to the basic request routing table.

By reading Request.sys, the operating system thus receives acknowledgment that the new requests exist. The operating system sets the service exchange field for each new request according to the request file.

THE SYSTEM SERVICE

After the operating system is bootstrapped, the system service also is loaded into memory. This is usually done by an entry in the SysInit.jcl batch file. (For details, see the CTOS System Administrator's Guide.)

As part of initialization, the system service calls ServeRq for each request it will serve. ServeRq updates the service exchange field (in the request routing table) for each request code to reflect the system service exchange.

If the system service is to be able to deinstall itself later or if it is a filter, it must call QueryRequestInfo, which determines the exchanges to be served, before calling ServeRq.

A filter substitutes its exchange for that of another system service. (See "Guidelines for Writing a System Service," and "Filters," later in this chapter.)

GUIDELINES FOR WRITING A SYSTEM SERVICE

NOTE: These guidelines for writing a system service apply to CTOS/VM operating systems. See the CTOS Operating System Manual, Volume 1, for certain additional guidelines that apply to system services to be run on previous operating system versions.

INITIALIZATION AND CONVERSION TO A SYSTEM SERVICE

A system service begins as an application program when it is first loaded into memory. (See Figure 31-4.)

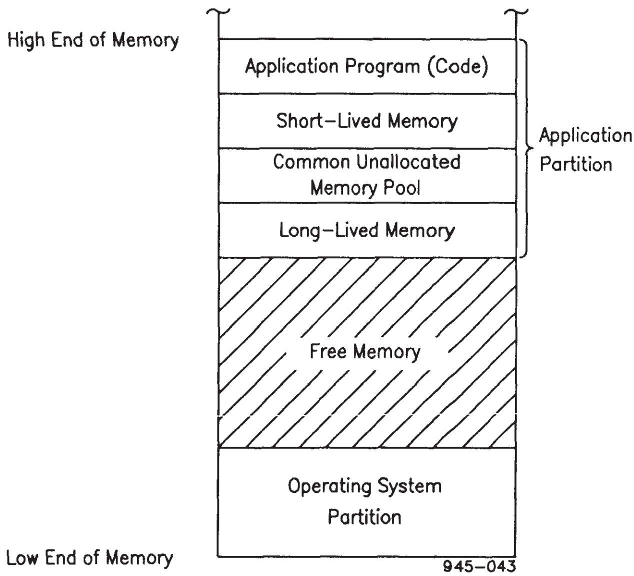


Figure 31-4. Before Conversion to a System Service

The typical operating sequence of a system service initializing itself and converting to a system service is described as follows:

1. Use ChangePriority if desired. A system service priority normally should be in the range of 10 to 64.

2. Use all required initialization operations, such as AllocExch, AllocMemorySL, and CreateProcess, to get required resources before converting to a system service.

3. Use the QueryRequestInfo operation to find out the current exchanges for all of the requests to be served. This is required if the system service is to be able to deinstall itself later or if the system service is going to filter messages destined to other system services. (For details, see "Deinstallation of a System Service" and "Filters," later in this chapter.)

4. Optionally use the SetMsgRet operation to provide the exit run file with an informative message indicating success or failure of the installation.

5. Use the ConvertToSys operation to become part of the operating system.

   (Figure 31-5 compares system memory before and after the ConvertToSys operation.)

6. Use the ServeRq operation for each request code to be served. In addition, the ServeRq operation must be used for each system request to be filtered. (See "System Requests," later in this chapter.) Note that it is best not to use the default response exchange or else the server will be unable to use the request procedural interface.

7. Use the Exit, ErrorExit, or Chain operation to reload the exit run file into memory. Note that since the program is now a part of the operating system, these calls will return to the new system service (for normal application programs, these calls never return).

8. Use the SetPartitionName operation to set an identifiable (up to 12 character) name for the partition. SysServiceXX is the default name, where XX is the user number.

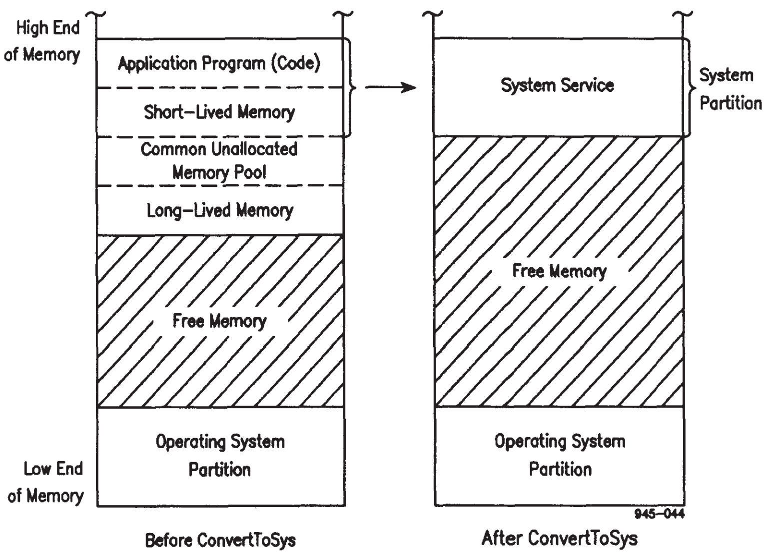


Figure 31-5. Conversion to a System Service

SYSTEM SERVICE MAIN PROGRAM

The program model of a system service is shown in Figure 31-6.

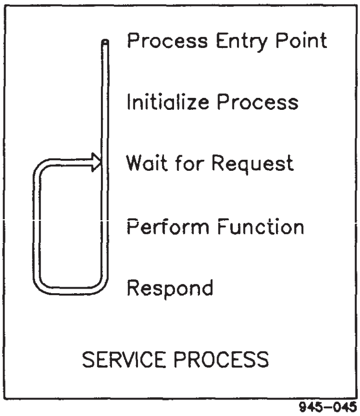


Figure 31-6. System Service Program Model

After initialization and conversion to a system program, the system service enters its main program. In the main program, it calls Wait and waits at its exchange. This gets the system service into its normal state: waiting to do work.

The loop in Figure 31-6 signifies the program instructions the system service executes when it performs a service for a client. After executing these instructions, the system service calls Respond and loops back to its waiting state.

RESTRICTIONS AND REQUIREMENTS OF OPERATION

As part of the operating system, a system service is a special type of program. It must adhere to the following specific rules to function correctly.

- It must not allocate or deallocate memory.
- It cannot write to the video or read from the keyboard, or call the ErrorExitString operation after calling ConvertToSys successfully.

All system services must perform some common functions in the system, by serving termination, abort, and swapping requests. (For details, see "System Requests," later in this chapter.)

GUIDELINES FOR DEFINING SYSTEM SERVICE REQUESTS

The information needed for defining a request is contained in RequestTemplate.txt, a special text file. This file is supplied as part of Standard Software. Note that you can also use a different file, Request.0.asm, which is part of Standard Software for earlier operating system versions.

Either file works. RequestTemplate.txt, however, is friendlier:

- It is a text file you can edit.
- You use the **Make Request Set** utility program, which is much faster than the assembler (used with Request.0.asm) and provides more comprehensive error checking. (For details on **Make Request Set**, see the CTOS System Administrator's Guide.)

Table 31-1 describes the fields in this text file.

Table 31-1

REQUESTTEMPLATE.TXT FIELDS
Page 1 of 2

Field	Description
:RequestCode:	Uniquely identifies the request. (For details, see "Request Codes" in Chapter 29, "Interprocess Communication.")
:RequestName:	Identifies the request to a user. This entry is optional but strongly recommended.
:Version:	Indicates whether a request has been updated (default = 0). Requests are generally not updated.
:LclSvcCode:	Is used by the operating system for a special case but is not generally used in writing system services (default = 0).
:ServiceExch:	Indicates the exchange to which the request is routed. This field is changed when a system service calls the ServeRq operation, which provides the exchange to which the request is routed.
:sCntInfo:	Indicates the number of bytes of control information (default = 6).
:nReqPbCb:	Indicates the number of request pb/cb pairs.

Table 31-1

REQUESTTEMPLATE.TXT FIELDS
Page 2 of 2

Field	Description
:nRespPbCb:	Indicates the number of response pb/cb pairs.
:Params:	Defines the request procedural interface. This field is used by the operating system for validation of request blocks.
:NetRouting:	Describes file system request routing. (See "Routing by File Handle" and "Routing by File Specification" in Chapter 29, "Interprocess Communication.")
:SrpRouting:	Describes how requests are routed among boards on the Shared Resource Processor (SRP). (See Chapter 30, "Inter-CPU Communication.")
:WsAbortRq:	Is the request code for the abort system request.*
:TerminationRq:	Is the request code for the termination system request.*
:SwappingRq:	Is the request code for the swapping system request.*

*This is a special system request. For details, see "System Requests," later in this chapter.

System services for the SRP must serve either local or global requests. Local requests are served on the same processor board as the system service. Global requests are served on any SRP processor board.

SRP system services must serve requests of the same SRP routing type. SRP routing types are described in Chapter 30, "Inter-CPU Communication."

GUIDELINES FOR CREATING A LOADABLE REQUEST FILE

To create a loadable request file, use either the RequestTemplate.txt or the Request.0.asm template file. (See "Guidelines for Defining System Service Requests," earlier in this chapter.) Using these templates to create a loadable request file is described below:

1. Copy the template to a file identifying the system service.

   - If you use RequestTemplate.txt, copy the template to a file, such as RequestServer.txt.

   - If you use Request.0.asm, copy the template to a file, such as Request.X.asm, where X identifies a group of requests for the system service.

2. Use the Editor to edit your file according to the instructions provided.

3. To build the request file,

   - If you used Request.0.asm, assemble and link your file to create the binary file, Request.X.sys.

   - If you used RequestServer.txt, run your text file through the **Make Request Set** utility, which reads your text file, checks for errors, and creates a binary file, RequestServer.bin.

4. Use the **Install New Request** utility to merge your request(s) with the system file, Request.sys. (For details on **Make Request Set** and **Install New Request**, see the CTOS System Administrator's Guide.)

5. Bootstrap the operating system. Bootstrapping results in the operating system reading the single system request file, Request.sys, and adding the loadable requests to the request routing table.

Table 31-2 compares and summarizes the templates.

Table 31-2

CREATING A LOADABLE REQUEST FILE

RequestTemplate.txt	Request.0.asm
Copy template to RequestServer.txt	Copy template to Request.X.asm
Edit the text file	Edit the macros
Use **Make Request Set**	Assemble and link to build Request.X.sys
Use **Install New Request** to merge your request(s) into a single system file, Request.sys	Use **Install New Request** to merge your request(s) into a single system file, Request.sys
Bootstrap system	Bootstrap system

SYSTEM REQUESTS

System requests are issued by the operating system to system services. These requests notify system services of clients that are terminating or being swapped to a disk file.

The system requests are

- termination
- abort
- swapping

(See the system request fields in Table 31-2. Also see RequestTemplate.txt file included with Standard Software for examples of how to define these requests.)

TERMINATION AND ABORT REQUESTS

Termination and abort requests function similarly in that they notify system services that clients have terminated. Upon notification, system services can release resources, such as open files and locked ISAM records, allocated to the terminating clients.

The operating system issues termination requests whenever a client terminates for the following reasons:

- The client called Chain, Exit, or ErrorExit.
- A user pressed **Action-Finish**.
- A partition managing program called the TerminatePartitionTasks operation to terminate the client.

In addition to termination requests, the operating system issues abort requests at a master

- when the master detects that it cannot communicate with a cluster workstation
- when a partition is vacated with the VacatePartition operation or through lack of an exit run file

These requests are issued to

- ensure that no requests will be returned to the program after it has been terminated and replaced in memory by another program
- inform servers that resources allocated to the program should be freed

System services must respond to outstanding requests before responding to termination or abort requests. Although a terminating client does not need the response, certain operating system structures the client was using, such as Z-blocks for interboard routing on the SRP, may be made unavailable for future use.

TERMINATION REQUEST TO THE FILE SYSTEM

The following is an example of how the file system service uses the termination request. The example also indicates the consequences of a file system not calling ServeRq to serve a termination request.

When a user initiates the **Copy** command in the Executive, the Executive makes requests to the file manager to read and write files to disk.

During execution of these requests, the user presses the key combination, **Action-Finish**. This terminates the Copy program and results in the operating system issuing a termination request to the file system process.

In response to the termination request, the file system process terminates any outstanding read or write requests initiated by the Copy program.

If the file system did not serve the termination request, the Copy program's exit run file, the Executive, would be reloaded into memory. An outstanding Write request responded to by the file system process at this time would result in the response data being written to the Executive's memory rather than to the Copy program's memory.

SWAPPING REQUESTS

Swapping requests are issued to system services whenever the operating system is going to suspend a program and swap it to disk. Swapping requests ensure that no responses are made to clients in a program that is not resident in memory.

When a system service receives a swapping request, it is required to respond to all outstanding requests with the same client user number and then to respond to the swapping request.

The system service can use either of the following strategies:

- It can hold the swapping request until all outstanding requests for the client are completed and then respond normally to the swapping request.
- It can respond to all outstanding requests for the client with status code 37 ("Service not completed"). The operating system intercepts this special response status code and the program is swapped to disk. Later, when the program is swapped back into memory, the operating system reissues the original outstanding requests to the system service.

It is transparent to the program that it is being swapped out of memory or that any of its requests are being handled other than in the usual manner.

[See the CTOS/VM Reference Manual, Chapter 3, "Operations," for the specific formats of (and additional information regarding) each of the system requests.]

FILTERS

A filter process is a system service that is interposed between a client and a system service process that operate as though they were communicating directly with each other. The filter does this by substituting its exchange for that of the original system service in the operating system request routing table.

If your system service acts as a filter, it can intercept requests intended for another system service, and either service them itself, or re-issue them to the original service after performing filtering or some other function.

TYPES OF FILTERS

Filters are of three types: replacement, one-way pass-through, and two-way pass-through.

REPLACEMENT

A replacement filter intercepts requests (using the Wait or Check Kernel primitive), performs a service based on the intercepted request, and then responds to the request. In this case the filter replaces the original system service.

ONE-WAY PASS-THROUGH

A one-way pass-through filter intercepts requests and then sends the request on to the system service exchange. It uses the ForwardRequest Kernel primitive to forward a request block to a system service for further processing.

Figure 31-7 shows how this type of filter is used.

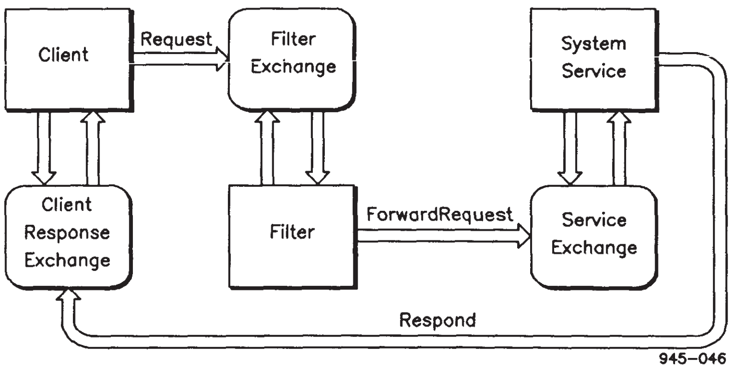


Figure 31-7. One-Way Pass-Through Filter

The sequence of events shown in Figure 31-7 is as follows:

1. The client issues a Request.

2. The filter proceeds from its Wait.

3. The filter issues ForwardRequest (or Send) to the original system service's exchange.

4. The system service proceeds from its Wait.

5. The system service calls Respond.

6. The client proceeds from its Wait.

To be compatible in protected mode, one-way pass-through filters must use ForwardRequest, instead of Send.

TWO-WAY PASS-THROUGH

A two-way pass-through filter intercepts requests and reissues them to the original system service exchange using the RequestDirect Kernel primitive. It also intercepts the Respond and responds back to the client.

Figure 31-8 shows how this type of filter is used.

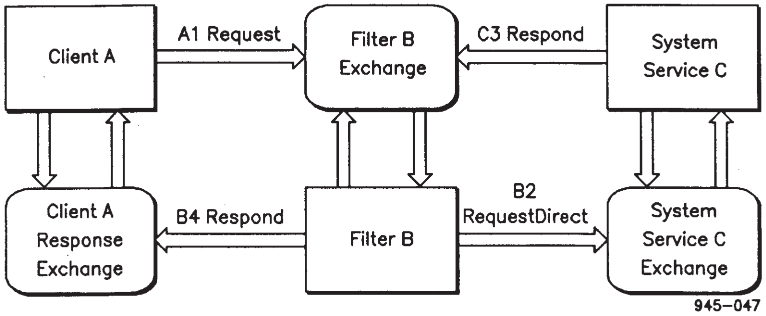


Figure 31-8. Two-Way Pass-Through Filter

The following sequence of events is shown in Figure 31-8:

1. The client issues a Request.

2. The filter proceeds from its Wait.

3. The filter changes the exchResp field to its own exchange and then issues RequestDirect to the original system service's exhange, then Wait.

4. The system service proceeds from its Wait.

5. The system service issues a Respond.

6. The filter proceeds from its Wait.

7. The filter changes the exchResp field back to the client's exchange and issues a Respond.

8. The client proceeds from its Wait.

SYSTEM REQUESTS FOR FILTERS

A filter that uses only the replacement method should have its own system requests for termination, abort, and swapping. (For details, see "System Requests," earlier in this chapter.) In this case the filter is the same as a normal system service.

A filter process that uses one of the pass-through methods of filtering must filter the system requests of the original system service(s). If the filter uses the two-way pass-through method for any requests, it also must use that method for the system requests.

USE OF FILTERS

Filters can be used in many ways. A filter, for example, might be used between the file management system and its client process to perform special password validation on all or some requests. Filters are commonly used by the keyboard service to filter keystrokes for various accounting purposes.

Cluster Agents and CT-Net Agents act as filters in directing IPC messages to other destinations for further IPC processing. (For details, see Chapter 29, "Interprocess Communication.")

EXAMPLE OF A FILTER NOT SERVING A SWAPPING REQUEST

The following example describes the consequences of a keyboard filter not performing a ServRq on keyboard swapping requests.

The Context Manager maintains an outstanding ReadActionKbd request to the keyboard manager to receive **Action** key combinations. The key combination, **Action-Next**, for example, alerts the Context Manager to switch to a different context (user number).

The two-way pass-through filter has been installed to intercept the ReadKbd requests.

Under the Context Manager, a user is running an Executive program as the current context. The Executive is issuing a series of ReadKbd requests while the user is typing characters onto the command line. The user types the characters **C**, **O**, and **P**, followed by the key combination, **Action-Next**.

The Context Manager, whose priority is higher than the Executive, receives the **Action-Next** key combination before the filter receives the P. In response, the Context Manager initiates a swap to bring in the chosen context.

A swapping request is issued by the operating system. The request bypasses the filter and goes directly to the keyboard process, which responds.

The filter, which was not notified of the context switch, holds onto the ReadKbd request. As a result, the swap file fails with status code 813 ("Cannot swap out this partition").

DEINSTALLATION OF A SYSTEM SERVICE

A system service may deinstall itself. To do this, you must write a utility program that runs at the same workstation as the system service and that issues a deinstallation request to the system service.

The deinstallation request should have the user number of the system service as one of the response parameters. Deinstallation should follow these steps:

1. The utility program issues a deinstallation request to the system service.

2. The system service performs a ServeRq on all of its requests to restore them to their original values.

3. The system service checks all of its exchanges and internal queues and responds to all requests it may still have, except the deinstallation request.

4. The system service calls SetPartitionLock(0).

5. The system service calls GetUserNumber to find its user number.

6. The system service copies its user number to the memory address of the deinstallation request response field and then responds to the request with 0 (ercOK) in the ercRet field.

7. The system service calls Wait and waits for the removal of the partition at one of its exchanges.

8. The utility program receives the response to its request. If the ercRet field is 0 (ercOK), it calls VacatePartition followed by RemovePartition, using the user number returned by the system service.

OPERATIONS

The system services management operations described below are categorized as basic or special. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

BASIC REQUESTS USED BY ALL SYSTEM SERVICES

QueryRequestInfo
: Determines the exchange to which a request and its local service code are routed.

ConvertToSys
: Converts all processes, short-lived memory, and exchanges in an application partition to system service processes, system memory, and system exchanges, respectively, in a system partition.

ServeRq
: Is used by a dynamically installed system service process to declare its readiness to serve the specified request code.

SetPartitionName
: Changes the name of the caller's partition.

SystemCommonConnnect
: Installs the memory address of a system-common procedure in the System Common Address Table at the specified reserved location.

GetNodeName
: Obtains the node name of the local node where this request is issued.

SYSTEM REQUESTS

System requests include termination, abort, and swapping requests (discussed earlier in this chapter).

32 PROGRAM AND PARTITION MANAGEMENT

Program and partition management provides you with information on how the operating system uses its memory resource.

The program management operations are used by a program to self-load into memory, to self-exit from memory, and to handle error conditions. These same operations are described in Chapter 4, "Program Management." This chapter, however, includes additional program management operations used by partition managing programs, such as the Context Manager, to facilitate program management within partitions under their control.

This chapter also introduces the partition management operations. These operations are typically used by partition managing programs to create and to remove partitions, for example.

AN EXECUTABLE PROGRAM

An executable program can consist of code, data, and one or more processes in a memory partition.

A program is loaded into a partition in memory from a disk-resident file or run file. Run files are created by compiling and/or assembling source language modules into object modules and linking the object modules together into code and data segments. (See Figure 32-1.)

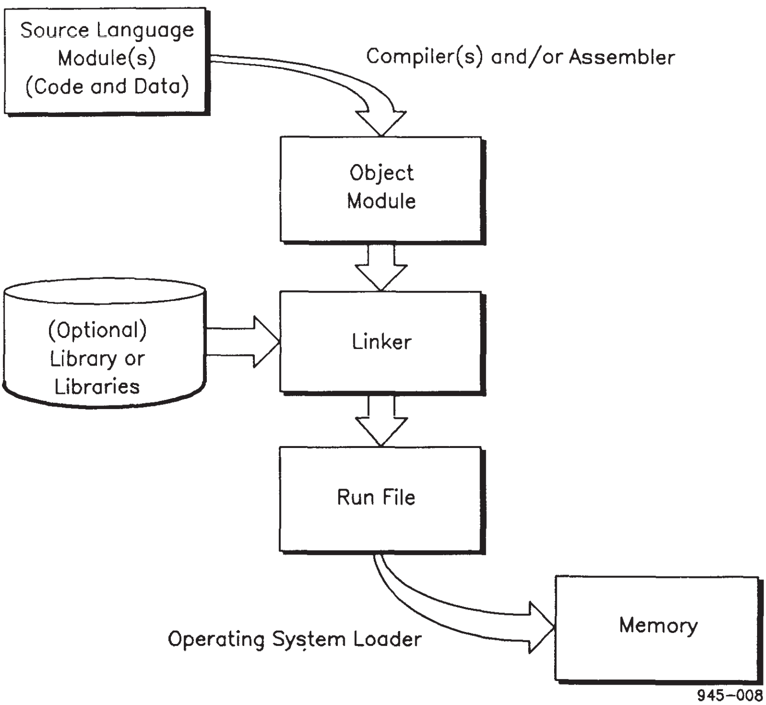


Figure 32-1. From Source Language Modules to Program in Memory

SEGMENTS

A code segment contains only processor instructions (code) and is never modified once it is loaded into memory. Several processes can execute instructions from the same code segment. (See "Code, Static Data, and Dynamic Data Segments" in Chapter 24, "Memory Management.")

A static data segment contains initial values of program data structures and is constantly being changed once in memory. Every invocation of a program gets a new static data segment.

LINKER

The Linker reads the object module(s) and combines them according to their segment names, class names, and directives from the user.

Segments can be combined based on a series of different segmentation models. Most operating system languages use the medium model, although the operating system also supports the small and large model. (For details, see the CTOS Programmer's Guide.)

A run file created by linking object modules produced by the Pascal compiler, for example, consists of one code segment for each object module included in the link and a single static data segment. The single static data segment, or DGroup, combines the static data and stack requirements of all the object modules.

A run file of this form is considered standard; assembly language programmers are urged to adopt this standard unless other considerations are overriding. The COBOL compiler and BASIC interpreter do not produce object modules. (For details, see the Linker/Librarian Manual.)

CODE SHARING

The program's code can be shared by another instance of the same program in a different partition (protected mode operating systems only). For example, if you were running the Executive in two different partitions concurrently under the Context Manager, the code from the Executive run file would be shared.

PROGRAM SIZING

You can size a program at link time (protected mode operating systems only). Sizing a program means controlling both

- the maximum amount of memory it can allocate
- the minimum amount of memory that the operating system will allocate for it before attempting to run the program

(For details, see the Linker/Librarian Manual.)

MULTIPROGRAMMING AND PARTITION MANAGEMENT

One of the features of the operating system is that it supports multiprogramming or the simultaneous execution of several programs in memory, each in its own partition. Partition management accomplishes this by coordinating programs. Partition managing programs, such as the Context Manager, provide this feature.

TYPES OF PARTITIONS

System memory consists of two types of partitions:

- System partitions: A system partition can contain the operating system or a dynamically installed system service. A system service manages resources that can be accessed by application programs or other system services.
- Application partitions: An application partition can contain an application program.

FIXED AND VARIABLE PARTITIONS

A partition can be a fixed partition or a variable partition. A fixed partition always uses a fixed amount of memory.

A variable partition (protected mode operating systems only) grows with a program's needs. It can use up to the maximum amount of memory you specified when you sized your program. (See "Program Sizing," earlier in this chapter.)

In addition, a variable partition permits code to be shared by another program of the same type in another variable partition. (See "Code Sharing," earlier in this chapter.)

USER NUMBER

A user number (historically the same as a partition handle) is a 16 bit integer that uniquely identifies the program and/or the resources associated with a partition. Resources include file handles, short-lived memory, long-lived memory, and exchanges. User number is not associated with a partition's particular size or physical location in memory. This is because partitions are not static memory cells into which programs are loaded: a partition is created at the time a program is loaded into memory and is removed when the program is terminated. (Also see "Partition Swapping," later in this chapter.)

When a partition managing program, such as the Context Manager, calls the CreatePartition operation to create a partition, the user number for the partition is returned. The partition managing program can use the user number to refer to the partition in subsequent operations such as GetPartitionStatus, LoadPrimaryTask, and RemovePartition.

A previously assigned user number can be obtained by supplying the partition name to the GetPartitionHandle operation. The user number is subsequently used in calls such as GetPartitionStatus or GetPartitionExchange.

A partition is removed using the RemovePartition operation. The specified user number is deallocated by the operating system and becomes available to be reissued in response to a CreatePartition call.

A program can obtain the user number of its own partition by calling the GetUserNumber operation.

OBTAINING PARTITION STATUS

A program can obtain status information about a specified application partition and the program executing in it (such as the user number and whether the program is sized) by using the GetPartitionStatus operation. (For details, see the GetPartitionStatus operation in the CTOS/VM Reference Manual, Chapter 3, "Operations.")

COMMUNICATION BETWEEN APPLICATION PARTITIONS

The Intercontext Message Server (ICMS) provides for communication between application partitions managed by partition managing programs. (For details, see Chapter 29, "Interprocess Communication." Also see the Context Manager/VM Manual.)

NOTE: This manual generally describes a logical model of the operating system rather than a particular implementation (such as real mode or protected mode). For implementation details, see the Release Notice for your version of the operating system.

MEMORY ORGANIZATION OF AN APPLICATION PARTITION

The memory organization of an application partition is shown in Figure 32-2. An application partition can contain

- application program code
- short-lived memory
- common pool of unallocated memory
- long-lived memory
- Local Descriptor Table (LDT) (protected mode only)

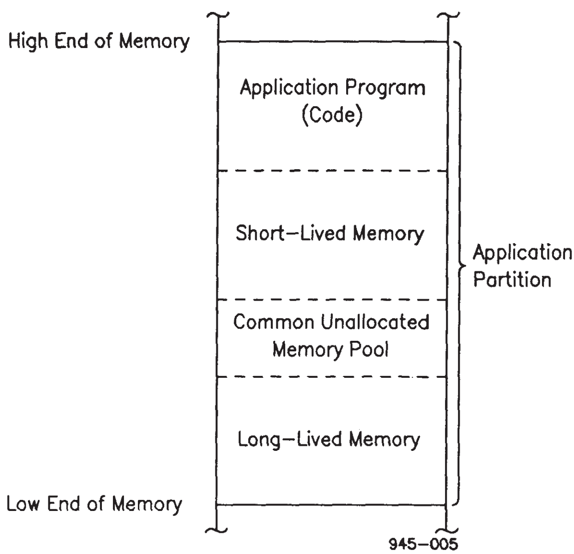


Figure 32-2. Memory Organization of an Application Partition

A program can allocate and deallocate the memory of its own partition. Long-lived memory is allocated from the low-address end and short-lived memory from the high-address end of the partition. A program cannot allocate or deallocate memory in other partitions. System data structures describing the partition and its current program can be located in separate memory.

PROGRAM LOADING INTO MEMORY

When a program is loaded into memory, the run file is read into the short-lived memory of the application partition. For real mode programs, any logical memory addresses existing in either the code or data segments (intersegment references) are adjusted to reflect the memory address at which the program is loaded. For protected mode programs, the Loader adjusts the base addresses in each LDT descriptor.

The Virtual Code Management facility allows you to run a program that is larger than the available memory in an application partition. If the Virtual Code Management facility is in use, all the static data segments and the resident code segment are loaded into memory. The nonresident code segments are loaded into memory only as needed. (For details, see Chapter 34, "Virtual Code Management.")

The program is loaded by the Chain, Exit, ErrorExit, LoadPrimaryTask, or LoadInterActiveTask operation.

LoadPrimaryTask and LoadInteractiveTask must be followed by a call to SwapInContext or AssignKbdOwner if a program is to be loaded into memory by a partition managing program.

Additional run files can also be loaded into the same partition in memory by the program management LoadTask operation, but this is not as common an occurrence. (For details, see "Application Partition with More Than One Run File," later in this chapter.)

EXIT RUN FILE

When the currently executing program exits, the exit run file is the next program that is loaded into the partition. Exit run files are user-specified. Each application partition has its own. For example, the Executive sets itself as the exit run file: The user starts the application from the Executive, and when the application is done, the Executive is reloaded.

A program can specify an exit run file for its partition by calling the SetExitRunFile operation. QueryExitRunFile can be called to determine the exit run file.

If no exit run file is specified in a partition, the partition becomes vacant.

TERMINATING PROGRAMS

The application program terminates itself by using the Chain, Exit, or ErrorExit operation.

In addition, a partition managing program can use the TerminatePartitionTasks and VacatePartition operations to terminate an application program in another partition. Both operations function in the same way in terminating the program in the partition.

They differ in that TerrainatePartitionTasks also loads and activates the partition's exit run file, if one is specified. If no exit run file is specified, TerminatePartitionTasks and VacatePartition are equivalent.

When a program terminates, the operating system issues termination requests. Termination requests (system requests) are messages that notify system services of a program's termination. Upon receipt of a termination request, system services release resources, such as open files, that may be allocated to the terminating program. (For details, see Chapter 31, "System Services Management.")

REMOVING PARTITIONS

An existing vacant application partition can be removed by using the RemovePartition operation.

An application partition is vacant when one of the following is true:

- It is first created.
- The current application program exits with no exit run file specified.
- The VacatePartition operation is performed.

DEALLOCATION OF SYSTEM RESOURCES

Only the resources allocated to an exiting program are deallocated when that program terminates.

The resources that are deallocated include

- Short-lived memory. (See Chapter 24, "Memory Management.")
- Exchanges. (See Chapter 29, "Interprocess Communication.")
- Files opened by the OpenFile operation (except long-lived files). (See Chapter 11, "File Management.")
- Timer Request Blocks allocated by the OpenRTCClock operation. (See Chapter 33, "Timer Management.")
- Communications channels allocated by the InitCommLine operation. (See Chapter 15, "Serial Port Management.")
- Global Descriptor Table selectors (SGs) (protected mode) (See the iAPX 286 Programmer's Reference Manual, the 80286 Architecture, and the 80386 Programmer's Reference Manual.)

PARTITION ORGANIZATION IN MEMORY

AT SYSTEM INITIALIZATION

When a system is initialized, the operating system is loaded into the low address and high address ends of memory in system partitions. Dynamically installed system services are loaded into system partitions located at the high address end of memory. All remaining memory is defined initially as free memory. Figure 32-3 shows how memory is organized at system initialization for protected mode and real mode operating systems.

Programs executing in system partitions are system service programs. Such programs (other than the operating system) start as ordinary application programs and then use the ConvertToSys operation to change the status of their partition from application partition to system partition. A program can call ConvertToSys as long as memory consists of a single application partition; otherwise, status code 810 ("Invalid request") or status code 206 ("Invalid user number") is returned. (System services are described in Chapter 31, "System Services Management.")

SINGLE APPLICATION PARTITION IN MEMORY

An application partition is a partition in memory in which an application program can be executed. Application programs can use the keyboard and video display, and can allocate memory dynamically. If the program is an interactive command interpreter, such as the Executive, you can use the program to load other programs, such as the Editor, Document Designer, or Multiplan, into the partition.

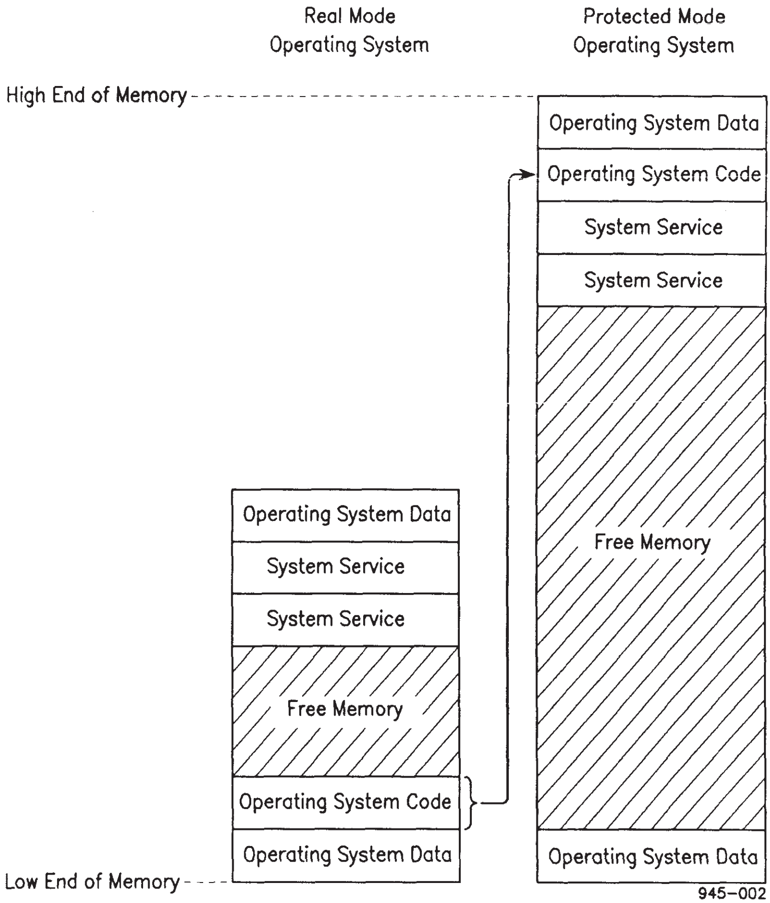


Figure 32-3. Memory Organization at System Initialization

Figure 32-4 shows typical memory organization when a single application partition containing a program is in memory.

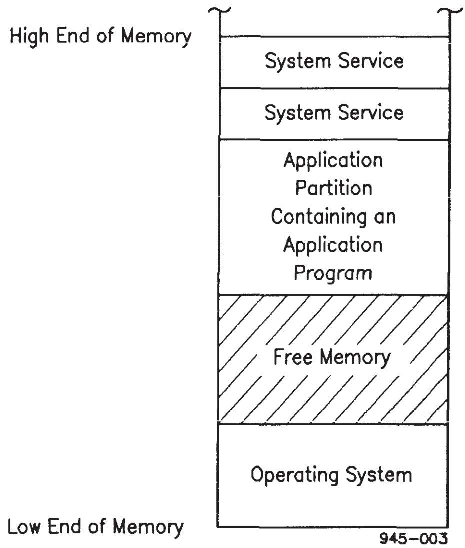


Figure 32-4. Memory Organization Showing a Single Application Partition Containing a Program

MORE THAN ONE APPLICATION PARTITION IN MEMORY

A partition managing program is designed to create and to manage other partitions, more than one of which can be in memory at once. The Context Manager is such a program and is used in the following discussion of multiple partitions.

You must install all system services before installing the Context Manager. For example, you cannot use the Executive commands to install system services from an Executive program in a partition under context management.

Figure 32-5 Part A shows what memory looks like when the Context Manager is first loaded into memory. The Context Manager is at the high address end.

When the user selects an application to start, the Context Manager dynamically creates a fixed or a variable application partition using the CreatePartition operation. The new partition is created just beneath the Context Manager, which remains in memory at the high address end. The Context Manager then loads the selected application program into that partition using the LoadPrimaryTask operation. The remaining unused memory is free memory. (See Figure 32-5 Part B.)

Each additional program started from the Context Manager is loaded just under the last until memory is full. Figure 3 2-5 Part C shows what memory looks like with the addition of a second program in memory.

When a user finishes a program, the partition that it was in becomes free memory as shown in Figure 32-5 Part D.

Partition Swapping

When the user chooses to start a program from the Context Manager and there is not enough free memory available to create a partition into which to load the program, the operating system selects which partition(s) to swap out to a file on disk or to extended memory (above the first megabyte). To do this, the operating system uses an algorithm, which takes into consideration

- whether the program is capable of swapping
- whether the program is currently using the video display (real screen) and keyboard

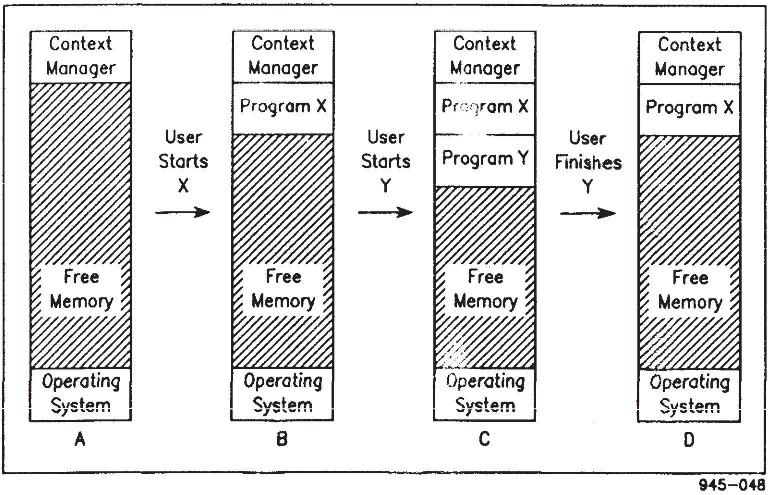


Figure 32-5. Memory Organization with More Than One Application Partition in Memory

When program(s) are swapped out of memory, the memory where the program(s) was located becomes free memory. This free memory is available

- to the Context Manager to create a new partition into which to load a new program
- to the operating system to swap a program back into memory from disk or extended memory

Figure 32-6 shows the following example sequence of what memory looks like when swapping occurs:

1. Figure 32-6 Part A shows Program W, Program X, and Program Y in memory partitions.

2. The operating system selects to swap Program X out to a disk file. The memory area where Program X's partition was located becomes free memory, as shown in Figure 32-6 Part B.

3. Figure 32-6 Part C shows memory after Program Z is swapped in.

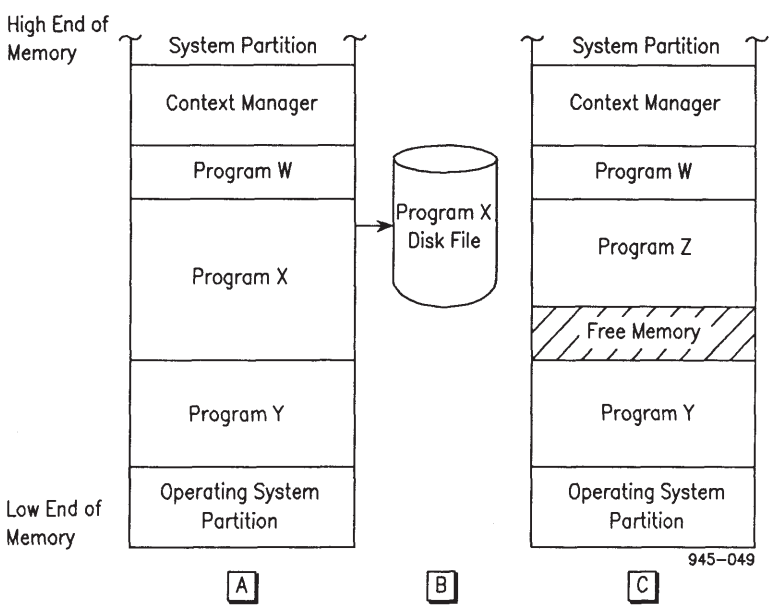


Figure 32-6. Swapping

Note that Program Z's partition is occupying a memory location that was previously occupied by Program X's partition. Program X and Program Z have unique user numbers associated with their partitions. This example illustrates that a user number does not indicate a unique physical location in memory. (See "User Number," earlier in this chapter.)

You can create a swap file or use the operating system swap file by default. (For details, see the CTOS System Administrator's Guide.)

APPLICATION PARTITION WITH MORE THAN ONE RUN FILE

Occasionally (but rarely), an application partition will contain more than one run file. This occurs when the original program in a memory partition calls the LoadTask operation to load an additional run file into the same partition.

In this situation, the original program is a primary task. Any subsequent run files are secondary tasks. These tasks have a very special relationship in that they share the partition's system data structures and resources. Because these tasks are interwoven and function as a group, each is not a program, but a dependent part of the overall program in the partition. Figure 32-7 shows the relationships of these tasks to the program in a partition. In this manual program can mean one or more run files in a partition.

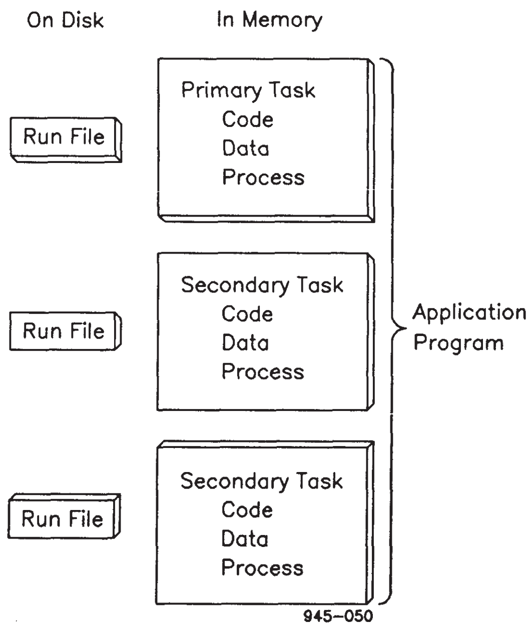


Figure 32-7. Program Consisting of More Than One Run File in an Application Partition

OPERATIONS

The program and partition management operations described below are categorized according to use. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

PROGRAM MANAGEMENT

The program management operations described below are categorized as operations used for error handling or for normal program loading and exiting from the same partition.

Error Handling

FatalError	Terminates operation of the application program after a catastrophic event.
CheckErc	Checks status codes. If CheckErc is called with a non-zero status code, FatalError is called with that value.
ErrorExit*	Terminates the current application program in an application partition and passes an abnormal status code to the exit run file.

*Dynamically installed system services use these operations at a certain time during installation. (For details, see Chapter 31, "System Services Management.")

ErrorExitString*
: Returns a string to the exit run file, which is usually printed.

Crash
: Causes operating system operation to terminate, a crash dump to be written, the operating system to be reloaded, and SignOn to display the cause of the crash when it is restarted.

SetMsgRet
: Same as ErrorExitString except the program does not exit.

Normal Program Loading and Exiting

Exit*
: Terminates the current application program in an application partition and passes a normal status code to the exit run file.

Chain*
: Replaces the current application program in an application partition with the specified run file.

SetExitRunFile
: Establishes a new exit run file for an application partition.

QueryExitRunFile
: Returns the name, password, and priority of the exit run file of an application partition.

---

*Dynamically installed system services use these operations at a certain time during installation. (For details, see Chapter 31, "System Services Management.")

TASKS

The operation below is used to load additional run file(s) into a partition that contains an existing run file(s). (For details, see "Application Partition with More Than One Run File," earlier in this chapter.)

LoadTask
: Loads and activates an additional (secondary task) run file as part of the current program in the application partition.

PARTITION MANAGEMENT

Basic Operations

GetUserNumber
: Allows a process to determine its own user number (which is historically the same as a partition handle).

GetPartitionHandle
: Returns the user number of a specified partition. The requesting process must supply the name of the requested user number's partition as a parameter to this operation.

GetPartitionStatus
: Returns status information about a specified application partition and the program currently executing in it.

SetPartitionName
: Changes the name of the requesting process's partition. (Note that SetPartitionName can change the name of any partition, but it is normally used to set the partition name of the caller.)

Program Swapping

SetSwapDisable Allows a program to specify that it can or cannot be swapped.

SwapInContext Requests that a specified user number's partition be swapped into memory.

Partition Creation Under Program Control

AssignKbd Assigns the keyboard to a partition.

AssignVidOwner Assigns the screen to a partition.

CreatePartition
Creates a new application partition, assigns its name, and returns a user number.

CreateBigPartition
Is the same as CreatePartition, except that CreateBigPartition allows you to create a new application partition that is larger than 1 megabyte (protected mode only).

CreateUser Creates a variable partition, specifying the size of the partition system data structures.

The program management operations described below are used by partition managing programs for loading programs into memory and for program exiting.

ExitAndRemove Terminates the current application program and removes the specified vacant partition. The user number is deallocated and becomes available to be reissued.

LoadPrimaryTask

Loads and activates the run file specified by the file specification in a vacant application partition.

LoadInteractiveTask

Is the same as LoadPrimaryTask but provides the additional option (by means of an fDebug parameter) to indicate whether or not the run file is to be debugged when it is loaded into the partition.

VacatePartition

Terminates the program in the application partition specified by the user number but does not load and activate the exit run file. The partition is left vacant.

RemovePartition

Removes the specified vacant application partition.

TerminatePartitionTasks

Terminates the program in the application partition specified by the user number and loads and activates the partition's exit run file.

Communication Between Partitions

SetPartitionLock

Declares whether a program executing in the specified application partition is locked. The locked partition cannot be vacated with the VacatePartition operation.

33 TIMER MANAGEMENT

The Timer Management facility provides for two types of system timers: a Realtime Clock (RTC) and a Programmable Interval Timer (PIT).

The RTC has a message-based interface you can use for accurate timing over long periods.

The PIT has a pseudointerrupt interface you can use for timing short intervals.

REALTIME CLOCK

The Realtime Clock (RTC) provides both the current date and time of day and the timing of intervals (in units of 100 milliseconds). (For a cluster workstation without a local file system, the current date and time are maintained at the master. For a cluster workstation with a local file system, the current date and time are maintained at both the master and at the cluster workstation.)

A client can request that a message be sent to a specified exchange either once after a specified interval or repetitively with a specified constant interval between send operations. The first time a message is sent to an exchange can be up to 100 milliseconds earlier than specified. Subsequent intervals are timed exactly.

PROGRAMMABLE INTERVAL TIMER

The Programmable Interval Timer (PIT) uses a 50 microsecond, high-resolution timing source. The PIT is controlled by a 16 bit counter and therefore has a maximum interval of approximately 3 seconds.

The PIT is used for high-resolution, low-overhead activation of user pseudointerrupt handlers. A client or an interrupt handler can request that a pseudointerrupt handler be activated after a specified interval.

TIMER MANAGEMENT OPERATIONS

There are three classes of timer management operations: Delay, Realtime Clock (RTC), and Programmable Interval Timer (PIT).

DELAY

The Delay operation allows a process to suspend execution for a specified interval (in units of 100 milliseconds).

REALTIME CLOCK

The OpenRTClock operation initiates the use of a data structure provided by a client for control of complex RTC services. This data structure, the Timer Request Block (TRB), is shared by the client and timer management. The CloseRTClock operation terminates sharing of the TRB.

The TRB defines the interval after which a message is sent to a specified exchange. The message can be sent either once after the specified interval or repetitively with the specified constant interval between send operations. The message is the memory address of the TRB itself.

The client must acknowledge receipt of the TRB (as described below) before timer management will send the same TRB again. This ensures that system resources (link blocks) are not consumed by queuing the same TRB at the same exchange many times. The client can also dynamically modify other fields of the TRB.

(See Table 4-29 in the <u>CTOS/VM Reference Manual</u> for the TRB format.)

Timer Management

Every 100 milliseconds, the timer management RTC interrupt handler performs the following sequence of operations on each active TRB. This sequence ensures that timer management will not send the same TRB again until the client decrements the cEvents field to 0.

1. If the counter field is 0, do nothing.

2. Decrement the counter field by 1.

3. If the counter field has not become 0, do nothing more.

4. If the cEvents field is 0, send a message to the exchange specified by the exchResp field. The message is the memory address of the TRB itself (not a copy of the TRB).

5. Increment the cEvents field by 1.

6. Copy the counterReload field to the counter field.

Timing a Single Interval

A client should use the sequence below to initialize a TRB to time a single interval.

1. Set the counter field to 0.

2. Call the OpenRTClock operation.

3. Set the cEvents field to 0.

4. Set the counterReload field to 0.

5. Set the counter field to the chosen interval.

Use the Wait or Check Kernel primitive (specifying the exchange specified by the exchResp field) to receive the indication that the interval expired. (Wait and Check are described in Chapter 29, "Interprocess Communication.") Remember that the RTC only operates in units of 100 milliseconds. Thus, if the counter field is set to 3, the TRB can be sent to the exchResp exchange in as few as 200 milliseconds or as many as 300 milliseconds. To reuse the TRB to time another single interval, repeat the sequence above from step 3.

Repetitive Timing

A client should use the sequence below to initialize a TRB for repetitive timing.

1. Set the counter field to 0.

2. Call the OpenRTClock operation.

3. Set the cEvents field to 0.

4. Set the counterReload field to the chosen interval.

5. Set the counter field to the chosen interval.

The first time that the TRB is sent to the exchResp exchange can be up to 100 milliseconds earlier than specified. Subsequent intervals are timed exactly. Exact timing is guaranteed because the counter field of the TRB is decremented even if the client has not finished processing the previous event. The cEvents field provides a continuous count of the events that have occurred but are not yet processed. If the client is too slow, the count in the cEvents field becomes ever larger. Under these circumstances, the count in the cEvents field provides a measure of how far behind processing has fallen.

The client should use the sequence below to process the TRB. This sequence avoids a race condition and yet processes the correct number of events.

1. Receive indication that the interval expired by using either the Wait or Check primitive and specifying the exchange specified by the exchResp field.

2. If the cEvents field is 0, processing is complete; return to step 1. (In this sequence, it is possible to receive a TRB in which cEvents is 0; thus it is necessary to perform this test before processing the event.)

3. Process the event. Processing is application-specific.

4. Decrement the cEvents field by 1. (It is not necessary to decrement the cEvents field in a single instruction unless the client is keeping a count of events.)

5. Repeat the processing sequence from step 2.

PROGRAMMABLE INTERVAL TIMER

The Programmable Interval Timer (PIT) is accessed through the SetTimerInt and ResetTimerInt operations.

The SetTimerInt operation establishes a pseudointerrupt handler in the application program to receive a pseudointerrupt after a specified interval (in units of 50 microseconds). (Pseudointerrupts are described in Chapter 36, "Interrupt Handlers.") The SetTimerInt operation specifies the memory address of a Timer Pseudointerrupt Block (TPIB) in user memory that must be allocated by the application.

(See Table 4-28 in the CTOS/VM Reference Manual for the TPIB format.)

NOTE: Other interrupt activity may result in a slightly longer PIT timed interval than requested. Very short requested intervals are particularly susceptible to this effect and can cause significant system overhead.

It is sometimes convenient to have a single pseudointerrupt handler service the pseudointerrupts associated with multiple TPIBs. To do this, the pRqBlkRet field of each TPIB must be the memory address of the same 4 byte memory area (or pRqBlkRet can be 0), and the SetTimerInt operation must be invoked for each TPIB. The pseudointerrupt handler must examine this 4 byte memory area to determine which TPIB caused activation of the pseudointerrupt handler. Even when the pseudointerrupt handler is serving only a single TPIB, pRqBlkRet must still be the memory address of the otherwise unused 4 byte memory area (or pRqBlkRet can be 0).

The ResetTimerInt operation terminates a previous SetTimerInt operation.

OPERATIONS

The timer management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

DELAY

Delay	Delays the execution of the client for the specified interval.

REALTIME CLOCK

OpenRTClock	Establishes a TRB between the client and timer management.
CloseRTClock	Terminates the use of the specified TRB.

PROGRAMMABLE INTERVAL TIMER

SetTimerInt	Establishes a PIT pseudointerrupt handler.
ResetTimerInt	Terminates the TPIB initiated by a SetTimerInt call.

34 VIRTUAL CODE MANAGEMENT

The Virtual Code Management facility (commonly known as the "Swapper") allows you to run a program that is larger than the available memory in an application partition. The Virtual Code Management facility is a set of object module procedures in the standard operating system library, CTOS.lib. These modules are linked with the program and become part of the run file. For protected mode, part of the Virtual Code Management facility also is in the operating system itself.

This chapter presents the Virtual Code Management facility from a theoretical point of view. It describes how the operating system handles the movement of program segments between disk and memory. For practical guidelines on how to incorporate the Virtual Code Management facility into your programs, see the CTOS Programmer's Guide.

Each application program using the Virtual Code Management facility is divided into variable-length code segments. The segments contain one or more complete procedures in object modules. One or more code segments are resident in memory. The others reside on disk in a run file. The Virtual Code Management facility brings them into memory as they are needed.

A code segment in memory that is no longer needed is discarded, and another code segment (called an overlay) is read into memory. When the first code segment is needed again, it is reread from the run file. Under this system, only code segments, and not data segments, are read into memory and discarded as necessary. Nothing is written back to disk, so there is no need for a disk swap file.

You can write a program with the intention of using the Virtual Code Management facility, or you can rather easily retrofit an existing program to use it. Few, if any, source program changes are needed: using the Virtual Code Management facility mainly involves specifying to the Linker your desired grouping of object modules into code segments.

The "Virtual Code Segment Management" section in the CTOS Programmer's Guide provides an overview of how to specify the modules you want to place in overlays. (Additional information is contained in the Linker/Librarian Manual, Section 1, "Using the Linker (Binder)." Also see Section 4, "Further Information About Linker Options," in the same manual.)

MODEL OVERVIEW

The Virtual Code Management facility allows the execution of programs whose code size exceeds the size of the partition in which they are run. To achieve this, only portions of the code exist in memory at any given time; the remainder are on disk. It is the job of the Virtual Code Management facility to ensure that the portions of the code that are currently needed for execution are actually in memory.

The code in the run file of a program using the Virtual Code Management facility either is part of one of several overlays, or is resident. (Hereafter, a program that uses the Virtual Code Management facility is called an overlay program.) When the overlay program begins execution, the resident code is loaded into memory, where it remains for the duration of the program's execution. At some point in the program's execution, when a call is made to a procedure in one of the overlays, the Virtual Code Management facility reads that overlay into memory into an area of memory called the overlay zone so that the program can continue execution.

The Virtual Code Management facility keeps as many overlays as possible in memory at once. When another overlay that would exceed the available space is called into memory, the Virtual Code Management facility uses a least-recently-used (LRU) algorithm to determine which currently resident overlay to discard.

The Virtual Code Management facility is designed to run in both real mode and protected mode and participates with the compatible run file format. That is, if an application program is written following the rules for protected mode programs, a single overlay program run file can be created that will run in both real mode and protected mode. Which mode it actually runs in depends on which operating system is present. (Guidelines for writing protected mode programs are contained in the Engineering Update for 2.0 CTOS/VM.)

The Virtual Code Management facility operates quite differently in protected mode than in real mode.

DATA STRUCTURES

The Virtual Code Management facility uses several data structures to keep track of the current locations of all of an overlay program's procedures (in memory or on disk, and in what overlay).

When an overlay program is linked, the Linker builds several data structures within it for use by the Virtual Code Management facility. When the program is running, the arrangement of its parts is as shown in Figure 34-1. The program's resident code and data are in high memory.

To show all aspects of the arrangement, the figure depicts the memory layout at some point during program execution, after several overlays have been brought into memory and discarded.

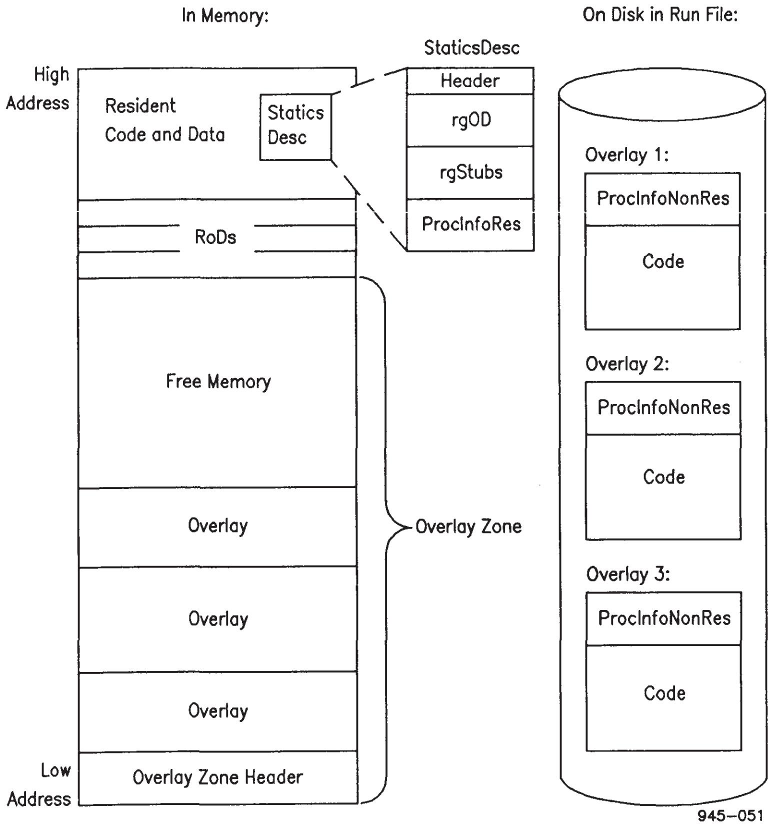


Figure 34-1. Virtual Code Facility Data Structures and Their Locations

OVERLAY ZONE HEADER

The overlay zone header is at the low end of the overlay zone. This structure describes the overlay zone, indicating how much space is used by overlays and (for real mode only) how much by return overlay descriptors (RODs). (For details on using RODs, see "Intercepting Returns," later in this chapter.) It also contains other reference information, including the locations of the StaticsDesc data structure and some of its substructures (described next).

STATICSDESC

The StaticsDesc structure is in the data segment (DGroup) of the overlay program. It consists of the following:

- a self-descriptive header
- an array of overlay descriptors (rgOD)
- an array of stubs (rgStubs)
- a ProcInfoRes structure

The overlay descriptors array (rgOD) contains an entry for each overlay in the program, indexed by overlay number. Each overlay descriptor identifies the location and size of the overlay in the run file.

The stubs array (rgStubs) contains a stub for each program procedure. In protected mode, the procedure's stub contains the protected mode selector (SL) and the offset of the procedure. (For details on protected mode SLs, see Chapter 3, "Using CTOS/VM Operations," and Chapter 24, "Memory Management.") In real mode, the stub for a procedure contains either the address of the procedure's current address in memory or, if the stub's procedure is not resident in memory, the address of the OverlayFault procedure.

The ProcInfoRes structure describes those procedures that are in the permanently resident portion of program code. Its header tells how many procedures are present in the resident code segments and identifies the index of the stub corresponding to the first public procedure in the resident. A public procedure is a procedure that can be accessed by other modules.

RETURN OVERLAY DESCRIPTORS

The return overlay descriptors (RODs) are overlay identifiers used by the Virtual Code Management facility in real mode when a return is done to a procedure that was discarded after it issued the corresponding call. (For details, see "Intercepting Returns," later in this chapter.) RODs are not used in protected mode.

PROCINFONONRES

All code segments in overlays reside in the overlay program's run file on disk. The ProcInfoNonRes structure is at the head of each overlay code segment. It contains the index of the corresponding overlay descriptor (for example, what overlay this is) and its size. It also contains a time-stamp field for use with the LRU algorithm.

Like the ProcInfoRes structure, ProcInfoNonRes identifies the index in the stubs array of the stub corresponding to the first procedure in this overlay that can be accessed by other modules. Additionally, it tells the number of procedures in the overlay. Finally, it identifies these procedures as near or far:

- A near procedure is referenced by the offset (IP) of the procedure's memory address. Near procedures can be called only by other procedures within the same module.

- A far procedure is referenced by both its code segment (CS) and its offset (IP). Far procedures can be called by procedures within the same or from within a different module.

(For details on how the Virtual Code Management facility handles these procedures, see "Intercepting Returns," later in this chapter.)

In real mode, the Virtual Code Management facility needs the information provided by the ProcInfoNonRes structure when it traces the stack to discard an overlay.

The stubs array contains one stub for each program procedure. The Linker changes all program procedural calls from Call Direct to Call Indirect as follows:

```
CALL DWORD PTR [stub + 1]
```

Thus, each procedure is called through its corresponding stub.

A stub has the 5 byte structure shown in Figure 34-2. In real mode, the first byte is either a JMP or a CALL instruction (opcode); in protected mode, the first byte always is a JMP. The remaining 4 bytes (in either mode) are a procedural address.

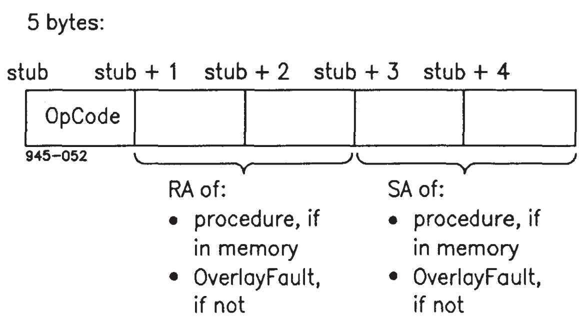


Figure 34-2. Stub Structure

PROTECTED MODE OPERATION

In protected mode, because each overlay is a separate segment, each overlay has a unique descriptor in the Local Descriptor Table (LDT). (See the iAPX Programmer's Reference Manual and the 80386 Programmer's Reference Manual for details.) The present bit within the descriptor indicates whether or not the segment is in memory. When an overlay program is first loaded into memory, the operating system marks the descriptors for all the overlays as not present.

Whenever any of the discarded overlays are referenced (whether a procedure is being called or being returned to), a segment not present fault will occur. A segment not present fault is an interrupt from which control is passed to the segment not present fault interrupt handler. (For details, see Chapter 36, "Interrupt Handlers.") The segment not present fault interrupt handler is the part of the Virtual Code Management facility that resides in the operating system.

The segment not present handler must determine which overlay is needed before it can read it into memory. The processor supplies to the handler the selector (SL) that caused the fault. The Virtual Code Management facility knows the SL of the first overlay in the LDT. It, therefore, can determine the overlay number for the desired overlay. It then uses the overlay number to index into the overlay descriptors array to find the address of the overlay on the disk. The Virtual Code Management facility then

1. makes room in the overlay zone

2. reads in the overlay

3. updates the descriptor to reflect the overlay address

4. sets the descriptor present bit

5. restarts the instruction

REAL MODE OPERATION

INTERCEPTING CALLS

In real mode, the stub contains the address of the OverlayFault procedure for each nonresident procedure (which, when the program is loaded, includes all procedures in overlays).

When a nonresident procedure is called, the call goes indirectly by means of the stub to OverlayFault. The OverlayFault procedure

- determines which overlay it should bring into memory by analyzing its own address, which is constructed using a flexible additive address mechanism

- examines the last 2 bytes of the original Call Indirect instruction to determine which stub the call came through, and therefore which procedure within the overlay is desired

INTERCEPTING RETURNS

NOTE: The following discussion assumes knowledge of stack format. (See the "Languages, Stack, and Calling Conventions" section in the CTOS Programmer's Guide for details.)

In real mode, the Virtual Code Management facility also intercepts returns to calling procedures. A calling procedure may be discarded from memory before it receives a return. A fatal error would occur if a return were made to a memory location previously occupied by a procedure that had since been discarded.

When the Virtual Code Management facility has chosen an overlay to discard, it performs the following procedures:

1. It traces the stack.

2. It finds the return address of the procedure being discarded.

3. It overwrites the return address with the OverlayReturnFault procedure's address.

This trace (exemplified below) is possible because the current stack base pointer (BP) is the memory address of the stack containing the BP address of the previous frame. (A frame is all of the information that is pushed on the stack when a procedure is called. The frame includes the parameters passed to the procedure and the information the procedure itself pushes on the stack during the course of execution.)

The BP of the previous frame, in turn, contains the previous BP, and so on, in a chain. The Virtual Code Management facility follows the chain of BPs, checking the return addresses as it goes, and overwriting any in the discarded procedure with the OverlayReturnFault address.

At this time, a return overlay descriptor (ROD) also is created for the discarded overlay, if the discarded overlay has any returns outstanding.

When the return occurs, it goes to OverlayReturnFault, the address of which now appears as the return address on the stack. The ROD identifies the overlay needed and the procedure within that overlay. OverlayReturnFault then brings this overlay into memory and passes control to the procedure, thus completing the call/return cycle.

OverlayReturnFault now marks the ROD as free so that RODs do not accumulate. (The number of existing RODs at any given time always equals the number of nonresident procedures with outstanding calls.)

Figure 34-3 illustrates stack tracing when an overlay is discarded.

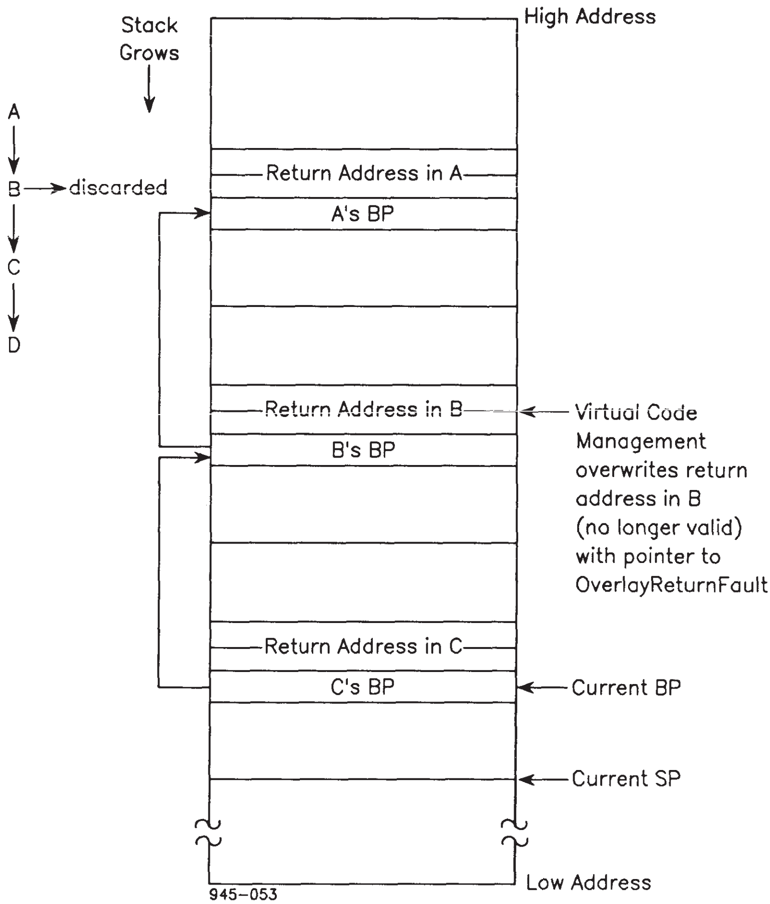


Figure 34-3. Tracing the Stack When an Overlay Is Discarded

The scenario leading up to Figure 34-3 is described as follows.

When Procedure A (in the resident portion of code) calls Procedure B (in an overlay), the Overlay Manager brings B into memory. B, in turn, calls Procedure C (also in an overlay), and C is brought into memory. Now Procedure C attempts to call Procedure D, but there is not enough room for D's overlay in the overlay zone.

The Virtual Code Management facility examines its statistics and concludes that Procedure B is in the LRU overlay and should therefore be discarded. (Procedure B still expects a return from Procedure C.) The Virtual Code Management facility discards the overlay containing B, creating a ROD to identify B. During this process, the Virtual Code Management facility must trace the stack to overwrite the return address of B with the OverlayReturnFault address.

Figure 34-3 shows the stack format at this point. By convention, the BP register contains the addresses of variables that are local to the current procedure. It is therefore necessary that the program save the BP of the calling procedure so that it can be restored at the return. The positions of the BPs and the return addresses of the procedures are shown in Figure 34-3. Each BP contains the address of the previous BP. The Virtual Code Management facility can jump from BP to BP, examining each return address and overwriting any return address belonging to the overlay that is being discarded.

When the Virtual Code Management facility reaches a BP containing an address that matches the saved address of the initial stack pointer (SP), it terminates the trace.

Figure 34-3 is a simplification showing only far calls and returns. The structures, ProcInfoRes and ProcInfoNonRes, identify each procedure within their overlays as near or far. The Virtual Code Management facility refers to these structures to determine whether it should read both a CS and an IP as a procedure's address (for a far procedure) or only an IP (for a near procedure).

After the overlay has been discarded, the Virtual Code Management facility compresses the remaining overlays toward the low end of the zone and brings in D.

Procedure D now executes and then returns to C, which is straightforward because C is still in memory. C, however, returns to the address on the stack where B's return address normally would be, but the return now goes to OverlayReturnFault.

OverlayReturnFault analyzes this return address, accesses the correct ROD, and determines that the overlay that contains Procedure B is must be brought back into memory. It then swaps B in, discarding another overlay if necessary. (Note that it is perfectly acceptable to discard the returning procedure to bring in the procedure receiving the return.)

IMPORTANCE OF CALL/RETURN CONVENTIONS

Because of this stack-tracing scheme, you must adhere to accepted call/return conventions. If the stack format is not what the tracing algorithm expects, the overlay program fails during the process of discarding an overlay. Note that this is important to the Virtual Code Management facility only if your program executes in real mode. The Virtual Code Management facility does no stack tracing in protected mode. Nevertheless, you should follow these conventions to create a compatible run file (that is, a run file that allows your program to operate correctly in protected mode and real mode.

REAL AND PROTECTED MODE OPERATION

CALLS TO PROCEDURAL ADDRESSES

In some programs, it is necessary to call a variable that is a procedure address rather than the actual procedure. The actual procedure to be used may be determined only at run time.

The Virtual Code Management facility can handle such calls as well as standard procedure calls. The first byte of the stub is either a JMP or a CALL instruction.

In an overlay program, the Linker assigns the address of the stub's first byte to all memory locations within a program that contains references to the procedure. If the procedure to be called is

- Resident in memory., this byte is the JMP instruction, and the remaining 4 bytes are the address to which the jump should occur.

- Not resident in memory, there are two cases. For protected mode, the stub's first byte is the JMP instruction, and the remaining 4 bytes are the address of the procedure. The referenced descriptor is marked "not present." For real mode, the stub's first byte is the CALL instruction, and the remaining 4 bytes are the address of the OverlayFault procedure, which in turn brings the needed overlay into memory.

For real mode nonresident procedures, OverlayFault knows from what stub it was called and thus can determine what procedure is needed.

ADJUSTING ADDRESSES

In real mode, once an overlay has been brought into memory, the Overlay Manager overwrites the stub address of a frequently called procedure with that procedure's actual current address in memory. Thereafter, performance is improved as calls to that procedure go to it directly until that overlay is discarded.

The Overlay Manager keeps track of calls to an overlay while the overlay is in memory. By doing this, the Overlay Manager can determine the most active overlays, which are retained in memory.

In real mode, however, once a procedure stub address has been overwritten with the procedure's actual memory address, calls to the procedure no longer go through OverlayFault and are not logged. To compensate for this omission, your program can call the MakeRecentlyUsed operation, which prevents an overlay from being inadvertently discarded from memory. This operation is unnecessary in protected mode: if it is called, it will perform no function other than to return status code 0 (ercOK).

When several overlays are in memory and the Overlay Manager needs to bring in another one for which there is not enough room, it uses this call-frequency data with its LRU algorithm to choose an overlay to swap out.

To enable reinitialization of its frequency-of-use log and to determine the new pattern of overlay use, the following compression procedure is performed:

- All remaining overlays are compressed toward low addresses.

- For real mode, the actual addresses of procedures within the overlays change. For protected, the descriptors for each moved overlay are updated to reflect their new locations.

- For real mode, all stubs are readjusted to the OverlayFault's address.

After this compression, the new overlay is brought into memory just above the highest existing overlay.

Overlays are available to programs that consist of more than one run file in an application partition. (For details, see Chapter 32, "Program and Partition Management.") The first run file contains the _primary task_ and is loaded by the Chain, ErrorExit, Exit, LoadInterActiveTask, or the LoadPrimaryTask operation. A subsequent run file loaded into the same partition contains a _secondary task_ and is loaded by the LoadTask operation. A secondary task, however, cannot be virtual if the primary task already uses Virtual Code Management.

OPERATIONS

The Virtual Code Management operations described below are categorized as basic or advanced. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

BASIC

InitOverlays	Initializes the Virtual Code Management facility.
InitLargeOverlays	Initializes the Virtual Code Management facility for large overlays.

ADVANCED

GetOvlyStats	Returns the size of the largest overlay, the size of the second largest overlay, and the total size of all overlays.
GetCParaOvlyZone	Returns the size of the overlay buffer measured in paragraphs.
ReInitOverlays	Allows the user to change the size of the overlay buffer to recover memory or extend the overlay buffer for better performance.
ReInitLargeOverlays	Is identical to ReInitOverlays, except the user describes the length of the overlay buffer as a count of paragraphs instead of bytes.

MoveOverlays — Changes the location of the overlay zone.

MakePermanent — Makes the overlay permanently resident in memory until it is released with a call to ReleasePermanence.

MakePermanentP — Makes an arbitrary overlay permanently resident in memory until it is released with a call to ReleasePermanence.

ReleasePermanence — Releases all overlays from permanent residence in memory.

MapIOvlyCs — Takes an overlay index and returns the address in memory of where the overlay is currently located.

MapCsIOvly — Takes the CS part of a memory address and returns the overlay in which that address is currently contained.

MapPStubPProc — Returns the last 4 bytes of a stub, which contain the address of a procedure.

MakeRecentlyUsed — Prevents an overlay from being inadvertently swapped out.

UpdateOverlayLRU — Is called from within one overlay to prevent any other overlay from being swapped out by updating the time of its most recent use so that it appears to have 0 age.

EnableSwapperOptions	Allows an arbitrary operation to be called each time OverlayFault is called. This call works in real mode only.
DeallocateRods	Removes outstanding RODs when the stack is unwound in an assembly language program.
ReInitStubs	Sets all stubs, as a one-time reset, to contain the OverlayFault address.

35 QUEUES AND QUEUE MANAGEMENT

QUEUES

A queue is a linked list of priority-ordered queue entry records. A file that contains a queue is called queue entry file. Queues are used by application programs and system services to communicate data within a workstation or between workstations. Because queues are disk-based, the data is preserved across system reboot or a power failure. Note that data is not preserved in similar circumstances when interprocess communication (IPC) or inter-CPU communication (ICC) is used.

Each queue entry file contains information for a single type of processing, such as spooled printing, BSC 3270 remote job entry (RJE), or SNA RJE. This information is created, accessed, and modified by both clients and servers, such as the spooler, BSC 3270 RJE, or SNA RJE.

To take advantage of queues, you must install the Queue Manager. The Queue Manager can be installed on a master or a standalone workstation.

The queue entry file consists of a header record followed by a series of queue entry records.

- The queue header contains all data that the Queue Manager needs to control the file.

- Each queue entry record consists of Queue Manager control information followed by specific data created and read by the client.

QUEUE MANAGER

The Queue Manager is a system service that maintains queues. It provides services such as adding or deleting queue entries, setting queue entries to be in service, or returning queue status information.

RUN FILES

The Queue Manager consists of two run files:

- The InstallQMgr.run run file is the program that installs the Queue Manager.
- The DeinstallQMgr.run run file is the program used to deinstall the Queue Manager.

INSTALLATION/DEINSTALLATION

The Queue Manager can be installed on a master or on a standalone workstation.

In a cluster configuration, the Queue Manager must be installed at the master. The servers that use the Queue Management facility, however, can be installed at cluster workstations as well as at the master. Multiple servers in different cluster workstations can serve the same queue simultaneously.

To install the Queue Manager, you can use

- A batch or Command Line Interpreter (CLI) utility when the system is bootstrapped. (See the CTOS System Administrator's Guide for details on the batch/CLI utilities.)

- The Executive **Install Queue Manager** command. This command allows you to configure use of the Queue Manager for greater flexibility. (See the Executive Manual for details.)

- The Print Manager. (See the Printing Guide for details.)

The Queue Manager can be deinstalled by running the program, DeinstallQMgr.run, or by using the DeInstallQueueManager operation.

OVERVIEW OF QUEUE MANAGEMENT

The queues used in the system can be defined by the administrator, application programs, or servers. Each queue is assigned a unique name and a queue entry file specification.

Clients can then add queue entries by using operations that reference a queue name. The client need not specify the location of the queue server. The first available server in the cluster can serve the queue entry.

Figure 35-1 shows an example of a cluster configuration with the Queue Management facility, a client, and a server (spooler).

The Queue Management facility acts as a central switch between clients and servers.

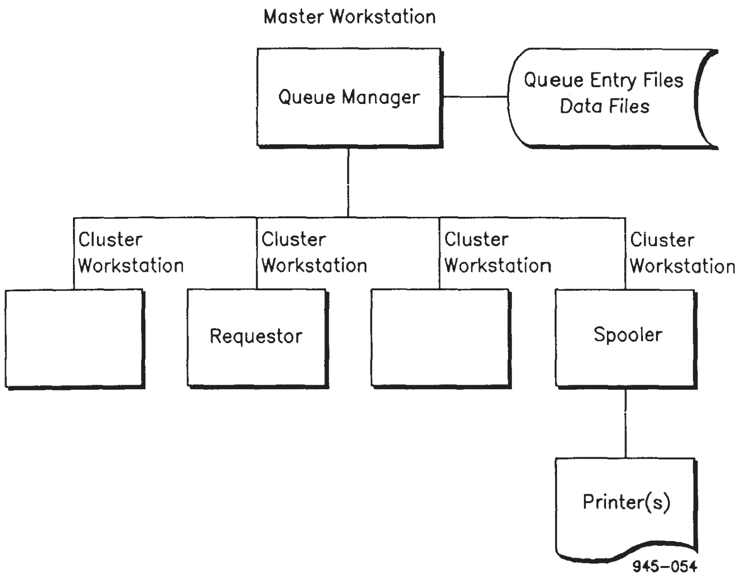


Figure 35-1. Example of a Configuration with the Queue Management Facility

CLIENTS

Clients submit requests for processing services, such as printing and transmission of files, to the Queue Manager. By using the Queue Management facility, clients can

- access queue entry files by using operations that specify the queue name
- submit entries to the appropriate queue
- delete previously queued entries
- obtain a list of entries queued

SERVERS

Servers (such as the spooler, BSC 3 270 RJE, and SNA RJE) serve the queue entry files. The Queue Management facility allows the server to

- specify the queue(s) they will serve
- process entries in the specified queue(s)
- request the removal of processed queue entries

SEQUENCE FOR USING QUEUE MANAGEMENT

A simplified sequence for installing and using the Queue Management facility is described below.

1. If an application program or a system service does not create queues dynamically, the system administrator can create a Queue Index File in the master. The Queue Index File is a text file that defines queues to be used in the system. The Queue Index File assigns to each queue a queue entry file for storing queue entries submitted by clients, the size of the queue entry, and the queue type.

   If queues are created dynamically, creation of the Queue Index File can be omitted. Dynamically installed queues are defined in the same way as queues defined by the Queue Index File. (For details, see "Dynamically Manipulating Queues," later in this chapter.)

2. The Queue Manager is installed on the master or the standalone workstation with a batch/CLI utility, the Executive **Install Queue Manager** command, or the Print Manager. The system administrator can choose to specify a maximum number of queues at installation time.

3. If a Queue Index File exists, the installed Queue Manager opens the queues in the Queue Index File. The queues are maintained in the master.

4. At any time after the Queue Manager is installed, servers or application programs can add queues with the AddQueue operation. The Queue Manager adds queues created by AddQueue to its tables, and it creates, if necessary, and opens a queue entry file. The number of running queues must not exceed the maximum number specified when the Queue Manager was installed.

5. A server (such as a spooler or RJE) intending to serve a particular queue uses the EstablishQueueServer operation to establish itself as an active queue server.

6. A client adds queue entries to the specified queue with the AddQueueEntry operation.

7. The server obtains a particular queue entry for processing with the operation MarkKeyedQueueEntry (or the next available queue entry with the MarkNextQueueEntry operation). The Queue Manager marks the queue entry as being in use to prevent other servers from operating on it. The marked queue entry remains in the queue entry file until it is removed (next step).

8. The server services the marked queue entry and then removes the processed entry from the queue entry file using the operation RemoveMarkedQueueEntry.

9. To discontinue serving a queue, the server removes itself from the list of active servers with the TerminateQueueServer operation.

QUEUE INDEX FILE

The Queue Index File is a text file that defines queues to be used in the system. It contains information such as the name of each queue to be used in the system and the associated queue entry file.

Queues also can be defined by the AddQueue operation. Both methods of defining queues can be used. (For details, see "Dynamically Manipulating Queues," later in this chapter.)

If required, the system administrator creates the Queue Index File [Sys]<Sys>Queue.Index in the master.

The Queue Index File is created with the Text Editor, Word Processor, or Document Designer. A record of the following format is required for each queue:

```
queueName/fileSpec/entrySize/queueType <RETURN>
                          .
                          .
                          .
```

where

queueName Is a user-defined queue name that is unique to the installation. The name can be any name of up to 50 characters, except the following system device names: COMM, KBD, LPT, NUL, PTR, TAPE, QIC, VID, and X25. Examples of acceptable names are SpoolerA, SPL, PrinterX, Centronix, Diablo, and RJEtoBoston.

fileSpec Is the file specification of the queue entry file in which queue entries submitted by clients are stored. An example would be [Winl]<Sys>SpoolerAQueueEntryFile).

entrySize Is the size of an entry for the queue entry file. The size is the number of 512 byte sectors per entry. For example, to define 1K byte entries, specify an entry size of 2. In this case, 984 bytes are usable, and 40 are reserved for the Queue Manager.

queueType — Is the type of the queue (an integer less than or equal to 255), which enables a consistency check. The Queue Manager checks the type against the type in operations to add entries to the queue and to establish servers for the queue. Types 0 through 80 are reserved for internal use. Types 1, 2, and 3 are assigned as follows:

Type	Assignment
1	Spooler queue
2	RJE queue
3	Batch queue

An example of a Queue Index File is shown in Figure 35-2.

```
SpoolerA/SpoolerAQueueEntryFile/1/1 <Return>
RJEBoston/RJEBostonQueueEntryFile/1/2 <Return>
```

945-055

Figure 35-2. Example of a Queue Index File

DYNAMICALLY MANIPULATING QUEUES

Application programs or servers can add queues dynamically by calling the AddQueue operation and supplying the same information that is contained in the Queue Index File record fields. (See the AddQueue operation in Chapter 3, "Operations," in the <u>CTOS/VM Reference Manual</u>.)

AddQueue can be used to add queues whether or not the system administrator created a Queue Index File.

A queue that is dynamically added has the additional feature of being able to be manipulated,. It can be either

- reset to empty by the CleanQueue operation

- removed by the RemoveQueue operation

The AddQueue operation has a queue handle parameter that allows the queue to be accessed by CleanQueue and RemoveQueue.

Queues in the Queue Index File do not have the same flexibility. The Queue Index File can be edited at any time, but changes to it do not take effect until the Queue Manager is reinstalled.

With the exception of the advantages described above for dynamically installed queues, all queues work the same way, as described in the remainder of this chapter.

QUEUE ENTRY FILE

Clients add entries to queue entry files (queues). In the case of RJE, entries are added to the transmit queue and removed from the receive queue. The control and status queues are used internally by the servers for control and status purposes.

For further information on the queues required in the Queue Index File, see

- Appendix A of this manual for the spooler
- the _2780/3780 RJE Terminal Emulator Manual_ for RJE

Each queue-oriented service generally requires more than one type of queue, although only one queue entry file is illustrated for each queue name in Figure 35-2. (See Table 35-1.)

The client specifies the queue name when submitting a queue entry for processing. The queue entry is automatically placed in the appropriate queue by the Queue Manager.

If a Queue Index File exists, the installed Queue Manager opens the queues specified in the Queue Index File. If a queue does not exist, it is created.

If a queue has insufficient space for adding an entry, the Queue Manager expands that queue by an increment sufficient to contain 30 entries.

QUEUE ENTRY FILE FORMAT

A queue entry file contains information for a single type of processing such as spooled printing or RJE.

Each queue entry file consists of a header record followed by a series of queue entry records.

The header contains all data that the Queue Manager needs to maintain the file. This data includes

- the queue type, such as spooler or RJE
- the queue version
- a listing of all queue servers
- two sets of head and tail pointers to a doubly linked list of queue entries

As a consistency check, the Queue Manager matches the queue type against the type in all client and server requests.

The queue version checks the queue entry file version against the Queue Manager version. A match ensures correct queue interpretation.

Two sets of head and tail pointers contain memory addresses in a doubly linked list of queue entries.

- One set contains the addresses of the first and last entries available for use.
- The other contains the addresses of the first and last entries currently being served or waiting to be served.

The entries are priority-ordered such that new entries are inserted after the last entry of higher priority, and before the first entry of lower priority.

QUEUE ENTRY FILE EXAMPLES

More than one type of queue entry file is generally required for each queue-oriented service. (For example, scheduling, control, and status queues are required for a spooler queue.) Table 35-1 shows examples of typical queues.

Table 35-1
EXAMPLES OF QUEUES

Server	Type	Number Required
Spooler	Scheduling Control Status	One per print class One per printer One per cluster configuration
Remote Job Entry (RJE)	Transmit	One per cluster configuration
Receive		One per cluster configuration

QUEUE ENTRY

A queue entry is a formatted request for processing that is added by clients to the specified queue. Clients and servers communicate by means of fields within the queue entries located at fixed offsets known to both the clients and the servers. When a server is available, it obtains a queue entry for processing.

A queue entry is a number of contiguous 512 byte sectors in a queue entry file. Each queue entry consists of the following two parts:

- The first 40 bytes are reserved for the Queue Manager and include control information, (For details, see "Queue Status Block," later in this chapter.)

- The remaining bytes are type-specific, that is, they are specific to the type of the queue. (See Tables A-1 through A-3 in Appendix A, "Spooler Management," for examples of spooler queue entries.)

CLIENT OPERATIONS

A client can add entries to queues, read queue entries (typically, to determine the sequence and status of entries), and delete specific queue entries.

ADDING AN ENTRY TO A QUEUE

A client adds an entry to the specified queue with the AddQueueEntry operation. The client specifies information, including

- A queue name that must correspond to an already created queue.

- A priority level (0 to 9 with 0 the highest), at which the entry is queued.

- The memory address of a buffer containing the type-specific portion of the queue entry.

- An optional time specification for the earliest time the entry is serviced,

- An optional time interval for requeuing of the entry after its removal from the queue entry file. The time interval is added to the time specification for servicing the entry.

Before adding a new entry to the queue, the Queue Manager checks the number of active servers. If no servers are actively serving the queue, some clients may select not to queue a new entry.

READING QUEUE ENTRIES

A client reads queue entries with the ReadNextQueueEntry operation for each entry to be read. ReadNextQueueEntry is typically used to list the contents of all entries by using commands such as the **Spooler Status** command. (See the Executive Manual.)

The client specifies the queue name, queue entry handle, and memory addresses of buffers to which the queue entry and Queue Status Block are returned. (See the following sections.)

Queue Entry Handle

A queue entry handle is a 32 bit integer that uniquely identifies a queue entry. The control portion of the queue entry (the first 40 bytes that are reserved for the Queue Manager) contains the queue entry handle of the logically following queue entry.

Queue Status Block

The MarkKeyedQueueEntry, MarkNextQueueEntry, and ReadQueueEntry operations accept a parameter that is the memory address of a Queue Status Block. These operations use the Queue Status Block to report a queue entry's server user number, priority, and the buffers in which the queue entry handles for the queue entry and the logically following queue entry are stored.

(See Table 4-21 in the CTOS/VM Reference Manual for the structure of the Queue Status Block.) The Queue Status Block is part of the control portion of the queue entry (the first 40 bytes that are reserved for the Queue Manager).

REMOVING AN ENTRY

A client removes a specific queue entry from the queue with the RemoveKeyedQueueEntry operation. The queue entry is identified by one or two key fields.

A key is a particular field or combination of fields in a data record upon which the search process is performed. The RemoveKeyedQueueEntry operation can specify that up to two key fields must match corresponding fields in the queue entry before the queue entry is removed.

SERVER OPERATIONS

A server can do all of the following:

- establish itself as an active server for the specified queue(s)
- mark and obtain queue entries for processing
- unmark queue entries or remove itself as an active server

ESTABLISHING SERVERS

A server must establish itself as a server for a specific queue with the EstablishQueueServer operation before it can serve the queue.

EstablishQueueServer enables the Queue Manager to keep a count of the number of servers serving each queue. The Queue Manager checks the count of servers before adding entries to a queue. If no servers are active, a client may select not to queue a new entry.

MARKING QUEUE ENTRIES

The server obtains a queue entry on which to operate with either of two operations:

- the MarkNextQueuedEntry operation to specify the next available queue entry
- the MarkKeyedQueueEntry operation to specify a specific queue entry

The Queue Manager marks the specified queue entry as being in use to prevent other servers from operating on it.

The marking operations prevent interference among multiple servers serving a single queue. When a queue entry is marked, it is not returned in subsequent marking operations.

UNLOCKING QUEUE ENTRIES

Entries are reset to the unmarked (not in use) state when

- The Queue Manager is installed.
- A server terminates operation for any reason, including malfunction of a cluster workstation. The Queue Manager searches all queues affected and resets any queue entries marked by servers from the malfunctioning workstation.
- A server no longer wishes to serve a queue and issues a TerminateQueueServer operation. The Queue Manager decrements the count of active servers for that queue and resets all entries previously marked by the terminating server.

QUEUE ENTRY FORMATS

(See Tables A-1 through A-3 in Appendix A, "Spooler Management," for the formats of the spooler scheduling, status, and control queues, respectively.) The queue entry format also can be used for user-defined servers. Queue entries must be large enough to accommodate the control portion of the queue entry (40 bytes that are reserved by the Queue Manager).

OPERATIONS

The Queue Management operations described below are categorized by user group. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

CLIENT GROUP

AddQueueEntry
: Adds an entry to the specified queue for processing by the appropriate queue server.

ReadKeyedQueueEntry
: Obtains the first queue entry in the specified queue with up to two key fields equal to the values specified, reads it into a buffer, and returns the Queue Status Block.

ReadNextQueueEntry
: Reads an entry from the specified queue into a buffer and returns the queue entry handle of the next queue entry.

RemoveKeyedQueueEntry
: Locates an unmarked entry in the specified queue with up to two key fields equal to the values specified and removes it from the queue.

SERVER GROUP

EstablishQueueServer
: Establishes that a server intends to service the specified queue.

MarkKeyedQueueEntry
: Locates the first unmarked entry in the specified queue with up to two key fields equal to the values specified, marks it as being in use, reads it into a buffer, and returns a queue entry handle for use in a subsequent RemoveMarkedQueueEntry operation.

MarkNextQueueEntry
: Leads the first unmarked entry in the specified queue into a buffer, marks it as being in use, and returns a queue entry handle. Entries are marked in order of priority.

RemoveMarkedQueueEntry
: Removes a previously marked entry from the specified queue.

RewriteMarkedQueueEntry
: Rewrites the specified marked queue entry with a new queue entry.

TerminateQueueServer
: Notifies the Queue Manager that a server is no longer serving the specified queue.

UnmarkQueueEntry
: Resets the specified queue entry as unmarked (not in use).

CLIENT/SERVER GROUP

The operations below can be used by any client or server.

AddQueue	Activates a new queue.
CleanQueue	Resets a queue to empty.
DeInstallQueueManager	Terminates operation of the Queue Manager and frees its memory partition.
GetQMStatus	Interrogates the Queue Manager about usage statistics, as well as the queues of the specified type.
RemoveQueue	Removes a queue dynamically.

36 INTERRUPT HANDLERS

To most programmers, interrupts are invisible events, handled automatically by system software. This chapter will be of interest primarily to systems programmers, communications programmers, and others concerned with handling low-level devices or program instruction errors.

TERMINOLOGY

The Intel microprocessors, upon which the operating system is based, support an interrupt handling mechanism that can be used for a variety of different purposes. For this reason, CTOS/VM supports a number of interrupt handling styles, some of which are only very distantly related.

To clarify differences, the following terms are used throughout this manual wherever interrupt handling is discussed. Note that these terms are not specifically CTOS concepts: they are terms used for the Intel family of microprocessors.

An interrupt is one of several types of control transfers initiated by the processor because of an event that requires immediate attention.

An Interrupt Vector Table (IVT) is an array of program addresses maintained by the operating system. When an interrupt occurs (in real mode), the processor hardware consults this table to decide where to transfer control. The table has 256 entries, each of which can correspond to a different interrupt source. All real mode interrupts are directed to an interrupt handling routine by means of this table.

An Interrupt Descriptor Table (IDT) is the protected mode equivalent of the IVT. For the purposes of this chapter, the two types of interrupt table are equivalent: each table is a 256-entry array that functions to direct interrupts to interrupt handling routines. The table used depends on whether the processor is in real mode or protected mode when the interrupt occurs. If the operating system does not support the use of both modes, only one or the other actually is present.

An interrupt handler is the code that receives control when an interrupt occurs. The entries in the IVT (or IDT) identify interrupt handlers.

An interrupt number is an integer in the range 0 to 255 that identifies the interrupt type (source of the interrupt). When an interrupt occurs, the hardware recognizes the interrupt type and the applicable interrupt number. The processor uses this number as an index into the IVT (or IDT).

Figure 36-1 shows the interrupt hierarchy. Each interrupt category includes one or more interrupt types.

The top-level categories are external interrupts and internal interrupts.

An external interrupt is an event triggered by a condition external to the processor. A peripheral device in need of service and a key pressed on the keyboard are examples of conditions that result in external interrupts. An external interrupt occurs asynchronously with the execution of the processor's instructions. It, therefore, can occur at an unpredictable time and usually is not related to the currently executing program.

A device interrupt is synonymous with an external interrupt. This is because an external interrupt results from an external device signal.

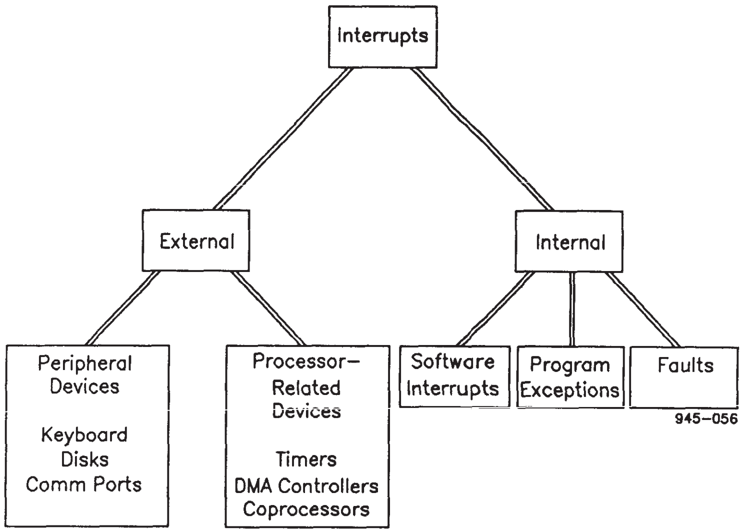


Figure 36-1. Interrupt Hierarchy

Note in Figure 36-1 that a programmable timer and a DMA controller are categorized as external devices (even if they are integrated into the processor chip). These devices are considered external to the processor because they operate asynchronously (in parallel to the processor's instruction stream). Floating-point coprocessors also are considered external devices for this same reason.

An internal interrupt is an immediate result of an instruction the processor tried to execute. Internal interrupts occur because instruction execution cannot, or should not, be allowed to proceed normally. An invalid opcode and an erroneous divide instruction are examples of conditions that result in internal interrupts.

Internal interrupts are unrelated to external events and have fewer conceptual implications. They can involve one process encountering an unexpected condition in a program, such as a divide by 0, or a deliberate process action, such as the explicit use of the INT instruction. In principle, an internal interrupt appears to be no different from a subroutine call.

EXTERNAL INTERRUPT HANDLING MODEL

An external interrupt generally is used to alert the processor to service an external device in a timely manner. Under CTOS/VM, external interrupts are managed by a general model that provides device handling and control over interrupt occurrence.

DEVICE HANDLING

Device handling is accomplished by a device handler program. Device handlers perform the hardware I/O to and from an external device. Handlers for some devices are included in the CTOS Kernel; others are part of system services or application programs.

Device handlers usually consist of a device handler process, which manages the device and initiates I/O, and a device interrupt handler, which executes when operations complete or status conditions change at the device. Figure 36-2 shows a typical device handler.

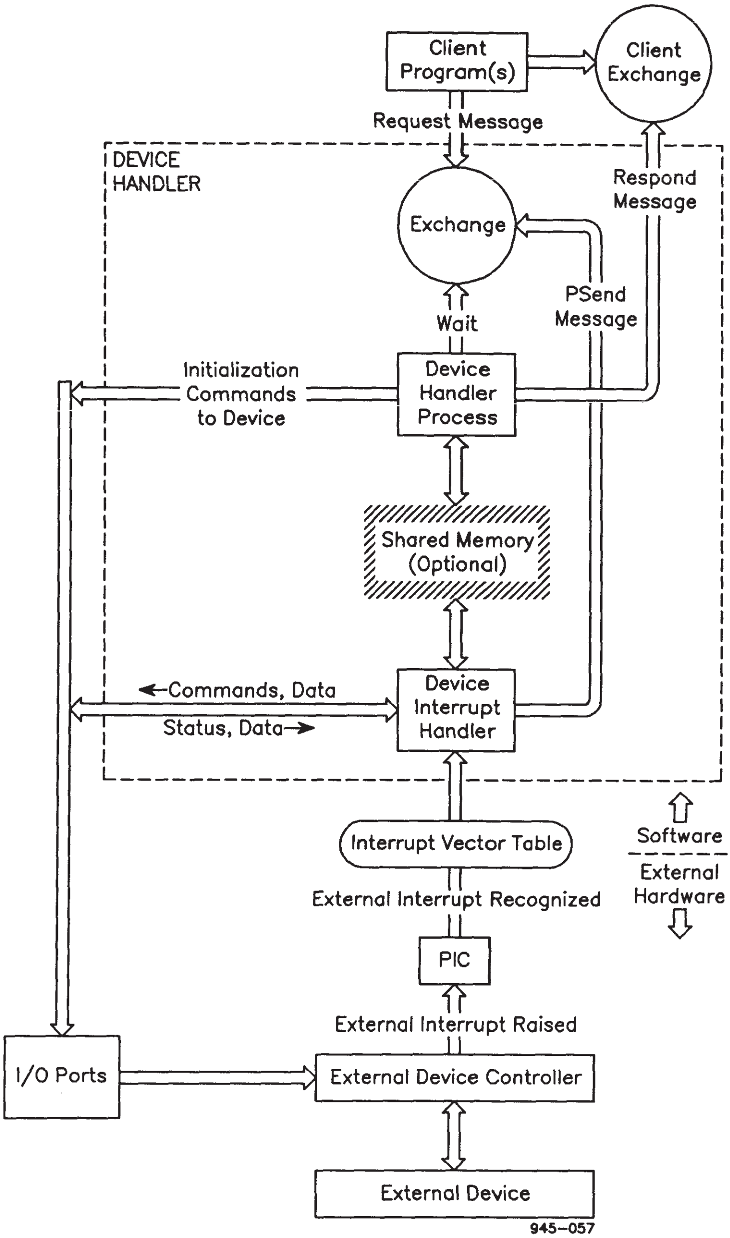


Figure 36-2. Device Handler

Although they execute asynchronously (as if they were two processes), the device handler process and the interrupt handler are two closely related parts of the same program. Communication and synchronization are accomplished by using the PSend Kernel primitive and, optionally, some shared memory, such as buffers and control information.

The device interrupt handler executes when the external interrupt occurs. If necessary, it may call PSend to start execution of the device handler process, which has been waiting at an exchange. PSend is effectively the only way the interrupt handier process and interrupt handler can synchronize. Only the device handler process (not the interrupt handler) may call the Kernel primitive Wait to wait at an exchange, so it is impossible for the device handler process to use PSend to send a message to the interrupt handler. Synchronization, therefore, is one-directional (from the interrupt handler to the process), although data communication can flow in either direction if shared memory is employed.

Device Handler Process

A typical device handler process spends most of its time idle. It waits at an exchange for either of two kinds of messages to reach it: commands from some program in the system that has work for the device, or messages (from its interrupt handler) that represent status or data from the device itself.

As such, the device handler process is both a clearing house for information related to the device and the agent responsible for determining what the device should do next. It is positioned between a client program using the device and the interrupt handler. (The interrupt handler, in turn, is positioned between the device handler process and the actual device.)

The device handler process does not run immediately when an interrupt occurs. It executes only if the interrupt handler sends it a message. Some interrupt handlers will send messages to their device handler processes each time an interrupt occurs; others do so only after a succession of interrupts have filled or emptied a data buffer. Devices that interrupt frequently enough can impede program performance to the extent that it would be prohibitively expensive to execute the device handler process after each interrupt. To maintain an acceptable performance level, the amount of work performed at each interrupt must be minimized. An an example, RS-232-C serial port devices cause frequent interrupts and, therefore, use an interrupt handler designed to optimize performance by avoiding the use of PSend on each interrupt. (For details, see "CRIHs and CMIHs," later in this chapter.)

Device Interrupt Handler

The device interrupt handler executes when an external interrupt occurs. It performs the following functions:

- as the primary responsibility, transfers data to or from the device or initializes DMA hardware that, in turn, performs such transfers
- checks error and status conditions in the device after each interrupt
- in some cases, processes data
- in some cases (such as with RS-232-C serial port handlers), performs low-level protocol functions
- decides when to start the device handler process executing (by calling PSend) to get further assistance

Because a device is prevented from causing additional interrupts while its interrupt handler is executing, the handler must service an interrupt expediently. Work that can be postponed (on input) or accomplished in advance (on output) should be performed by the device handler process, rather than the interrupt handler. Device handler processes typically can be interrupted, even by their own device interrupt handler, except during execution of critical code regions when the interrupt flag is turned off, disabling all external interrupts. As a result, device handler processes can take longer (than their interrupt handlers) to do their processing, without causing interrupts for the device to be lost. (See "Pending and Lost Interrupts," later in this chapter.) The interrupt handler often is programmed to buffer the I/O, effectively extending the time during which the device can transfer data before assistance from the device handler process is required.

CONTROLLING WHEN EXTERNAL INTERRUPTS OCCUR

An external interrupt can occur after any instruction the processor executes. External interrupts, however, can be controlled by the interrupt flag and the Programmable Interrupt Controller (PIC).

The Interrupt Flag

Most external interrupts are maskable, which means that the processor can prevent them from occurring. This type of interrupt control is used, for example, to prevent interrupts while critical code regions are executing. Masking an interrupt is accomplished by clearing the interrupt flag in the flag word (disabling interrupts). Maskable interrupts can occur only when this flag is set (enabling interrupts).

When an external interrupt occurs, the processor hardware disables interrupts automatically. Certain interrupt handler styles allow the operating system to enable interrupts again before executing the interrupt handler,- other styles keep interrupts disabled until the interrupt handler exits. (For details, see "CTOS/VM Interrupt Handler Styles," later in this chapter.)

The Programmable Interrupt Controller

The Programmable Interrupt Controller (PIC), a device closely associated with the CPU, extends interrupt enabling and disabling as implemented in the processor's interrupt flag to multiple levels. It can be viewed as a part of the processor's interrupt mechanism rather than a separate external device. (In some Intel microprocessors, the PIC is packaged as part of the same chip as the processor).

The PIC prioritizes interrupt signals from external interrupt generating devices and associates these sources with interrupt numbers. Each device is wired to the PIC at a separate PIC input pin associated with a priority. Thus, hardware design fixes device priority. [Note that nonmaskable interrupt (NMI) sources are the only type that is not wired to the PIC. For details, see "Nonmaskable Interrupts (NMI)," later in this chapter.]

Devices with less patience are given a higher priority. Patience is the amount of time that can safely elapse before an interrupt is serviced by its interrupt handler. As an example, a hard disk drive has infinite patience. It can revolve forever while waiting to service an interrupt; the only penalty is increased rotational delay. On the other hand, a keyboard controller would require that the interrupt handler empty a one-character hardware buffer of a typed character before the operator pressed another key. Otherwise, an overrun would occur, because the buffer could not hold an additional character.

PIC management typically is an operating system function. Certain interrupt handler styles, however, require that the programmer issue PIC commands.

By issuing PIC commands, devices can be masked selectively. When an external interrupt occurs, the PIC automatically masks interrupts from the device causing the interrupt and from any other lower priority devices. Another interrupt from the same source cannot occur until the interrupt handler exits (even in cases where the operating system enables interrupts again before executing the interrupt handler). When the interrupt handler is ready to exit, the device is unmasked by sending an end-of-interrupt (EOI) command to the PIC. In some interrupt handler styles, the operating system sends the EOI command/ in others, the command is sent by the user-written interrupt handler. (For details, see "CTOS/VM Interrupt Handler Styles," later in this chapter.)

If the processor interrupt flag is set, the PIC's selective masking can result in nesting of interrupt handlers for interrupts generated by devices of different priorities. Figure 36-3 shows an example of how this works. In the figure, the first interrupt (Int1) results in the PIC masking that priority level and all lower priority interrupt sources. When the first interrupt is followed by a second, higher priority interrupt (Int2), the PIC masks the higher priority level and all lower priority levels for the duration of the second interrupt handler's execution. At EOI for the second interrupt, the second masking is removed, but the first one remains in place until EOI occurs for the first interrupt.

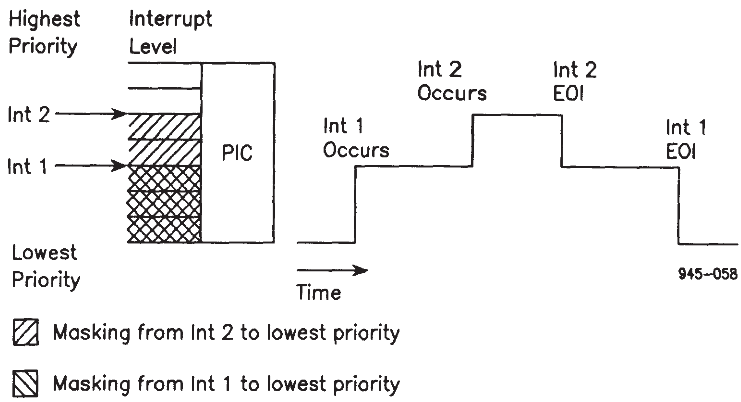


Figure 36-3. Interrupt Nesting

Device prioritization, therefore, ensures that devices may interrupt other device interrupt handlers with a lower priority, but not vice versa.

Pending and Lost Interrupts

If an event occurs that would cause an interrupt while that interrupt is masked or interrupts as a whole are disabled, that interrupt is not prevented entirely. It is merely deferred until it is enabled. Such an interrupt is called a pending interrupt.

Generally, one interrupt signal per device can be held pending at a time. If more than one interrupt signal occurs for the same device while the interrupt is disabled, only one of the interrupts ultimately occurs. Those that do not occur are lost interrupts. Such interrupts result in an overrun or underrun condition.

Most device controllers have special hardware to detect lost interrupts so that the device handler can report an I/O error. To prevent such errors, interrupts should only be masked for brief periods.

Nonmaskable Interrupts (NMIs)

A few external interrupt sources cause NMIs. An NMI will occur regardless of the state of the interrupt flag or the PIC. It always causes interrupt number 2, which is dedicated to servicing NMIs on Intel microprocessors. All other interrupt numbers are maskable when caused by an external source.

On CTOS/VM processors, NMI sources include memory parity errors and bus ready timeouts. (The latter are caused by addressing a nonexistent peripheral device, or by a device for which initialization is programmed erroneously.) Even these NMI sources can be masked by writing appropriate commands to specific external hardware that controls them. Thus, there actually are no nonmaskable external interrupts.

Internal interrupts are never maskable.

CTOS/VM INTERRUPT HANDLER STYLES

CTOS/VM supports two styles of interrupt handlers: raw and mediated. The two styles have different calling conventions and programming rules. They allow the programmer to decide between convenience or performance.

A raw interrupt handler offers performance over convenience in the following ways:

- The processor hardware transfers control directly to the user-written handler, which must be written in assembly language. (Coding in a high-level language defeats the purpose of writing the raw handler.)

- The handler must leave processor interrupts disabled. Because an RIH cannot be interrupted, nesting of interrupts cannot occur while it is executing.

Caution: In real mode, a raw interrupt handler executes on the stack of the currently running process. For this reason, a raw interrupt handler must carefully control its stack depth. A handler that uses in excess of 64 words of stack space can overwrite the memory of another process and cause system crashes with status code 22 ("Bus timeout"), 28 ("Invalid Opcode"), or 91 ("Operating system checksum error"). In protected mode, each interrupted process (interrupt task) has its own stack.* Stack overflow causes the system to crash with status code 92 ("Interrupt stack overflow").

Raw interrupt handlers are used for servicing high-speed, non-DMA devices.

*A process is called a task in the iAPX 286 Programmer's Reference Manual and the 80386 Programmer's Reference Manual.

A <u>mediated interrupt handler</u> provides convenience over performance in the following ways:

- It can be written in a high-level language as well as assembly language.
- It permits automatic nesting of interrupt handlers by priority since processor interrupts are enabled during its execution. This means that a mediated interrupt is designed to be interrupted, if necessary, by higher priority device interrupt handlers.
- The operating system performs part of the handler's work by saving and restoring all registers and performing some or all of the EOI processing.

Mediated interrupt handlers are recommended except for devices where interrupts occur so frequently that they would significantly impede program performance. Keyboard interrupts, for example, are serviced by mediated handlers.

There is also a difference in style between interrupt handlers for RS-23 2-C serial port communications devices and all other interrupt handlers. RS-232-C devices require special interrupt handling, because they cause frequent interrupts and, typically, several devices share an interrupt number. For these reasons and others, RS-23 2-C interrupt handlers have unique calling conventions and programming rules.

Differences in interrupt handling result in the four styles of CTOS/VM interrupt handler shown in Figure 36-4.

	RS-232-C serial port communications interrupt handler	Non RS-232-C interrupt handler
Raw interrupt handler	CRIH	RIH
Mediated interrupt handler	CMIH	MIH

945-059

Figure 36-4. Interrupt Handler Styles

Programmers concerned with RS-23 2-C devices should read the "Communications Programming" section in the CTOS Programmer's Guide as well as the next section.

CRIHs AND CMIHs

Figure 36-5 shows the program logic of a CRIH and a CMIH. The InitCommLine operation establishes the interrupt vector and the communications channel on that vector for the interrupt handler.

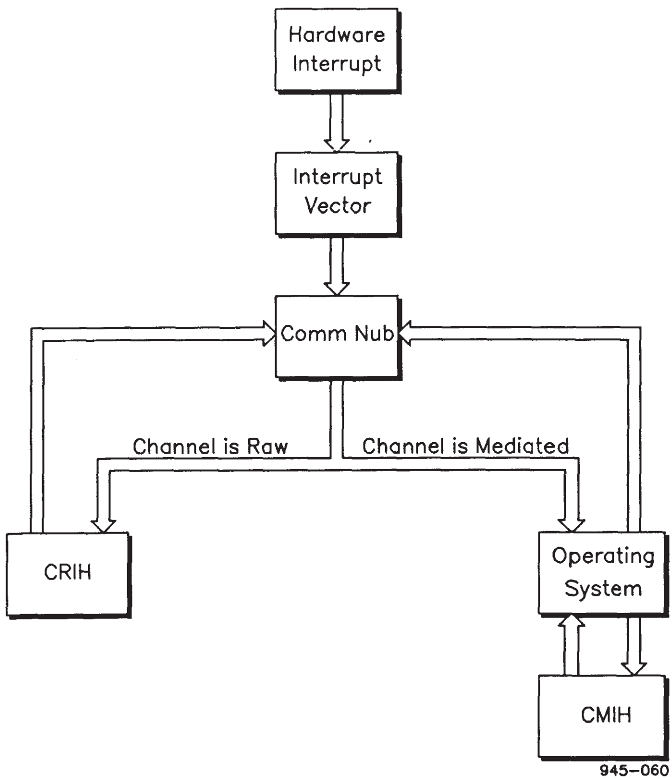


Figure 36-5. CRIHs and CMIHs

The Comm Nub shown in Figure 36-5 is a part of the operating system that dispatches CRIHs and CMIHs. A single hardware interrupt vector (PIC input pin) can support multiple communications channels belonging to different application programs. The Comm Nub directs the interrupt to its proper handler. It queries the serial controller's status to determine which channel is servicing the interrupt. Then, it determines whether the interrupt is a CRIH or a CMIH.

If the channel is serviced by

- A CRIH, the Comm Nub transfers control to the appropriate user-written CRIH. The user-written CRIH returns to the Comm Nub when it has completed processing the interrupt.
- A CMIH, the Comm Nub transfers control to the operating system, which in turn transfers control to the appropriate user-written CMIH. The user-written CMIH returns to the operating system, which then returns to the Comm Nub.

Guidelines for writing RS-232-C RIHs and RS-232-C MIHs are described in the following sections.

GUIDELINES FOR WRITING A CRIH

To write efficient CRIHs, observe the following guidelines:

1. Interrupts must remain disabled for the duration of the interrupt.

2. In real mode, all processing is done on the stack of whatever process happened to be running at the instant the interrupt was taken, unless the CRIH requests that the Comm Nub call PSend to activate the device handler process. In such a case, after the CRIH returns, the Comm Nub switches to the operating system's interrupt stack before calling PSend. (The interrupted process is not resumed immediately if the awakened device handler process has a higher priority.)

3. A CRIH should use PSend to activate a device handler process only when necessary (not on every interrupt). This is because PSend overhead (process scheduling and context switching) usually exceeds the overhead of the rest of the Comm Nub and the CRIH itself. (See "Device Handler Process," earlier in this chapter.)

   The CRIH should communicate with its device handler process only as much as required. Usually the receive CRIH has a multicharacter buffer that it fills, and the transmit CRIH has a buffer that it empties before the device handler process is dispatched.

   A typical error is to have the transmit CRIH activate the transmitting device handler process (which is waiting for buffer space) as soon as 1 byte of space is available. A better scheme is to have the transmit CRIH wait until the buffer is one-third to one-half empty. This avoids dispatching the transmitting process after each character sent, once the buffer is full.

4. Code a CRIH as tightly as possible. This code runs every time a character is sent or received: a few instructions can make a visible difference at a high baud rate or when multiple channels are in use simultaneously. Let the Comm Nub set up DS and BX so you can quickly locate the data structure needed to service the interrupt.

Figure 36-6 summarizes the guidelines for writing a CRIH.

---

1. All registers are saved for you.
2. No parameters are provided on the stack when the CRIH is called.
3. The Comm Nub sets DS to the selector of pDsBx and BX to the offset of pDsBx.
4. Interrupts are disabled upon entry and must remain disabled for the duration of the CRIH.
5. You do not do any of your own device controller or PIC EOI processing.
6. You may not do a PSend on your own. The Comm Nub can call MediateIntHandler followed by a call to PSend for you. You may cause a PSend upon exit from the CRIH by leaving a non-zero value in the AX register. If AX = 0, not PSend occurs. If AX <> 0, AX is used as an exchange and DS:BX as the memory address of the message (pMsg).
7. Exit using RET. All registers are restored for you automatically.

Figure 36-6. User-Written CRIH Summary

GUIDELINES FOR WRITING A CMIH

The CMIH is very similar to an MIH. (For details, see "Guidelines for Writing an MIH," later in this chapter.) The following are a few ways in which the CMIH differs:

- The InitCommLine operation is used to allocate the interrupt vector (rather than SetIntHandler, which is used by MIHs).
- The entry in the IVT (or IDT) does not direct the interrupt to the entry point of the CMIH. Instead, the interrupt is directed to the Comm Nub.
- In real mode, the Comm Nub switches control to the operating system's stack. (In protected mode, each interrupted process executes on its own stack.)
- The user-written CMIH can use both the ReadCommLineStatus operation and the WriteCommLineStatus operation, as well as the PSend, SetTimerInt, and the ResetTimerInt operations (used by MIHs).
- When the user-written CMIH is called, one parameter is supplied. The parameter, pDsBx, is user-defined but normally indicates which of the communications channels is being serviced by the CMIH.

Figure 36-7 summarizes the guidelines for writing a CMIH.

1. All registers are saved for you.

2. One 4 byte parameter is supplied on the stack when the CMIH is called: the pDsBx provided to InitCommLine.

3. DS is set to the selector of the pDsBx parameter of InitCommLine upon entry; SS does not match DS.

4. Interrupts are enabled during the CMIH: you may disable them briefly, if necessary.

5. You may use the SetTimerInt and ResetTimerInt operations.

6. You may do PSend(s) on your own.

7. You do not do any of your own device controller or PIC EOI processing. (This is performed by the Comm Nub.)

8. Exit using RET. All registers are restored for you automatically. The values you leave in AX and other registers when you exit do not matter.

Figure 36-7. User-Written CMIH Summary

RIHs and MIHs

Figure 36-8 shows the program logic of an RIH and an MIH. The SetIntHandler operation is used to allocate the interrupt vector.

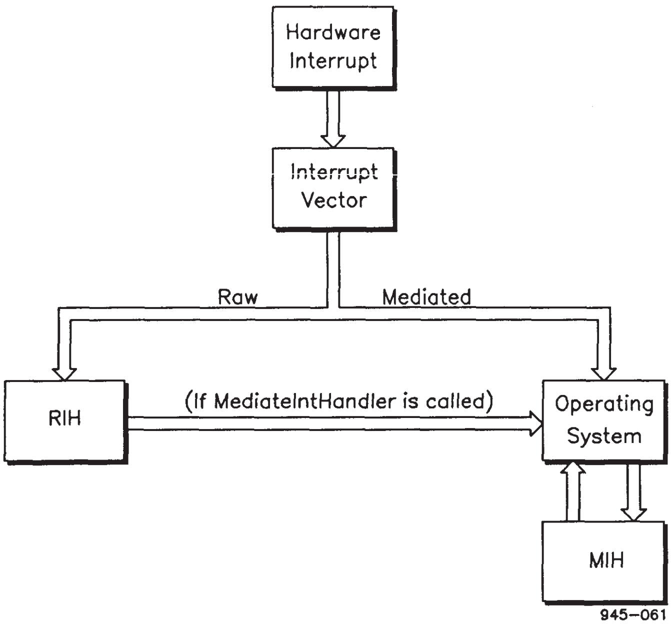


Figure 36-8. RIHs and MIHs

GUIDELINES FOR WRITING AN RIH

The RIH must conform to the following rules. When an interrupt occurs, the RIH does the following:

1. In real mode it saves any registers that will be used.

2. It handles the device that generated the interrupt and processes as necessary.

3. It issues an EOI command to the device controller (if required by the device) and also an EOI command to the master PIC.

4. In real mode, it restores the saved registers,

5. It uses the IRET instruction to reenable processor interrupts while returning to the point of interrupt. In protected mode, the JMP instruction must follow IRET to transfer control to the beginning of the RIH.

The only operation an RIH can use is MediateIntHandler. It is used to convert the RIH to an MIH if the RIH determines that the device handler process needs notification for some reason. (For details, see "Guidelines for Writing an MIH," later in this chapter.) In this case, the RIH does not perform steps 4 and 5, above.

Figure 36-9 summarizes the guidelines for writing an RIH.

1. In real mode, you must save all registers you use, because the only registers saved by the hardware are the flags, CS, and IP.

2. No parameters are supplied when the RIH is called. (The RIH must be able to execute independently of parameters.)

3. DS is set according to SetIntHandler upon entry; SS does not match DS.

4. Interrupts are disabled upon entry and must remain disabled for the duration of the RIH.

5. You must do all of your own device controller and PIC EOI processing.

6. You must not make any system calls except MediateIntHandler. If you call MediateIntHandler, your RIH becomes a MIH (which can call PSend, SetTimerInt, or ResteTimerInt).

7. Exit using IRET. (However, if you called MediateIntHandler, you must follow the MIH exit and termination procedures. For details, see "MIH," nesxt in this chapter.) In real mode, you must restore all registers you use before you exit or call MediateIntHandler.

8. In protected mode, IRET must be followed by a JMP to the beginning of the RIH.

Figure 36-9. User-Written RIH Summary

GUIDELINES FOR WRITING AN MIH

Figure 36-10 summarizes the guidelines for writing an MIH.

---

1. All registers are saved for you upon entry.
2. No parameters are supplied upon entry.
3. DS is set according to SetIntHandler upon entry; SS does not match DS.
4. Interrupts are enabled during the MIH: you may disable them briefly, if necessary.
5. You set the fDeviceInt flag in SetIntHandler to TRUE; you do not do PIIC EOI processing.*
6. You may use the SetTimerInt or ResetTimerInt operations.
7. You may do PSend(s) on your own.
8. Exit using RET. The values you leave in AX and other registers when you exit do not matter. All registers are restored for you automatically.

*Both the PIC and the interrupting device require an EOI command for successful completion of interrupt processing or the system will hang.

Figure 36-10. User-Written MIH Summary

EXAMPLES OF CTOS/VM EXTERNAL INTERRUPT HANDLERS

PARALLEL PORT INTERRUPT HANDLERS

The SetLpISR operation establishes the printer interrupt handler [also called a printer interrupt service routine (PISR)] to process interrupts generated by parallel printer port interfaces. (For details, see Chapter 16, "Parallel Port Management.")

PISRs can be linked to the System Image and declared at system build. Alternatively, they can be linked with a dynamically installed system service or an application program and declared through the use of the SetLpISR operation.

X-BUS INTERRUPT HANDLERS

Three levels of interrupt handlers are provided for X-Bus modules: XINT0, XINT1, and XINT4. XINT0 and XINT1 are nonshareable. XINT4 is shareable.

XINT0 And XINT1

XINT0 and XINT1 are for X-Bus modules that require a fast interrupt handler. The interrupt handler can be either raw or mediated by the operating system, as specified in the SetIntHandler operation.

Since the XINT0 and XINT1 interrupts are nonshareable, a system can be configured to have at most two modules that require these interrupts. X-Bus modules that require these fast interrupt levels must be able to use either, as instructed by software.

XINT4

XINT4 is set up for modules that can tolerate a slower latency.

This interrupt level is implemented by the Xbif system service as a chain of interrupt handlers that are invoked in a round-robin fashion whenever an XINT4 occurs.

Each interrupt handler is of type Boolean and returns FALSE (0h) or TRUE (0FFh) in the AL register of the microprocessor. TRUE is returned if the XINT4 was generated by the module that the interrupt handler services.

This protocol means that the interrupt handler for any X-Bus module must have a way to determine whether or not its module generated an interrupt.

The SetXbusMISR operation is used to establish an XINT4 multiplexed interrupt handler [also called a multiplexed interrupt service routine (MISR)]. Additionally, SetXbusMISR controls dedicated XINT1 interrupt handler allocation.

PSEUDOINTERRUPTS

A pseudointerrupt shares an interrupt vector among several application programs.

Pseudointerrupts are implemented in software rather than in hardware. In this sense, they are not really interrupts. However, they are similar to interrupts in that they result in an interrupt handler being executed.

An interrupt handler activated by a pseudointerrupt executes in the same environment and has the same responsibilities and privileges as an interrupt handler activated directly by a hardware interrupt.

The Programmable Interval Timer (PIT) uses a pseudointerrupt mechanism. The SetTimerInt operation establishes a PIT pseudointerrupt handler to service timer pseudointerrupts. (For details, see Chapter 33, "Timer Management.") Pseudointerrupts, in this case, allow each of several software routines to function as though it has exclusive use of the high-resolution PIT.

In a master, for example, the Cluster Line Protocol Handler, the 3270 Terminal Emulator, and a user-written device handler for realtime data acquisition equipment would need high-resolution interval timing concurrently. Each of the three pseudointerrupt handlers performs the same logical (but not device-dependent) processing as if it were servicing an external interrupt from the PIT itself.

The XBif system service uses the pseudointerrupt mechanism for interrupts generated by X-Bus modules. The XINT4 interrupt handler is implemented as a chain of interrupt handlers invoked in a round-robin fashion whenever an XINT4 interrupt occurs. (For details, see "X-Bus Interrupt Handlers," earlier in this chapter.) The QIC tape system service, CT-Net Ethernet media system service, and Telephone Server are examples of programs that use Xbif.

INTERNAL INTERRUPTS

An internal interrupt is caused by instruction execution. Depending upon the type of internal interrupt, the instruction may or may not have completed successfully.

There are three major types of internal interrupt: software interrupts, program exceptions, and faults.

(Internal interrupts are sometimes referred to as exceptions. See the iAPX 286 Programmer's Reference Manual and the 80386 Programmer's Reference Manual.)

SOFTWARE INTERRUPTS

A software interrupt is caused by the program explicitly using the INT instruction. In some operating systems (notably MS-DOS), this is the standard way to transfer control to the operating system to request services. A software interrupt is simply a specialized type of subroutine call: typically, different interrupt numbers correspond to different services, and arguments and results are passed in registers.

Application programs do not use software interrupts to request services of the operating system. However, some versions of CTOS make use of software interrupts internally, and software interrupts are used when MS-DOS is run under CTOS.

PROGRAM EXCEPTIONS

An program exception is the processor's response to an invalid instruction that cannot be executed. Program exceptions include

- divide error
- overflow (INTO instruction)
- bounds check
- invalid opcode

A program exception usually indicates a program error.

FAULTS

A fault occurs in protected mode only when the processor detects a condition that calls for operating system intervention. There may be nothing wrong with the instruction being executed. It fact, it is sometimes possible for the fault handler to resume execution of the program after attending to the condition that caused the fault.

Faults include

- general protection fault
- segment not present
- stack exception
- page fault

The segment not present fault is an example of a processor-supported fault used for segment swapping. The operating system arranges for application programs to cause segment not present faults when they attempt to access code segments currently not in memory. The fault signals the operating system to read the missing segment into memory. After the segment is read in, the program is resumed as if nothing had happened. (For details, see Chapter 36, "Virtual Code Management.")

It is possible to restart a program after most types of faults, because the instruction that caused the fault was not executed. The saved CS:IP is the memory address of that instruction, not the one after it. After servicing the fault, the operating system returns to the program (using the IRET instruction), and the instruction is restarted. Provided the faulting condition has been removed, program execution proceeds normally.

Because faults are potentially restartable, fault handlers are transparent to a program; program exceptions, on the other hand, generally are fatal. Note that certain kinds of general protection faults do not follow this rule, because a general protection fault indicates a program error.

Faults usually fall into the category of internal interrupts that are handled by the operating system. Fault handlers are rarely user-written.

TRAP HANDLERS

Program exceptions and software interrupts are often handled by system services or application programs. The usual way of creating the handler is to install a trap handler.

A trap handler is an interrupt handler that is in effect only for the program installing it. Other programs may install their own trap handlers, which perform different functions. A trap handler applies to all processes in the program.

As an example, some programming language run time packages include trap handlers to handle divide error program exceptions. When a program written in such a language causes a divide error, the run time package prints an error message or takes other action appropriate to the language.

The SetTrapHandler operation is used to establish a trap handler for the currently executing program in real or protected mode.

In protected mode, SetTrapHandler uses an 80286 trap gate. A trap gate is an interrupt structure in the IDT that references the interrupt handling procedure. The Set386TrapHandler operation establishes a local handler using an 80386 trap gate. An 80386 trap gate supports virtual 8086 mode.

In real mode, the trap handler of the last program that established a local handler remains in effect. Other programs must not use this handler, or unpredictable results will occur.

In protected mode, system default trap handlers exist for use by programs that do not establish their own local handlers. A system service may replace a system default handler by using the SetDefaultTrapHandler operation.

Not all internal interrupts are handled by trap handlers. As an example, fault handlers, like external interrupts, usually are installed by the operating system using SetIntHandler. This results in a handler that is in effect system-wide.

PACKAGING OF INTERRUPT HANDLERS

Additional interrupt handlers can be linked either with an application program or with a system service. The system service can be linked with the System Image at system build, or it can be dynamically installed.

The following operations are used to inform the operating system of the existence of an interrupt handler in an application program or in a dynamically installed system service:

- InitCommLine
- SetIntHandler
- SetDefaultTrapHandler
- Set386TrapHandler
- SetTrapHandler
- SetLpISR
- SetXbusMISR
- SetTimerInt

APPLICATION PROGRAM

Packaging an interrupt handler with an application program permits the interrupt handler to occupy memory only when the application program that needs it is in memory. Also, somewhat less effort is required to package the interrupt handler with an application program. Generally, an interrupt handler that is used only by one application program should be packaged with that program.

SYSTEM SERVICE

If an interrupt handler must be available continuously, even while one application program is being replaced with another, the interrupt handler must be packaged with a system service. An interrupt handler that supports a device attached to a master (on behalf of application programs executing in cluster workstations) must be packaged with a system service in the master (and also must use the formal Request/Respond model of interprocess communication). Packaging an interrupt handler with a system service reduces application program run file size, which would otherwise include the interrupt handler. Generally, an interrupt handler that is used by all or most application programs should be packaged with a system service.

OPERATIONS

The interrupt handler operations described below are presented alphabetically. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

InitCommLine — Establishes an interrupt vector and the communications channel on that vector for a CRIH or CMIH.

MediateIntHandler — Converts an RIH to an MIH.

PSend — Is a Kernel primitive that functions identically to the Send primitive but is used instead of Send in interrupt handlers.

ReadCommLineStatus — Can be used by a CRIH, CMIH, or an application process to query certain RS-232-C signals not defined in the serial communications controller.

ResetTimerInt — Can be used by a CMIH or MIH to terminate the Timer Pseudointerrupt Block (TPIB) initiated by a SetTimerInt call.

Set386TrapHandler — Establishes a trap handler for 80386 microprocessor-based systems using an 80386 trap gate. Set386TrapHandler is always raw and is part of the process context for all processes in a partition (as opposed to being system-wide).

SetDefaultTrapHandler
Establishes a system default trap handler in protected mode to handle program exceptions and software interrupts identified by the iTrap parameter. The system default handler is accessible by any user number that has not used SetTrapHandler to establish a local trap handler.

SetIntHandler — Establishes an RIH or MIH. Unlike SetTrapHandler, SetIntHandler disables swapping of the caller and is always in effect system-wide.

SetLdtrDS — Sets the Local Descriptor Table register (LDTR) and DS registers of the caller in protected mode. SetLdtrDS also updates the LDT field in the caller's Task State Segment so that the LDT selector is preserved across a task switch. (See the iAPX 286 Programmer's Reference Manual and the 80386 Programmer's Reference Manual.)

SetLpMISR — Establishes the printer interrupt handler to process interrupts generated by parallel printer port interfaces.

SetTimerInt — Can be used by a CMIH or MIH to establish a PIT pseudointerrupt handler.

SetTrapHandler Establishes a trap handler in real or protected mode. In protected mode, SetTrapHandler uses an 80286 trap gate. SetTrapHandler is always raw and is part of the process context for all processes in a partition (as opposed to being system-wide).

ResetXbusMISR Purges an interrupt handler previously established using SetXbusMISR.

SetXbusMISR Establishes an XINT4 multiplexed interrupt handler. SetXbusMISR also controls the allocation of dedicated XINT1 interrupt handlers.

WriteCommLineStatus
Can be used by a CRIH, CMIH, or an application process to raise or lower certain RS-232 signals not defined by the serial communications controller.

37 X-BUS MANAGEMENT

The intermodule, general-purpose expansion bus (X-Bus) management provides a high-speed bus for the interaction of various system modules with each other and with the workstation processor module.

X-BUS OVERVIEW

The X-Bus originates at the processor module. System modules are linked to the X-Bus to the right of the processor module as shown in Figure 37-1.

The system modules are linked to and interact with the workstation processor module by means of the X-Bus.

The X-Bus provides the necessary signals for

- memory and I/O transfer
- direct memory access (DMA)
- interrupt programming

(See the hardware manuals for details on the X-Bus.)

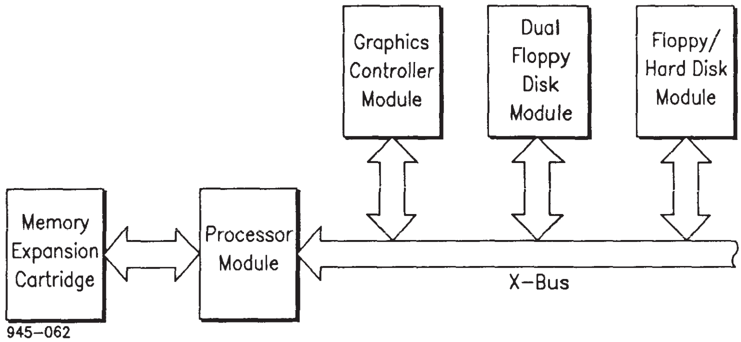


Figure 37-1. X-Bus Configuration

X-BUS MODULE IDs AND BASE I/O ADDRESSES

Each X-Bus module or input device for a workstation has a module or input device ID associated with it. The ID is a 16 bit number that uniquely identifies the module type. The bootstrap ROM in the main processor polls each module on the X-Bus and each device on the input device bus (I-Bus) and builds a table of IDs that describes the workstation hardware configuration.

The bootstrap ROM also assigns blocks of base I/O addresses to each module in the following way:

- 100h through 1FFh are for the first module on the right of the processor.

- 200h through 2FFh are for the next module, and so on.

The base I/O address is used for I/O access to the module.

Once a program has identified a particular module from the module's ID, that module's base I/O address can be computed from the position of the module on the X-Bus using the formula

base I/O address=100h*(position-1)

where position is 1 for the processor module, 2 for the next module to the right, and so on.

If a system service, for example, is installed to interface with a controller on the X-Bus, the system service first determines if the right type of controller is present. To do this, the system service calls the QueryModulePosition operation. QueryModulePosition calls GetModuleID repeatedly, incrementing the X-Bus position value by 1 each time GetModuleID is called until either the chosen ID is located or status code 35 ("No such module") is returned. If the module is located successfully, the system service then can compute its base I/O address.

X-BUS MODULE/PROCESSOR MEMORY ACCESS

There are three memory usage classes of X-Bus modules: master, slave, and master/slave.

An X-Bus memory master is a device that can access the processor RAM, but the processor cannot access the module's memory address space.

An X-Bus memory slave is a device that cannot access the processor RAM, but the processor can access the module's memory address space.

An X-Bus memory master/slave is a device that can access the processor RAM and in which the processor can access the module's memory address space.

ACCESSING X-BUS MODULE MEMORY

In most cases, the application programmer will be concerned with accessing X-Bus module memory. As an example, if you are writing a program that will manipulate the pixels of a bit map workstation video display, some of your program instructions will require manipulation of the Graphics Controller memory.

Using X-Bus Operations to Access Module Memory

To access X-Bus module memory, your program must call the MapXBusWindowLarge operation, specifying the module and the amount of module memory needed to be accessed in that module. MapXBusWindowLarge returns the memory address(es) of the required number of contiguous, 64K-byte segments.

MapXBusWindowLarge must be called at least once before the program attempts to access the module's memory. It must by called again if the program accesses a different module.

MapXBusWindowLarge is compatible in real mode and in protected mode. There are, however, a few differences that you need to be aware of. These are described in the following sections later in this chapter:

- "Specifying a Window Size"
- "Accessing Modules in Protected Mode"
- "Accessing Modules in Real Mode"

MapXBusWindow is an older operation that performs the same function of providing access to X-Bus module memory. It returns the address of only one 64K byte segment, however. Because MapXBusWindow can result in programs that are not compatible in protected mode, it is recommended that you use MapXBusWindowLarge for all new programs.

Specifying a Window Size

Each X-Bus module that contains memory accessible by the processor must have an X-Bus window entry in the system generation prefix files. (For details, see the CTOS System Administrator's Guide.) The window may be 480K, 224K, or 96K bytes. At system initialization, the operating system determines the X-Bus window size of each X-Bus module.

For real mode, the operating system reserves a region of addresses at the end of the 1 megabyte processor address space at system initialization. The size of this region is the maximum X-Bus window size of all X-Bus modules attached to the workstation.

ACCESSING MODULES IN PROTECTED MODE

Calling MapXBusWindowLarge in protected mode allows your program to access an X-Bus module's memory, as described earlier in "Accessing X-Bus Module Memory."

From the viewpoint of the programmer, protected mode implementation of MapXBusWindowLarge is totally transparent. MapXBusWindowLarge returns selectors (SLs) for the amount of memory that your program specifies based on the sWindow parameter. Because protected mode provides a 16 megabyte address space (or greater), it can accommodate mapping of X-Bus module memory to addressable memory regions above the first megabyte without the use of the extended address register (EAR), described next in this chapter.

ACCESSING MODULES IN REAL MODE

In real mode, calling MapXBusWindowLarge to access X-Bus module memory requires that the processor set up an extended address register (EAR). The EAR is used to map a portion of the main processor's address space into the X-Bus memory address space instead of its own (and therefore decreases the address space of the processor). The real mode processor generates a 20 bit address (1 megabyte address space). To this address, the EAR adds an extra 4 bits. The X-Bus module is programmed to respond to this 24 bit address. Each module responds to a different 1 megabyte base address range out of the total 16 megabyte range, depending on its position in the X-Bus.

(For details on the EAR, see the hardware manual for your processor module.)

From the viewpoint of the programmer, real mode implementation of MapXBusWindowLarge reduces the address space available to the processor by the size of the largest memory window in the system. If, for example, the module with the largest window has a 480K byte window, the maximum memory available is 512K bytes. Additional memory beyond 512K bytes is invisible to the processor, as memory addresses between 80000h and F8000h are mapped to the X-Bus.

X-BUS DMA

A DMA controller in the processor module controls the transfer of data over the X-Bus from a memory master or master/slave to the main processor's memory.

All X-Bus memory master or master/slave modules other than disk and graphics devices use channel 1, mode 3 DMA (verify mode) when accessing the main processor's memory.

This arrangement is required for operation with CTOS/VM and other Convergent X-Bus modules. The operating system initializes channel 1 DMA in this mode on powerup. Channel 0 is used by communications and channel 3 by the hard disk.

COMMUNICATION AND START-UP PROTOCOLS

An X-Bus module may communicate with a program on the processor module through its I/O space and/or by using memory either in the module's address space or in the processor's address space.

If the communication is through I/O space or through a structure in the module's memory address space, additional programming steps are necessary to set up the communication.

If the communication is through a memory structure in the processor address space, the module must be informed of the structure address, as such structures cannot be at fixed memory locations.

The communications structure location can be given to the X-Bus module either by using the module I/O space, or by using a protocol that uses the X-Bus Initialization Structure (XBIS), described next.

XBIS

The XBIF System Service provides a standard way that intelligent modules can use to establish communication with software running on the processor.

The XBIS, a 16 byte structure at memory location 400h, provides an area in the main processor's memory in which a program can communicate with a memory master module. In general, the program

- reserves the XBIS using the LockXBIS operation
- initializes the memory master module
- frees the XBIS structure using the UnlockXBIS operation

This general procedure is exemplified below using Voice/Data services and the Voice Processor module.

The Voice Processor module uses a private data structure for communication. To establish communication, Voice/Data services

1. call LockXBIS

2. place the Voice Processor module number in the XBIS data structure using the memory address returned from LockXBIS

3. place the physical memory address (obtained by using PaFromP) of the Voice Processor's private data structure in the XBIS data structure

4. write a value to the Voice Processor module base I/O address space (using GetModuleID to obtain the base I/O address)

This causes the Voice Processor module to read location 400h to

1. obtain the address of the private data structure

2. write a status byte to location 400h

3. interrupt the CPU in the main processor module

Voice/Data services then free the XBIS with UnlockXBIS.

X-BUS INTERRUPTS

Three interrupt levels are provided for X-Bus modules: XINT0, XINT1, and XINT4. (For details, see Chapter 36, "Interrupt Handlers.")

OPERATIONS

The X-Bus management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

QueryModulePosition
: Determines the bus position of a module. The X-Bus or input device bus (I-Bus), type code, and module number (if there is more than one module of the same type) are specified and the position is returned.

GetModuleID
: Provides access to the workstation module identification tables that are constructed by the boot ROM for the X-Bus and the I-Bus.

MapXBusWindowLarge
: Returns memory addresses for accessing the memory within an X-Bus module.

SwapXBusEAR
: Returns the word value that was previously written to the EAR. Any program using this operation is responsible for restoring the previous value when finished accessing X-Bus memory. (SwapXBusEAR is needed in real mode only.)

SetXbusMISR
: Establishes an XINT4 multiplexed interrupt handler.

ResetXbusMISR — Purges a previously established interrupt handler using SetXbusMISR.

LockXbis — Reserves the XBIS structure at location 400h. LockXbis returns the memory address 400h.

UnlockXbis — Frees the XBIS structure for use by other programs.

Mode3DmaReload — Sets up and programs DMA to and from X-Bus memory master devices using X-Bus mode 3 DMA.

MapXBusWindow — Is the same as MapXBusWindowLarge. For protected mode compatibility, MapXBusWindowLarge should be used in all new programs.

38 CONFIGURATION MANAGEMENT

Configuration management instructs you in setting up the operating system.

The following references describe system administrative actions primarily:

- Chapter 39, "Cluster Management," describes cluster configurations and how the cluster works. It includes the cluster operations used in exercising administrative control over the cluster.

- The CTOS System Administrator's Guide lists and describes the files that you need to create a bootable volume.

The following references involve programmer actions to reconfigure the operating system (rather than administrative actions):

- Chapter 40, "Native Language Support," presents ways you can nationalize programs using the Native Language Support (NLS) tables. It also describes message files.

- The CTOS System Administrator's Guide and the operating system Release Notice describe generating a system (SysGen). SysGen consists of changing the default parameter values and/or removing functionality to build a customized operating system version.

39 CLUSTER MANAGEMENT

Cluster management enables communication among cluster workstations and the master with which it is connected.

The master can be a master workstation or a Shared Resource Processor (SRP).

CLUSTER ENVIRONMENT

One high-speed, RS-422 channel is standard on each workstation. In cluster configurations connected to a master workstation, the master and all of the workstations connected to it use this channel for intercluster communications. For large clusters with an SRP master, multiple RS-422 channels are provided.

Each RS-422 channel is called a line and has a number associated with it. In a cluster configuration connected to a master workstation, there is one line, line 0. SRP clusters, however, have two lines per Cluster Processor board installed. The first board has line 1 and line 2, the second has line 3 and line 4, and so on. (There is no line 0 for an SRP cluster.)

The RS-422 channel operates at either 307K bytes or 1.8 megabytes. (See the CTOS System Administrator's Guide for details on configuring cluster line speed.)

STATUS

The master keeps statistics about errors and normal operational parameters. The GetClusterStatus operation makes these statistics available to any program at any workstation.

The GetClusterStatus operation should be used instead of GetWsUserName to obtain the same as well as additional information about user statistics.

POLLING

The master uses a technique called polling to check workstations that seek to use the RS-422 line for intercluster communications.

Polling starts every 1/20 of a second (one poll cycle) when a timer interrupt goes off, or whenever a response is ready to be returned to the workstation that initiated the request.

This method of polling

- guarantees that a workstation will be polled at least once every 1/20 of a second when the cluster is not busy
- polls active workstations more often than those not active

ROLL CALL

The master takes roll call by sending a message to each workstation that is currently online. If a workstation has a request to send to the master, it sends the message at this time; otherwise it informs the master that it has nothing to send.

The master notes which workstations had data to send during roll call.

REPOLL

When the master gets to the end of the list of workstations it is polling, it checks to see if there is any time left in the poll cycle.

If there is time left, the master polls each workstation that was active again.

Repoll is repeated until a new poll cycle starts or no workstations were active in the last poll.

Polling is totally transparent to the programmer or workstation user. It appears, however, that the programmer/user is in control.

REQUEST ROUTING ACROSS THE CLUSTER

Request routing depends upon how the request is defined and where the system service is installed. For further information, see

- Chapter 29, "Interprocess Communication," for request routing by file handle and file specification
- Chapter 30, "Inter-CPU Communication," for request routing between processor boards on the SRP
- Chapter 31, "System Services Management," for defining requests to be used with user-written system services

If you follow the conventions for routing requests described in the chapters above, not only will routing work correctly at your workstation, it will also work across the cluster or CT-Net.

To have a request served locally, you must install the system service at your workstation, or status code 33 ("Service not available") is returned.

The cluster operations described at the end of this chapter are used only in programs, such as the **Cluster Status** Utility (described in the CTOS System Administrator's Guide), to exercise administrative control over the cluster.

A request, such as Read or Write to a file server in a cluster, uses the interprocess communication (IPC) request/response model. (For details, see Chapter 29, Interprocess Communication.") The same requests are used throughout the cluster. (Note that they differ only in the way routing is defined for them.) For this reason, there are no explicit operations in this chapter for communication over the cluster.

OPERATIONS

The cluster management operations are described below. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations," for a complete description of each operation.)

GetClusterStatus
: Returns usage statistics for each communications channel and the workstations attached to it.

QueryWsNum
: Returns the number of the cluster workstation. QueryWsNum returns 0 if executed on a standalone workstation.

DisableCluster
: Allows an application program on the master workstation to disable polling of the cluster workstations after a specified time period. DisableCluster is also used to resume polling of the cluster workstations.

MegaFrameDisableCluster
: Allows an application program on the SRP to disable polling of all cluster workstations (except those on the line specified to stay up) after a specified time period. MegaFrameDisableCluster also is used to resume polling of cluster workstations on the SRP.

40 NATIVE LANGUAGE SUPPORT

Native Language Support (NLS) supports internationalization and nationalization of software. Additionally, NLS operations can be used by application programs to store messages in a separate message file.

Internationalization means language independence. Source code is internationalized when it is written in such a way that the resulting program can run in different languages without modifying the run file itself.

A language definition includes those requirements of a language that are unique to that language. French is different from German or English in ways that are obvious. Examples of other language requirements include currency symbols, such as the English pound sign or the U.S. dollar, and date/ time formats with various arrangements for the month, day, and year.

Nationalization results in software that runs using a single language definition.

External modifications can be made to an operating system so that the resulting system is nationalized.

Application programs typically display messages to the screen in a particular language. Operations are provided that allow messages to be removed from an application and instead be placed in a message file. The resulting program code remains language-independent. Other features of the message file itself allow for flexibility in displaying messages in different ways.

INTERNATIONALIZATION

To provide for internationalization, NLS includes a set of NLS tables.

The NLS tables control a number of different internationalizable aspects of software. Included among the tables, for example, are an uppercase to lowercase characters table, a date/time formats table, and a symbols table for number and currency. The NLS tables allow you to nationalize operating systems in different ways.

The NLS tables are in the Standard Software source file, [Sys]<Sys>Nls.asm.

As shipped, Nls.asm defines the proper tables for the United States. Using these tables, you can

- make any changes you want to Nls.asm
- assemble Nls.asm to create Nls.obj
- link Nls.obj to create the NLS configuration file [Sys]<Sys>Nls.sys

(See the Nls.asm file as released with Standard Software for details.)

When the operating system is bootstrapped, it searches for [Sys]<Sys>Nls.sys. If present, the operating system loads the contents of this file into memory, making these tables available to application programs by means of programmatic calls.

THE NLS TABLES

Table 40-1 shows the NLS tables contained in Nls.sys.

Table 40-1
NLS TABLES

Table Name	Code	Signature	Size (bytes)*
Keyboard Mapping	0	KE	Variable
File System Case†	1	FS	258
Lowercase to Uppercase†	2	XT	258
Video Byte Streams	3	VS	Variable (166 max.)
Uppercase to Lowercase†	4	LW	258
Keycap Legends	5	KC	Variable
Date and Time Formats	6	DT	Variable
Number and Currency Formats	7	NC	9 to 11
Date Name Translations	8	NT	Variable
Collating Sequence	9	CT	Variable
Character Class†	10	CC	258
Yes or No Strings	11	YN	Variable

* Includes the 2 byte signature.
† Is an n-element array; n = size (bytes)

Each of the NLS tables begins with a two-character (2 byte) signature to ensure validity of the table. The data for the table follows immediately thereafter.

When the operating system loads the NLS tables at boot time, it verifies that the signatures of the tables it knows about (0 through 11) are correct. Other tables can be added if desired.

If the table address is obtained by using the table code (ID code), the address returned is the signature address.

The NLS operations provided in the standard operating system library, Ctos.lib, give your programs access to the functionality of the NLS tables. (These operations are listed as utility operations in "Operations," at the end of this chapter.)

The NLS operations provide a layer of software that returns nationalized results depending on the values provided in the NLS tables.

In most cases, the programmer does not need to know the actual structure of any of the tables. If an application program is written using the appropriate NLS operations, proper results are returned to the program for French, if there is a French Nls.sys, or German, if there is a German Nls.sys. If there is no Nls.sys file, all of the operations return the U.S. standard.

The NLS tables contain the selections for all of the NLS options except for fonts and message text. (Messages are described in "Message File Creation," later in this chapter.)

NLS TABLE DESCRIPTIONS

KEYBOARD MAPPING

The Keyboard Mapping table is used to map keys pressed by the user to their character codes. If no table is present in the configuration file, the keyboard table as defined during system generation (SysGen) is used. The memory address of the Keyboard Encoding table is altered to reflect the address of the NLS Keyboard Mapping table, if present. The formats of the Keyboard Encoding table and the NLS Keyboard Mapping table are the same, ensuring backward compatibility. (See Chapter 10, "Keyboard Management," for details on keyboard mapping.)

The basic NLS Keyboard Mapping table (that is, no diacritical key handling) is 386 bytes: the signature is 2 bytes, and the table is 384.

A diacritical key handling portion of the table is provided for displaying characters with diacritical marks, such as the German **a** with an umlaut. The first key of a diacritical key pair enables diacritical mode; the second key displays the diacritical result.

The length of the diacritical key handling portion of the table can vary. It is determined by the following:

- the total count of diacritical keys (2 bytes)
- the diacritical key sequences [diacritic key pairs and their resultant values (3 bytes for each sequence)]

The Nls.asm file provides an example of the diacritical portion of the Keyboard Mapping table. The example shows how to edit the table to assign diacritical control to any keys you choose.

The total length of the Keyboard Mapping table is variable. In practice, however, the table will not ever be much larger than 400 bytes.

FILE SYSTEM CASE

The File System Case table is an optional table used by the file system for case-insensitive comparison and hashing. If the table is not present, lowercase roman letters are mapped to uppercase roman letters. No other characters are mapped.

It is recommended that you use only one such table definition on all systems because problems can occur if you interchange file systems, such as floppy disks, between systems with different language definitions.

LOWERCASE TO UPPERCASE

The Lowercase to Uppercase table is used by NlsCase for case-insensitive comparisons and by other application programs, which must force a conversion of case. The Document Designer **Replace** command, for example, allows replacement control. If your application program requires collation, you should use the NlsCollate operation rather than using this table.

VIDEO BYTE STREAMS TEXT

The Video Byte Streams Text table is used by video byte streams to allow translation of the prompts

Press NEXT PAGE or SCROLL UP to continue

and

Press NEXT PAGE to continue

which are displayed from within video byte streams. Each string should be 80 bytes or fewer.

This table should need to be accessed only by video byte streams.

UPPERCASE TO LOWERCASE

The Uppercase to Lowercase table is used by programs, such as NlsCase and NlsULCMPB, which must force a conversion of case.

This table can be used by programmers to translate characters.

KEY CAP LEGENDS

GetNlsKeycapText uses the Key Cap Legends table to specify the text strings to be displayed by programs when making reference to any of the key caps that commonly contain legends.

(For a description of the key cap values, see the GetNlsKeycapText operation in Chapter 3, "Operations," in the CTOS/VM Reference Manual.)

Part or all of the key cap text may be translated. The maximum size allowed is 15 bytes of text (plus the 1 byte text string length). Any character codes can be used within the key cap names. (Character codes are described in Chapter 10, "Keyboard Management.") It is recommended, however, that your program continue using the convention of displaying all uppercase letters for key cap text.

DATE AND TIME FORMATS

The Date and Time Formats table is used to specify format templates to control date and time construction. This table allows variations of date and time by country and by application program.

The format strings serve as templates for the NlsStdFormatDateTime operation. This NLS operation substitutes the actual date and time for control letters embedded in the format strings. (For details, see Appendix G, "NLS Templates," in the CTOS/VM Reference Manual.) Ten control letters denote the various types of information used to construct the resultant date and time string. The control letters are listed and defined in Appendix G.

Control letter order is significant. For example, whichever control letter appears first in the list is used to select am, pm, noon, or midnight. Table entries should be the lowercase version of the intended letter.

NUMBER AND CURRENCY FORMATS

The Number and Currency Formats table is used by GetNlsDateName and NlsParseTime to control the formatting of numbers and currency fields.

Key elements in the table are presented in Table 40-2.

Table 40-2

NUMBER AND CURRENCY FORMATS KEY ELEMENTS
(Page 1 of 2)

Element	Description
DecimalCh	A single ASCII character, either 2Ch (,) or 2Eh (.), used to indicate the decimal point in numbers. The default is 2Eh (.).
TriadCh	A single ASCII character, either 2Ch (,), 2Eh (.), or 20h (space), used to indicate the separation of numbers into triads (that is, thousands, millions, and so on). The default is 2Ch (,) . Note that use of space is not fully supported at this time, and thus its use may be ignored by some programs, or it may cause substitution of one of the other characters.
fFirstTriad	A flag that controls the rules for placing the triad character in the thousands position. If TRUE, the triad separator in the thousands position always appears when the number contains four or more digits to the left of the decimal. If FALSE, the thousands triad separator is suppressed when no more than one additional digit appears to the left. This notation is commonly used in France. The default is TRUE.

Table 40-2

NUMBER AND CURRENCY FORMATS KEY ELEMENTS

(Page 2 of 2)

Element	Description
	Note that some application programs never use triad characters, and others use them selectively or optionally. This flag merely controls the formatting when the program is using triad characters.
ListSepCh	A single ASCII character, either 2Ch (,) or 3Bh (;), used to indicate the separation of numbers within a list. The default is 2Ch (,). Note that this specification is used only by application programs that would otherwise have a conflict with the use of 2Ch (,) as the decimal point character.
iCurrency-Pos	A value to control the position of the currency symbol. Zero (0) indicates the leading currency symbol; 1 indicates the trailing currency symbol. Other values are reserved for future expansion. Note that embedded currency symbols, such as 5$33 to indicate $5.33, are not currently supported.
sbCurrency-Symbol	A string of up to 4 bytes containing the currency symbol. The first byte is the length of the string; the remaining 1 to 3 bytes contain the currency symbol.

DATE NAME TRANSLATIONS

The Date Name Translations table is used by GetNlsDateName and NlsParseTime to translate names of the months and days of the week.

More than one set of names can be defined. The format templates given to NlsStdFormatDateTime allow selection of which set of date names to use.

The maximum length of the date name string is 20 bytes.

(For a description of the index values used to reference the date names, see the GetNlsDateName operation in Chapter 3, "Operations," in the CTOS/VM Reference Manual.) Names in this table should be lowercase. The format templates can be used to control selective conversion to uppercase.

COLLATING SEQUENCE

The Collating Sequence table actually consists of four tables used by the NlsCollate operation.

The first table uses a simple substitution of character codes. This allows for reordering of the sort order including the mapping of uppercase and lowercase letters onto the same code values.

A second table defines 2-for-1 substitutions. Examples are the German "ß" which collates as ss and a with an umlaut, which collates as AU.

A third table defines of 1-for-2 substitutions. For example, the Spanish ch collates after c.

The last (256 byte) table determines character priority. This table is used only when all prior tests have resulted in equality. This table is sometimes used for case differences, such as collating lowercase after uppercase only when otherwise equal; however, it is used more commonly for accent mark priorities. The vowel e, for example, is considered equal in all its forms except for priority, which alternates between uppercase and lowercase versions, first with no accents, then with acute, grave, circumflex, and umlaut.

CHARACTER CLASS

The Character Class table is used by NlsClass to indicate the class of the character with the corresponding code.

Possible classes and their code values are as follows:

Class	**Code**
Numeric	0
Alphabetic	1
Special	2
Graphic	3
Blind	4

Graphic indicates that the character is used for line drawing or other special graphic purposes. Blind means that the character is not generally intended for display purposes.

YES OR NO STRINGS

The Yes or No Strings table is actually two tables used by the NLS operations, NlsYesOrNo and NlsYesNoOrBlank. One table is a list of words meaning "yes" in a particular language; the other table is a list of words meaning "no."

NLS OPERATIONS

Table 40-3 summarizes what NLS operations are available to provide the functionality of each of the NLS tables:

Table 40-3

NLS OPERATION SUMMARY

Table	Operation(s)
Keyboard Mapping	None (used by operating system keyboard process)
File System Case	None (used by operating system file system)
Lowercase to Uppercase	NlsCase
Video Byte Streams	None (used by video byte streams)
Uppercase to Lowercase	NlsCase ULCMPB (or NlsULCMPB)
Key Cap Legends	GetNlsKeyCapText
Date and Time Formats	NlsStdFormatDateTime
Number and Currency	Formats NlsNumberAndCurrency
Date Name Translations	GetNlsDateName NlsParseTime
Collating Sequence	NlsCollate
Character Class	NlsClass
Yes or No Strings	NlsYesOrNo NlsYesNoOrBlank

INTERNATIONALIZING APPLICATION PROGRAMS

This section discusses the internationalization of existing or new application programs.

Certain NLS operations exist for which there are equivalents in the standard operating system library, CTOS.lib. To internationalize your programs so that they work on any nationalized operating system, you must use the operations for internationalizable programs shown in Table 40-4. The equivalent operations are provided only for the purpose of programs that depend on using them.

Table 40-4

OPERATIONS FOR INTERNATIONALIZABLE PROGRAMS

Recommended Operation(s)	Equivalent Operation(s)
ULCMPB NlsCollate (for a richer set of rules)	NlsULCMPB
NlsStdFormatDateTime	NlsFormatDateTime FormatDateTime

EXISTING PROGRAMS

If you want to internationalize an existing program, you need to identify those operations for which there are internationalizable versions, as shown in Table 40-4. The effort that is required to make these changes is reasonably minor when compared to the benefits attained by being able to run the program on any nationalized operating system.

NEW PROGRAMS

If you are considering writing a new application program, use the internationalizable versions as specified in Table 40-4.

QUERYING THE NLS TABLES

The NLS operations allow you to query the NLS tables. The first parameter to each NLS operation is pNlsTableArea. This is the memory address of the NLS tables to be used for the call.

Typically, you want to use the NLS tables loaded at boot time. These tables can be used by all programs in the system. They are located in a system-common NLS table area.

The easiest way to access the system-common NLS table area is to pass NIL as the value of pNlsTableArea. NIL is interpreted as the beginning address of this area.

You could alternately pass the memory address of a different set of NLS tables that you created and linked with your application program. This might be useful if you want to have a single program that works correctly in two or more languages at the same time.

NATIONALIZATION

Nationalization allows operating systems to reflect a particular language definition.

You can modify NLS.asm to reflect the requirements of a particular language. To do this, you must change the applicable table(s) in the source file, NLS.asm, to meet your requirements before you assemble and link to create [Sys]<Sys>Nls.sys. (For details on the Nls.asm file, see "Internationalization," earlier in this chapter.) Thereafter, the operating system is nationalized for the modifications you made.

The Keyboard Mapping table, for example, can be changed to reflect the key positions of a French AZERTY or an English QWERTY keyboard. The Date and Time Formats table can show a date/month/year arrangement rather than a month/day/year as the standard representation.

You can also use the NLS tables selectively. If, for example, the only nationalization requirement is to change the currency symbol from the U.S. dollar sign to the English pound sign, you would include only the Number and Currency Formats table in the NLS table area. This eliminates the work of including unnecessary tables and saves operating system memory.

In addition, you can link additional sets of nationalized NLS tables with your application program. This might be useful if you want to have a single application that would work correctly in two or more languages at the same time.

MESSAGE FILE CREATION

Messages can be removed from an application and instead be placed in a message file. The resulting program code remains language-independent.

To take advantage of message files, use the message operations in CTOS.lib. (For a list of the message operations, see "Operations," at the end of this chapter.) The message operations retrieve the messages from the file.

Message files eliminate linking strings with your program. You can nationalize program strings simply by editing the message file.

The message file actually exists in two forms: text and binary. The text form is designed to be human-readable and consists of entries of the form

```
<colon>number<colon> <delim>TestString<delim>
```

for example,

```
:2000: "This is a sample text message."
```

By convention, a text file has the name PackageMsg.txt. Once the text file has been created, it must be converted to a binary form to be used by the program.

To convert to a binary file, use the **Create Message File** in the Executive. (See the Executive Manual for details.) Fill out the command form as shown below:

```
Create Message File

   Text file        PackageMsg.txt

   [Message file]   ______________________
```

By convention, the name of the binary file is the same name as the text file, except that the .txt extension is replaced with .bin. This is the default of **Create Message File**.

The binary file created above is thus

PackageMsg.bin

USING MESSAGE FILES

There are two ways to use message files. Application programs that may have a large number of messages can use macros within messages. Alternately, a separate set of message operations can be used for system services or application programs that may need fewer messages.

Macros

To provide added flexibility to the messages created, each message may have one or more macros embedded in the text. A macro is identified by a leading percent sign (%), followed by one or more characters with no spaces.

Macros are expanded at run time with data supplied by the ExpandLocalMsg, GetMsg, or PrintMsg operation, or with data supplied by programs using any of these operations. The GetMsgUnexpanded operation can be used to retrieve a message from the message file and to place the unexpanded message in memory (that is, it does not expand any macros that may be present in the message).

(See Appendix H in the CTOS/VM Reference Manual for details on defining message file macros.)

Using a Small Number of Messages

The OpenServerMsgFile, CloseServerMsgFile, and GetServerMsg operations are provided for when a system service or an application program requires using very few messages. With these operations, the entire contents of the message file are copied into a memory buffer, and the messages are extracted from that buffer. These operations do not support macro expansion.

OPERATIONS

The NLS operations described below are categorized as utility or message-related. Operations are arranged in a most to least frequent use order. (See the CTOS/VM Reference Manual, Chapter 3, "Operations for a complete description of each operation.)

UTILITY

GetNlsDateName
: Returns a string containing the names of the months and the weekdays, as well as the strings: Am, Pm, Noon, and Midnight.

GetNlsKeyCapText
: Returns a string that contains the text to be displayed by programs when reference to a labeled key is desired.

GetpNlsTable
: Returns the memory address of an NLS data table located in the NLS table area.

NlsCase
: Translates a given character from lowercase to uppercase, or from uppercase to lowercase.

NlsClass
: Takes a given character and returns the class of that character.

NlsCollate
: Compares two strings to determine if they are equal or if one is greater than the other. Programs should use this operation rather than NlsULCMPB or ULCMPB for a richer set of collation rules.

NlsFormatDateTime
: Converts from date/time format to textual string format. This operation employs a user-supplied format template in an alternate set of NLS tables linked with an application program (rather than the NLS tables loaded at boot time).

NlsNumberAndCurrency
: Returns the address of the Number and Currency Formats table.

NlsParseTime
: Converts a string into an expanded date/time structure.

NlsStdFormatDateTime
: Converts from date/time format to text string format. This operation uses an index into a set of template strings in the Date and Time Formats table loaded at boot time. Programs should use this operation rather than NlsFormatDateTime or FormatDateTime for ease in nationalization.

NlsULCMPB
: Same as ULCMPB.

ULCMPB
: Compares two strings, using the lowercase to uppercase conversion table, if present, to carry out the case-insensitive comparison. ULCMPB returns 0FFFFh if the two strings are equal; otherwise, it returns a word containing the index of the first two characters in the strings that are different. Programs should use ULCMPB instead of NlsULCMPB for ease in nationalization.

NlsVerifySignatures
: Validates an alternate set of NLS tables (that is, it ensures that the signatures embedded within the alternate table area provided match those expected to be there).

NlsYesNoOrBlank
: Performs a case-insensitive string comparison against nationalized words meaning yes or no and also checks for a null string.

NlsYesOrNo
: Performs a case-insensitive string comparison against nationalized words meaning yes or no.

MESSAGES

CloseMsgFile
: Closes an open message file.

CloseServerMsgFile
: Closes a message previously opened by a call to OpenServerMsgFile.

ExpandLocalMsg
: Expands any macro definitions contained in a message that resides in local memory.

GetMsg
: Retrieves a message from the message file and places the expanded message in memory.

GetMsgUnexpanded
: Retrieves a message from the message file and places the unexpanded message in memory (that is, it does not expand any macros that may be present in the message).

GetServerMsg — Extracts a particular message (string) from a message file that was previously initialized by InitServerMsgFile.

InitMsgFile — Opens a binary message file for subsequent retrieval of numbered messages.

OpenServerMsgFile — Initializes a message file for use by a system service or an application program that is using a relatively small message file.

PrintMsg — Retrieves a message from the message file and places the expanded message in a user-supplied video byte stream.

GLOSSARY

<$> directories. An area of memory on disk in which temporary files can be created. When a request with the directory name of <$> is given as part of a file specification, the operating system expands the directory name to the form <$000>nnnnn>, where nnnnn is the user number associated with the application partition. The scratch volume should contain the <$> directory. See also **User number.**

Abort request. Notifies system services that clients have terminated. Upon notification, system services can release resources, such as open files and locked ISAM records, allocated to the terminating clients. Issuing an abort request ensures that no requests are returned to the program after it has been terminated and replaced in memory by another program. The abort request also informs system services that resources allocated to the program should be freed.

Accessed bit. See **Access rights byte.**

Access rights byte. One byte of a descriptor that contains information about a segment, such as whether the segment is present in memory, what the privilege level is, and whether the segment contains code or data. The segment descriptor access rights byte contains, in addition, an accessed bit for use by least-frequently-used algorithms in virtual memory management.

Action code. The keyboard code generated when a key (**Cancel**, **Help**, **0** through **9**, or **F1** through **F10**) is pressed in conjunction with Action. Programs can call ReadActionCode or ReadActionKbd to obtain the action code of a specified key combination. See also **Action key.**

Action key. This key is processed specially, even in unencoded mode. The interpretation of all other keys is modified while **Action** is depressed. The key combination **Action-Finish** terminates the execution of the application program and invokes the Executive. **Action-A** and **Action-B** invoke the Debugger if the Debugger is included in the system at system build. Key combinations that include **Action** are available for application program interpretation. This allows the program to test for special operator intervention without preventing type ahead. Key combinations that include **Action** are processed immediately when they are typed. This processing is independent of characters or keyboard codes stored in the type-ahead buffer. See also **Action code.**

AL. Accumulator general register low byte.

Allocation bit map. Controls the assignment of disk sectors. It consists of 1 bit for every sector on the disk. The bit is set if the sector is available. The allocation bit map is disk-resident.

Alphanumeric style RAM. The video hardware controller for character attributes, such as blinking, half-bright, reverse video, and underlining, which are present on monotone graphics workstations.

Alternate request procedural interface. A convenient way to issue a request that uses a user number other than that of the caller. This frees the client from having to construct a request block. The alternate request procedural interface is constructed by prefixing the name of an operation with **Alt** and supplying the chosen user number as the first parameter to the procedure. For example, to issue a CloseFile request with user number 5 and file handle (fh), the request would be written as AltCloseFile(5, fh).

Application partition. A partition of user memory in which an application program can execute. A workstation can have any number of application partitions, with an application program executing concurrently in each. See also **System partition.**

Application process. Executes code in the application program. It is not a system service process. See also **System service process.**

Application program. Can consist of code, data, and one or more processes executing in an application partition. If the program is executing in a variable partition, the program's code can be located anywhere in memory and can be shared by the same type of program in a different variable partition.

Application System Control Block (ASCB). Communicates parameters, the termination code, and other information between an exiting application program and a succeeding application program in the same partition. See also **Variable Length Parameter Block.**

Application Workstation. See **AWS Workstation.**

ASCB. See **Application System Control Block.**

Asynchronous mode. See **Asynchronous operation.**

Asynchronous operation. Asynchronous operation is a procedure or protocol that allows for a response within a window of time rather than at an exact time interval.

Asynchronous Terminal Emulator. The **Asynchronous Terminal Emulator** (ATE) command allows a workstation to emulate an asynchronous, character-oriented ASCII terminal (glass TTY). (See the Executive Manual for details.)

ATE. See **Asynchronous Terminal Emulator.**

AVR. Automatic Volume Recognition.

AWS. See **AWS Workstation.**

AWS Workstation. A workstation that has no multibus slots. AWS workstations are supported on prior operating system versions with which CTOS/VM is cluster-compatible.

Bad sector file. Contains an entry for each unusable sector of a disk. The bad sector file is 1 sector in size.

Banner page. Optionally printed by the spooler before the printing of each file. The banner page is visually distinctive and also identifies the file being printed. The banner page can contain the text of a notice file. See also **Notice file** and **Spooler**.

Base I/O address. An address on the X-Bus assigned to an X-Bus module by the bootstrap ROM. A base I/O address is used for I/O access to that module.

Binary mode. One of three printing mode options in the printer: Generic Print System, pre-GPS spooler, and communications byte streams. Binary mode does not print the banner page before each file, send extra code not in the file to the printer, or recognize the escape sequence. See also **Image mode** and **Normal mode**.

Bit map workstation. Uses video software to emulate a character map to support character-only application programs. See also **Character map**, **Character map workstation**, and **Video refresh**.

Block. An area of memory allocated for use by Inter-CPU or cluster communications. See also **X-Block**, **Y-Block**, and **Z-Block**.

Blocked. A record file with several records per physical sector. See also **Record Sequential Access Method** and **Spanned**.

Boot block. The area of memory that contains the information passed to the operating system by the bootstrap ROM.

Bootstrap. To start (to boot) the system by reloading the operating system from disk. On other systems, this is often known as initial program load (IPL).

BP. Base Pointer general register.

BSWA. See **Byte Stream Work Area**.

Buffer management modes. The Direct Access Method provides two modes of buffer management, write-through and write-behind. See also **Write-behind mode** and **Write-through mode**.

Byte stream. A character-oriented, readable (input) or writable (output) sequence of 8-bit bytes used by the Sequential Access Method to transfer data to or from a device. An input byte stream can be read until either the program chooses to stop reading or it receives status code 1 ("End of file"). An output byte stream can be written until the program chooses to stop writing. See also **Byte Stream Work Area**, **Communications byte stream**, **Disk byte stream**, **Generic Print System byte stream**, **Keyboard byte stream**, **Pre-GPS Spooler byte stream**, **Printer byte stream**, **Sequential Access Method**, **Tape byte stream**, **Video byte stream**, and **X.25 byte stream**.

BSWA. See **Byte Stream Work Area**.

Built-in. A program is built-in if it is part of the operating system core, which is always in memory. A dynamically installed program, on the other hand, is a program that can be added or removed at any time without regenerating the operating system. The file system is an example of a built-in system service. The Queue Manager is dynamically installed.

BX. Base general register.

Byte Stream Work Area. A 130 byte memory work area for the exclusive use of Sequential Access Method procedures. Any number of byte streams can be open concurrently, using separate Byte Stream Work Areas.

Case value. A value used to identify which command invoked the current command when more than one possibility exists. The case value is held in the Variable Length Parameter Block and can be queried by a run file to determine which command actually invoked it.

cb. A variable prefix that indicates the count of bytes in a string of bytes.

Character attribute. Controls the presentation of a single character. The standard character attributes are reverse video, blinking, half-bright, underlining, bold, and struck-through. See also **Screen attribute**.

Character cell. The pattern of illuminated dots (or pixels) on the video display of a workstation. The character cell size can be used by a program that calls the QueryVidHdw or QueryVideo operation to obtain other information describing the level of video capability of the workstation.

Character Class table. An NLS table used by the NlsClass operation to indicate the class of a given character. The class can be numeric, alphabetic, special (nonalphanumeric but commonly displayed), graphic, or blind (not generally displayed). See also **NLS table**.

Character code. In character mode, the 8 bit byte returned by certain keyboard management operations (in contrast to the keyboard code returned when the keyboard is in unencoded mode). The character code signifies the depression of a key other than **Shift**, **Code**, **Lock**, or **Action**. Depression of **Shift**, **Code**, and **Lock** does not generate a character code, but influences the character codes generated for other keys depressed simultaneously. **Action** has a special, system-wide meaning. See also **Character mode**.

Character map. The area of memory that holds the coded representation of the characters displayed on the video display of a character map workstation. See also **Video refresh**.

Character map workstation. Contains video hardware that supports the character map for the video display of characters. The hardware reads characters and attributes from memory and converts them from the extended ASCII (8 bit) representation to a pattern of dots (or pixels) that it displays on the video display of a workstation. During this translation, the video hardware references a font that is loaded into memory under program control. See also **Bit map workstation** and **Character map**.

Character mode. In character mode (the default mode), the client process receives an 8 bit character when a key other than **Shift**, **Code**, **Lock**, or **Action** is pressed. See also **Character code** and **Unencoded mode**.

Character set. See **Standard character set**.

Check. A Kernel primitive used by a client to determine if a message is queued at a specified exchange. If one or more messages are queued, the message that was first queued is removed from the queue, and its memory address is returned to the client. If no messages are queued, status code 14 ("No message available") is returned.

CLI. See **Command Line Interpreter**.

CISR. See **Communications Interrupt Service Routine**.

Client process. A process that makes a request of a system service. Any process, even a built-in operating system process, can be a client process, since any process can request system services. See also **Queue Manager** and **System service process**.

Cluster configuration. A local resource-sharing network consisting of a master connected to cluster workstations. One high-speed RS-422 channel is standard on each workstation. In cluster configurations connected to a master workstation, the master and all of the workstations connected to it use this channel for intercluster communications. For large clusters with an SRP master, multiple RS-4 22 channels are provided. The operating system executes in each cluster workstation and in the master. Also see **Cluster workstation**, **CT-Net**, **Master**, and **Master workstation**.

Cluster workstation. A workstation in a cluster configuration, connected to a master. See also **Cluster configuration** and **Master**.

Cluster Workstation Agent. The Cluster Workstation Agent converts interprocess requests to interstation messages for transmission to the master. The Cluster Workstation Agent service process is included at system build in a system image that is to be used on a cluster workstation. See also **Master** and **Master Workstation Agent**.

Context Manager. See Context Manager/VM.

Context Manager/VM. A partition managing program. See **Partition managing program**. (See also the Context Manager/VM Manual.)

Code segment. A variable-length (up to 64K bytes) logical entity consisting of reentrant code and containing one or more complete procedures. See also **Data segment**, **Segment**, and **Virtual Code Management facility**.

Collating Sequence table. The Collating Sequence table actually consists of four NLS tables used by the NlsCollate operation for collating strings. See also **NLS table**.

Color control structure. Used by programs to set the color in any of the three palettes of the color palette structure and to switch the graphics bit map to use either of the two graphics palettes. In addition, a program can turn the alpha character map and graphics bit map on or off independently. To set values for fields in the color control structure, the program must call the ProgramColorMapper operation. See also **Color palette**, **Character map**, and **Graphics bit map**.

Color mapper. A portion of the memory into which the color palette is loaded. The color mapper thus determines what colors are visible on the screen. See also **Color palette**.

Color palette. The color palette structure contains three palettes: one for characters (alpha) and two for graphics (graphics1 and graphics2). A set of eight colors can be used for color specification on certain workstations. See also **Color mapper**.

Command Line Interpreter. A software program on an SRP processor that reads the Job Control Language (initialization) file to install the processor's system services.

Command name. The string a user types on the command line in the Executive to call a program. When the user presses **Return**, the Executive is given the command and responds by displaying the appropriate command form to the screen.

Comm nub. The part of the operating system that dispatches RS-232-C communications interrupts. The comm nub passes control from the hardware interrupt to a user-written RS-232-C communications interrupt handler (also called an interrupt service routine) according to the instructions in an InitCommline operation. When the interrupt handler has completed processing the interrupt, it passes control back to the comm nub.

Common unallocated memory pool. A single contiguous area of memory in each application partition from which long-lived and short-lived memory segments are allocated.

Communications byte stream. A byte stream that uses a communications channel. See also **Byte stream**, **Byte Stream Work Area**, **Disk byte stream**, **Generic Print System byte stream**, **Keyboard byte stream**, **Pre-GPS spooler byte stream**, **Printer byte stream**, **Sequential Access Method**, **Tape byte stream**, **Video byte stream**, and **X.25 byte stream**.

Communications Interrupt Service Routine (CISR). Similar to a mediated interrupt handler, except that a CISR serves only one of the two communications channels of the SIO communications controller (also called a communications interrupt handler). See also **Mediated interrupt handler**.

Communications status buffer. A system structure that contains statistics for the master and the workstations connected to it.

Configuration file. Specifies data to be used by the operating system, a utility, or an application program. Example configuration files are Config.sys and the device configuration files created by the Create Configuration File command through the Executive. (See the Executive Manual.)

Conforming code/expand-down data segment bit. One of the bits in the access rights byte. See **Access rights byte**.

Context switch. Occurs when a process is interrupted and its register contents are saved. When a process is preempted by a process with a higher priority, the operating system saves the hardware context of the preempted process in that Process Control Block. When the preempted process is rescheduled for execution, the operating system restores the content of the registers.

The context switch permits the process to resume as though it were never interrupted. See also **Process**, **Process context**, and **Process Control Block**.

Control information. The data after the request block header and before the first request address/size (pb/cb) pair.

CP. Cluster Processor.

CPU. The CPU (central processing unit) is the microprocessor.

Crash dump area. The crash dump area (the file [Sys]<Sys>CrashDump.Sys) contains a binary memory dump in the event of a system failure.

CRC. Cyclic Redundancy Check.

CS. Code segment.

Current. A current user number is the one that is presently executing.

CT-Net Agent. Receives requests over CT-Net destined for system services located at remote nodes and forwards these requests to the remote nodes. See also **CT-Net** and **CT-Net Server**.

CT-Net configuration. See **CT-Net**.

CT-Net. A network consisting of nodes connected by communications lines over long distances. A node is a junction in CT-Net (such as a workstation or a processor board on the SRP). CT-Net provides access to the system services of interconnected cluster configurations.

CT-Net Server. Responds to requests from CT-Net Agents. The Net Server receives a request block from the Net Agent, executes the request on behalf of the remote client, and returns the response to the originating Net Agent. See also **CT-Net Agent** and **CT-Net**.

CTOS. Convergent Technologies' operating system, which runs on the Intel 8086 Microprocessor.

CTOS/VM. Convergent Technologies' operating system for "virtual machine" workstations and the SRP.

Cursor RAM. Part of the advanced video capability, which allows software to specify a 10 by 15 bit array as a pattern of pixels in place of the standard cursor (a blinking underline). The cursor bit array is superimposed on the character and blinks.

CWS. See Cluster Workstation.

DAM. See Direct Access Method.

Data block. Either a quarter-inch (QIC) tape fixed-sized (512 byte) physical record or a half-inch tape variable-sized record.

Data segment. Contains data; it can also contain code, although this is not recommended. If a data segment is shared among processes, concurrency control is the responsibility of those processes. A data segment that is automatically loaded into memory when its containing run file is loaded is called a static data segment, to differentiate it from a dynamic data segment that is allocated by a request from the executing process to the memory management facility. See also **Code segment** and **Segment**.

Date/time format. Provides a compact representation of the date and the time of day that precludes invalid dates and allows simple subtraction to compute the interval between two dates. The date/time format is represented in 32 bits to an accuracy of 1 second.

Date and Time Formats table. An NLS table used to specify format templates to control date and time construction. This table provides for variations of date and time by country and by application program. See also **NLS table**.

Date Name Translations table. An NLS table used by GetNlsDateName and NlsParseTime to translate names of the months and days of the week. See also **NLS table**.

DAWA. See Direct Access Work Area.

DCB. See Device Control Block.

DCE. Data communications equipment.

Default response exchange. Each process is given a unique default response exchange when it is created. This special exchange is automatically used as the response exchange whenever a client process uses the request procedural interface to a system service. For this reason, the direct use of the default response exchange is not recommended. The use of the default response exchange is limited to requests of a synchronous nature. That is, the client process, after specifying the exchange in a Request, must wait for a response before specifying it again (indirectly or directly) in another Request. See also **Exchange** and **Response exchange**.

Descriptor privilege level. A feature of protected mode that indicates the privilege level of a segment. See also **Access rights byte**.

Device. A physical hardware entity. Printers, tape, floppy disks, and hard disks are examples of devices.

Device Control Block. There is a Device Control Block (DCB) for each physical device. The DCB contains information, generated at system build, about the device. For a disk, the information includes how many tracks are on a disk, the number of sectors per track, and so forth. The DCB contains the memory address of a chain of I/O blocks. The DCB is memory-resident.

Device-dependent. Describes program interfaces closest to the actual hardware. A device dependent program is limited to performing I/O to a limited number of devices. See also **Device-independent**.

Device-independent. Describes program interfaces that are not close to the hardware. A device-independent program can perform I/O to a variety of devices. The Sequential Access Method operations, such as OpenByteStream, ReadByteStream, and CloseByteStream, are device-independent operations. See also **Device-dependent**.

Device password. Protects a device.

Device specification. Consists of a devname (device name).

Devname. Device name; the only element of a device specification.

Diacritical key handling. Part of a keyboard map-ping table that provides for the display of characters with diacritical marks, such as the German a with an umlaut. See also **Diacritical key pair**, **Keyboard Encoding table**, and **Keyboard Mapping table**.

Diacritical key pair. A pair of keys that provides diacritical key handling. The first key of a diacritical key pair enables diacritical mode; the second key displays the diacritical result. See also **Diacritical key handling**.

Direct Access Method. Provides random access to disk file records identified by record number. The record size is specified when the DAM file is created. DAM supports COBOL relative I/O, but can also be called directly from any of the Convergent languages. See also **Direct Access Work Area**.

Direct Access Work Area (DAWA). A 64 byte memory work area for the exclusive use of the Direct Access Method procedures. Any number of DAM files can be open simultaneously using separate DAWAs. See also **Direct Access Method**.

Direct Memory Access (DMA). Access to memory that does not require processor intervention. A DMA controller in the processor module controls the transfer of data over the X-Bus from a memory master or master/slave to the main processor's memory.

Direct printing. Transfers text directly from application program partition memory to the specified parallel or serial printer interface of the workstation on which the application program is executing. Direct printing is always accessed through the Sequential Access Method (disk byte streams). See also **Disk byte stream**, **Spooled printing**, and **Pre-GPS spooler byte stream**.

Directory. A collection of related files on one volume. A directory is protected by a directory password.

Directory password. Protects a directory on a volume.

Directory specification. Consists of a node (node name), volname (volume name), and a dirname (directory name).

Dirname. Directory name; the third element of a directory specification or a full file specification.

Disk byte stream. A disk byte stream is a byte stream that uses a file on disk. See also **Byte stream**, **Byte Stream Work Area**, **Communications byte stream**, **Generic Print System byte stream**, **Keyboard byte stream**, **Pre-GPS spooler byte stream**, **Printer byte stream**, **Sequential Access Method**, **Tape byte stream**, **Video byte stream**, and **X.25 byte stream**.

Disk extent. One or more contiguous disk sectors that compose all or part of a file.

DMA. See **Direct Memory Access**.

Doorbell interrupt. A handshake protocol in which a device interrupts another device by writing to a doorbell interrupt location. The device being interrupted responds by servicing the interrupt and resetting the interrupt request on the device generating the interrupt. A timeout may or not be implemented. A doorbell interrupt is used on the SRP for notifying a processor board that it has received a message from a remote processor board.

DP. Data processor.

DS. Data segment.

DTE. Data terminal equipment.

Dynamic data segment. See **Data segment**.

Dynamically installed system service. A program that was loaded as an application program and converted itself into a system service using the ConvertToSys operation. (See Chapter 31, "System Services Management.") Once installed, a dynamically installed system service has the same capabilities as a system service that was linked with the System Image during system build. A dynamically installed system service must use CTOS/VM operations (rather than system build parameters) to identify the request codes that it serves, specify its execution priority, establish its interrupt handlers, and so forth.

EAR. See **Extended Address Register**.

EOF. End of File.

EOI. End of Interrupt.

EOM. End of Medium.

EOT. End of Tape.

Erc. A status (error) code.

ES. Extra Segment.

Escape sequence. A special sequence of characters that invokes special functions. See also **Spooler escape sequence**, **Submit file escape sequence**, and **Multibyte escape sequence**.

Event. In the context of timer management, an event occurs when an interval elapses. See also **System event**.

Event-driven priority scheduling. When processes are scheduled for execution based on their priorities and system events, not on a time limit imposed by the scheduler. See also **Process** and **System event**.

Exchange. The path over which messages are communicated from process to process (or from interrupt handler to process). An exchange consists of two first-in, first-out queues: one of processes waiting for messages and the other of messages for which no process has yet waited. An exchange is referred to by a unique 16 bit integer. See also **Default response exchange** and **Response exchange**.

Executive. An interactive application program that accepts commands from the workstation user and requests the operating system to load programs to execute those commands. This function can be performed by the Convergent Executive or by a user-written Executive. The Executive is loaded from the file [Sys]<Sys>Exec.Run if specified as the SignOnExitFile. (See the Release Notice for the current operating system version.) The file [Sys]<Sys>Exec.Run usually contains the Convergent Executive; however, it can contain a user-written Executive.

Exit run file. A user-specified file that is loaded and activated when an application program exits. Each application partition has its own exit run file.

Extended partition descriptor. Located in each application partition and contains specifications for the current application file and exit run file.

Extended User Control Block. Located in each application partition and contains the offset of the Partition Descriptor. See also **Partition Descriptor**.

Extension File Header Blocks. Required for each file that contains more than 32 disk extents. See also **File Header Block**.

External interrupt. Caused by conditions that are external to the processor and are asynchronous to the execution of processor instructions. There are two kinds of external interrupts: maskable and nonmaskable. See also **Internal interrupt**, **Maskable interrupt**, and **Nonmaskable interrupt**.

FAB. See **File Area Block**.

FALSE. Represented in a flag variable as 0.

Far procedure. Referenced by the procedure's code segment (CS) and offset (IP). A far procedure can be called by procedures within the same or from within a different module.

FCB. See **File Control Block**.

fh. File handle.

FHB. See **File Header Block**.

FIFO. First-in-first-out.

File. A set of related bytes (on disk) treated as a unit.

File Area Block. There is a File Area Block for each disk extent in an open file. The FAB specifies where the sectors are and how many there are in the disk extent. The FAB is pointed to by a File Control Block or another FAB. The FAB is memory-resident. See also **Disk extent**.

File System Case table. The File System Case table is an optional NLS table used by the file system for case-insensitive comparison and hashing. See also **NLS table**.

File Control Block. There is a File Control Block (FCB) for each open file. The FCB contains information about the file such as the device on which it is located, the user count (that is, how many file handles currently refer to this file), and the file mode (modify, peek, or read). The FCB is pointed to by a User Control Block and contains a pointer to a chain of File Area Blocks. The FCB is memory-resident.

File handle. A 16 bit integer that uniquely identifies an open file. It is returned by the OpenFile operation and is used to refer to the file in subsequent operations such as Read, Write, and DeleteFile.

File Header Block. There is a File Header Block (FHB) for each file. The FHB of each file contains information about that file such as its name, password, protection level, the date/time it was created, the date/time it was last modified, and the disk address and size of each of its Disk Extents. The FHB is disk-resident and one sector in size. See also **Extension File Header Block**.

File password. Protects a file in a directory on a volume.

File protection level. Specifies the access allowed to a file when the accessing process does not present a valid volume or directory password.

Filename. File name; the fourth element of a full file specification.

Filter process (user-defined). A user-written system service process that can be included in the System Image at system build or dynamically installed at any time. A filter process is interposed between a client process and a system service process that operate as though they are communicating directly with each other. The Service Exchange table is adjusted to route requests through the desired filter process.

Filter process (local file system). See **Local file system**.

Fixed partition. Always uses a fixed amount of memory. See also **Variable partition**.

Font. A bit array for each of the 256 characters in the character set that defines the representation of each character when displayed on the video display.

Font RAM. For the video, contains a bit array for each of the 256 characters in the character set. The font RAM can be modified under software control.

ForwardRequest. A Kernel primitive that can be used by a one-way pass-through filter to forward a request block to a system service for further processing. The system service responds directly to the client.

FP. File Processor.

Frame. A separate, rectangular area of the screen. A frame can have any desired width and height (up to those of the entire screen).

Frame descriptor. A component of the Video Control Block containing all information about one of the frames. The number of frame descriptors in the Video Control Block is specified at system build. See also **Video Control Block**.

Free memory. Unused system memory.

Full file specification. Consists of a node (node name), volname (volume name), dirname (directory name), and filename (file name).

GDT. See **Global Descriptor Table**.

Generic Print System (GPS). The Generic Print System is made up of a set of dynamically installed system services, which work together to handle communication between application programs, the operating system, and the printers and plotters currently installed. GPS is the software underlying the Print Manager. (For details, see the Generic Print System Programmer's Guide and the Printing Guide.)

Generic Print Access Method (GPAM). The Generic Print Access Method provides high-level I/O for complex documents that may include text, graphics, or special text attributes. GPAM is an object module library that provides device independent formatting commands used for printing. GPAM is used typically to add rich formatting characteristics to text that is output to a printing device. (For details, see the Generic Print System Programmer's Guide and the Printing Guide.)

Generic Print System byte stream. A byte stream sent to a GPS printing device. See also **Generic Print System** and **Print Manager**.

Global Descriptor Table. A protected mode structure that contains descriptors for segments, which are shared by all programs. See also **Local Descriptor Table (LDT)** and **Segment descriptor**.

Global request. A request that can be served by a system service on any SRP processor board. See also **Local request**.

GPS. See **Generic Print System**.

Graphics bit map. The graphics bit map is a three plane bit map that is manipulated by operations in the graphics library. (See the Graphics Programmer's Guide.)

Graphics style RAM. The video hardware controller of character attributes, color, and intensity on color graphics workstations. Color and intensity specifications are available with the attributes of reverse video and underlining. An 8 byte memory work area is allocated to specify the entries that are passed to the graphics style RAM. Each byte uses the low-order 6 bits for color specification and the high-order 2 bits for reverse video and underlining, respectively. See also **Character attribute**.

Hashing techniques. See **Randomization techniques**.

High-level interface. Programmatic interfaces, which, when used exclusively, provide device-independence to a program. See also **Device-independent** and **Low-level interface**.

High-resolution. The video resolution of a graphics controller that produces 12 X 20 pixel (illuminated dot) characters on the screen. See also **Low resolution**.

ICC. See **Inter-CPU communication**.

ICC Server Agent. On each SRP processor board, issues requests on behalf of a client on a different processor board.

ICMS. See **Intercontext Message Server**.

IDT. See **Interrupt Descriptor Table**.

Image mode. One of three printing options in the Generic Print System, pre-GPS spooler, printer, and communications byte streams. Image mode prints the banner page before each file and recognizes escape sequences but performs no code conversions. See also **Normal mode** and **Binary mode**.

Intercontext Message Server (ICMS). Used by application programs to communicate with programs in other application partitions. The requesting program sends an interprocess communication message to ICMS. ICMS, in turn, uses interprocess communication to forward the message to the receiving program. ICMS prevents messages from being sent to programs while they are swapped out of memory.

Indexed Sequential Access Method (ISAM). Provides efficient, yet flexible, random access to fixed-length records identified by multiple keys stored in disk files. (See the ISAM Manual.)

Input byte stream. See **Byte stream**.

Interactive. A program is that interfaces with the user. The Executive is an example of an interactive command interpreter.

Inter-CPU communication. Inter-CPU communication is used by the Kernel on a processor board in an SRP to send request and response messages between boards.

Interface level. Implies the relative degree of program control over a hardware device.

Internal interrupt. Caused by and is synchronous with the execution of processor instructions.

Internationalization. Language independence. Source code is internationalized when it is written in such a way that the resulting program can run in different languages without modifying the run file itself.

Interrecord gap. The space between records in a half-inch tape file.

Interrupt. External or internal; an event that interrupts the sequential execution of processor instructions. When an interrupt occurs, the current hardware context (the state of the hardware registers) is saved. This context save is performed partly by the processor and partly by the operating system. See also **External interrupt**, **Internal interrupt**, **Maskable interrupt**, **Non-maskable interrupt**, and **Pseudointerrupt**.

Interrupt Descriptor table. The protected mode equivalent of the Interrupt Vector table. The tables function similarly in that each directs interrupts to the appropriate interrupt handling routines.

Interrupt handler. A locus of computation that is given control when an interrupt occurs. Since an interrupt handler is not a process, it is permitted to invoke only a few specific operations. Operating system interrupt handlers are provided for each interrupt type. Each interrupt handler services all interrupts of a single type. The operating system supports two kinds of interrupt handlers, mediated and raw. Different styles of mediated and raw interrupts exist for RS-232-C communications and all other (non-RS-232-C) interrupt types. See also **Mediated interrupt handler** and **Raw interrupt handler**.

Interrupt number. Each potential source of interrupt is assigned an interrupt number in the range 0 to 255 that identifies the interrupt type (source of the interrupt). When an interrupt occurs, the hardware recognizes the interrupt type and the applicable interrupt number. The processor uses this number as an index into the Interrupt Vector table (real mode) or the Interrupt Descriptor table (protected mode) to vector (direct) the interrupt to the appropriate interrupt handler. See also **Interrupt** and **Interrupt handler**.

Interrupt service routine. An interrupt service routine is an interrupt handler. See **Interrupt handler**.

Interrupt Vector table. The Interrupt Vector table is a real mode structure that contains a 4 byte entry for each interrupt type. Each 4 byte entry contains the logical memory address (CS:IP) of the first instruction to be executed when an interrupt of that type occurs. See also **Interrupt number.**

IOB. See **I/O Block**.

I/O Block (IOB). Used by the operating system as temporary storage during Read, Write, and other I/O operations. The IOB contains information obtained from the request block. The number of IOBs specified at system build must be adequate for the maximum number of input/output operations that will be in progress simultaneously. The IOB is memory-resident.

IPC. Interprocess communication. (See Chapter 29, "Interprocess Communication Management.")

IPL. Initial Program Load.

ISAM. See **Indexed Sequential Access Method**.

IWS. See **IWS Workstation**.

IWS Workstation. A Convergent workstation that has two (or optionally five) multibus slots. IWS workstations are supported on prior operating system versions with which CTOS/VM is cluster-compatible.

Kernel. The most primitive and the most powerful component of the CTOS/VM operating system. It executes with a higher priority than any process but it is not itself a process. The Kernel is responsible for the scheduling of process execution; it also provides IPC primitives.

Keyboard byte stream. A byte stream that uses the keyboard. See also **Byte stream**, **Byte Stream Work Area**, **Communications byte stream**, **Disk byte stream**, **Generic Print System byte stream**, **Pre-GPS spooler byte stream**, **Printer byte stream**, **Sequential Access Method**, **System input process**, **Tape byte stream**, **Video byte stream**, and **X.25 byte stream**.

Keyboard code. In unencoded mode, the 8 bit byte returned by certain keyboard management operations. The keyboard code identifies the key in the low-order 7 bits and indicates the direction of key motion in the high-order bit. A 0 indicates key depression,- 1 indicates key release. Also see Unencoded mode. (The keyboard codes are in Appendix C of the CTOS/VM Reference Manual.)

Keyboard Encoding table. Used in converting the sequence of keyboard codes to 8 bit character codes. The table controls several aspects of the keyboard-code-to-character-code translation: the character code to generate if **Shift** is/is not depressed/ whether **Lock** has the effect of **Shift** for a key; whether the key is typematic (repeats); the initial delay before beginning typematic repeating; the frequency of typematic repeating; and whether a key responds to diacritical key handling. The Keyboard Encoding table can be modified dynamically, as well as at system build. See also **Keyboard Mapping table** and **Diacritical key pair**. (See Appendix B of the CTOS/VM Reference Manual for the default contents of the Keyboard Encoding table.)

Key Cap Legends table. The Key Cap Legends table is an NLS table used by the NlsKeycapText operation to specify the text strings to be displayed by programs when making reference to any of the key caps that commonly contain legends. See also **NLS table**.

Keyboard Mapping table. An NLS table used to map keys pressed by the user to their character codes. If the NLS Keyboard Mapping table is loaded into memory as part of the Nls.sys file, the memory address of the Keyboard Encoding table defined during system generation is altered to reflect the address of the NLS Keyboard Mapping table. Keyboard mapping is implemented by either table. See also **Keyboard Encoding table** and **NLS table**.

Language definition. Includes those requirements of a language, such as currency symbols and date/time formats, that are unique to that language.

LDT. See **Local Descriptor Table**.

lfa. See **Logical file address**.

Limit checking. A protection feature of protected mode that places limitations on the memory a program can access.

Linear memory address. The linear memory addresses are at relative distances from memory address 0 in physical memory and can be compared to each other on this basis. See also **Linear address space**.

Linear address space. Begins at physical memory address 0 and extends linearly to the maximum amount of physical memory that actually can be addressed by a program. Linear addresses thus are a relative distance from the address 0 in physical memory and can be compared to each other on this basis. The linear address space is equivalent to the physical address space. Unless paging is enabled, a linear address also is equivalent to its physical address. See also **Linear address**, **Physical address space**, and **Physical memory address**.

Link block. A system data structure that is used to queue messages at exchanges. Each link block contains the address of the message and the address of the next link block (if any) that is linked onto the exchange. Two pools of link blocks are specified at system build, a general pool and a special pool used only by the PSend primitive. A call to the Request primitive reserves 1 link block from the general pool for the corresponding Respond primitive. For these reasons, the number of link blocks in each pool can be specified at system build.

Linker. Links one or more object files into a run file to be loaded into memory. (See the Linker/ Librarian Manual.)

Loadable Request file. A file containing request definitions for a system service(s). The Loadable Request file is used to merge new requests with the requests already defined in Request.sys. The merge occurs during installation of the system service(s) onto the system disk. When bootstrapped, the operating system reads Request.sys, loads it into memory, and adds the requests it contains to the basic request routing table.

Local Descriptor Table (LDT). A protected mode structure in memory that contains descriptors for segments accessible to a run file. The operating system constructs the LDT based on information provided by the Linker.

Local file system. Allows a cluster workstation to access files on a local hard disk(s) as well as files on the hard disk(s) at the master. The filter process of the local file system intercepts each file access request and directs it to the local file system or to the master workstation.

Local resource-sharing network. A cluster configuration consisting of cluster workstations connected to a master.

Local request. Served by a system service on the same processor board of an SRP as the client.

Log file. An error-logging file. An entry is placed in the Log File ([Sys]<Sys>Log.Sys) for each recoverable and nonrecoverable device error. This file can be used as a general-purpose logging file, for example, to write entries for accounting information for system services.

Logical file address. A logical file address (lfa) is used to locate a particular sector of a file. An lfa specifies the byte position within a file; it is the number (the offset) that would be assigned to a byte in a file if all the bytes were numbered consecutively starting with 0. An lfa is a 32 bit unsigned integer that must be on a sector boundary and is therefore a multiple of 512. For example, the lfa of the third sector of a file is 1024.

Logical memory address. The 3 2 bit memory address (usually abbreviated as memory address) as viewed by the application program. It consists of a 16 bit segment address (SA) and a 16 bit relative address (RA) or offset. A byte of memory does not have a unique logical memory address. The same byte of memory can be referred to by many different combinations of SAs and RAs. See also **Offset** and **Segment address**.

Long-lived memory. An area of memory in an application partition. It is used for parameters or data passed from an application program to a succeeding application program in the same partition. If a character map other than the one in the system partition is needed, it must be allocated in the long-lived memory area of the application partition. See also **Application partition** and **System partition**.

Low-level interface. A programmatic interface that is close to the actual hardware. Programs using low-level interfaces are device-dependent.

Low resolution. The video resolution of a graphics controller that produces 9 X 12 pixel (illuminated dot) characters on the screen. See also **High resolution**.

Maskable interrupt. Given a priority and controlled by the programmable interrupt controller and can be masked (ignored) by the use of the processor interrupt-enable flag. A maskable interrupt can be masked selectively by programming the programmable interrupt controller. See also **External interrupt** and **Nonmaskable interrupt**.

Master. Either a master workstation or an SRP.

Master processor. Either a file processor or a data processor on an SRP.

Master File Directory. There is an entry for each directory on the volume in the Master File Directory (MFD), including the Sys directory. The position of an entry within the MFD is determined by randomization (hashing) techniques. The entry contains the directory's name, password, location, and size. The Master File Directory is disk-resident.

Master workstation. A master workstation can serve a cluster configuration. The master workstation provides file system, queue management facility, and other services to all the cluster workstations. In addition, it supports its own interactive programs. See also **Cluster workstation** and **Cluster configuration**.

Master Workstation Agent. Reconverts a message passed between workstations in a cluster to an interprocess request and queues the request at the exchange of the system service on the master that actually performs the desired function. See also **Cluster Workstation Agent** and **Master**.

Mediated interrupt handler (MIH). One of two procedural styles for handling an interrupt. The other style is a raw interrupt handler. When compared to a raw handler, a mediated interrupt handler executes more slowly. This is because it can be written in a high level language, interrupts are enabled during its execution (so that it can be pre-empted), and it can communicate its results to processes through certain Kernel primitives. An example of a mediated interrupt handler is the keyboard interrupt handler. See also **Interrupt handler** and **Raw interrupt handler**.

Memory address. See **Logical memory address**.

Message. The entity transmitted between processes by the interprocess communication facility. It conveys information and provides synchronization between processes. Although only a single 4 byte data item is literally communicated between processes, this data item is usually the memory address of a larger data structure. The larger data structure is called the message, while the 4 byte data item is conventionally called the address of the message. The message can be in any part of memory that is under the control of the sending process. By convention, control of the memory that contains the message is passed along with the message.

MFD. See **Master File Directory**.

MIH. See **Mediated interrupt handler**.

Modify mode. One of three ways that a file can be opened using an operation, such as OpenFile or OpenFileLL, that can open a file. Modify mode is used to write to the file. Access in modify mode permits the returned file handle to be used as an argument to all operations that expect a file handle. See also **Peek mode** and **Read mode**.

Multibyte escape sequence. A special sequence of characters that is available to disable video byte stream interpretation of special characters except 0FFh. (See the Table J-7 in Appendix J in the CTOS/VM Reference Manual for the video byte stream interpretation of special characters. See Tables J-1 through J-6 in the same appendix for the multibyte escape sequences that can disable the special interpretations.)

Multiprogramming. The ability to run more than one program in memory at the same time. Multiprogramming supports the independent invocation and scheduling of multiple processes. In addition, it provides for concurrent I/O and for multiple processor implementations. See also **Partition managing program**.

Multitasking. See **Multiprocessing**.

Multiprocessing. The ability for any program to have more than one process (thread of execution). Multiprocessing also is called multitasking.

Nationalization. Results in software that runs using a single language definition. See also **Language definition**.

Near procedure. Referenced by the offset (IP) of the procedure's memory address. Near procedures can be called only by other procedures within the same module.

Network. See **CT-Net**.

Network routing. See **CT-Net**.

NGEN. See **NGEN Workstation**.

NGEN Workstation. A Convergent workstation that has two (or optionally five) multibus slots.

NMI. See **Nonmaskable interrupt**.

NLS table. One of several (optional) internationalizable tables supplied as part of Standard Software in the source file, [Sys]<Sys>Nls.asm. Included among the NLS tables, for example, are a table for uppercase to lowercase characters, a date/time formats table, and a symbols table for numbers and currency. The NLS tables can be edited, assembled, and linked to create the NLS configuration file [Sys]<Sys>Nls.sys. When the operating system is bootstrapped, the contents of this file are loaded into memory, making the NLS tables available to application programs via NLS operations. See also **System-common NLS table area**.

Node. The first element (node name) of a full file specification. A node is also a master or a cluster workstation that is part of a CT-Net. See also **CT-Net**.

Nonmaskable interrupt (NMI). Has a higher priority than a maskable interrupt. An NMI cannot be masked through the use of the processor interrupt-enable flag; however, bits in the I/O Control register allow each of the four conditions that cause NMI to be masked individually. These conditions are write-protect violation, nonexistent or device-addressed memory parity error, and power failure detection. See also **Maskable interrupt**.

INDEX

(Bolded page numbers represent primary discussions of entries.)