VAX

SOFTWARE HANDBOOK

digital

VAX

SOFTWARE HANDBOOK

digital

This handbook was designed, produced, and typeset
by DIGITAL's New Products Marketing Group
using an in-house text-processing system.

TABLE OF CONTENTS

PART IV
SITE CONSIDERATIONS

PREFACE

At DIGITAL we recognize that when you *buy* our computers you are making an *investment* in the future. So we design our computer systems to meet your needs today while anticipating tomorrow. Our systems are designed to let your applications grow with you. After all, that is what investments are all about. *At DIGITAL, we are committed to making our computers the best investment around. VAX epitomizes that commitment.*

You will encounter the term "VAX architecture" in this handbook. To really appreciate what a good investment VAX systems and software are, you need to know something about the architecture - what it is and why it is so important.*

The VAX architecture defines how a VAX processor will behave in relation to software. It is the standard to which all of the VAX processors must conform. It means that all of the software described in this handbook will run on any member of the VAX family of processors, from the new VAX-11/730 to the VAX-11/782 attached processor system. It also means that all of the software you have developed to run on one kind of VAX will run on any other kind of VAX. Since any future VAX system will conform to this same architecture, your investment in software engineering is protected with VAX.

The demands of software were central to the design of the architecture. In fact, the VAX architecture and the VAX/VMS operating system were conceived and designed together. We made sure that the VAX architecture enhanced the efficacy of the VAX/VMS operating system, and that the operating system takes advantage of the VAX processors. So, every VAX processor offers 32-bit virtual addressing, a sophisticated memory management and protection mechanism, and hardware assisted process scheduling and synchronization; all of which is exploited fully by VAX/VMS. In this way, VAX can provide a multi-user system for large and complex applications, and it can compile and run huge programs (up to four gigabytes) concurrently. This sort of capability used to be the exclusive domain of large and expensive mainframes. It means the application investment you make in VAX today has plenty of room to grow tomorrow.

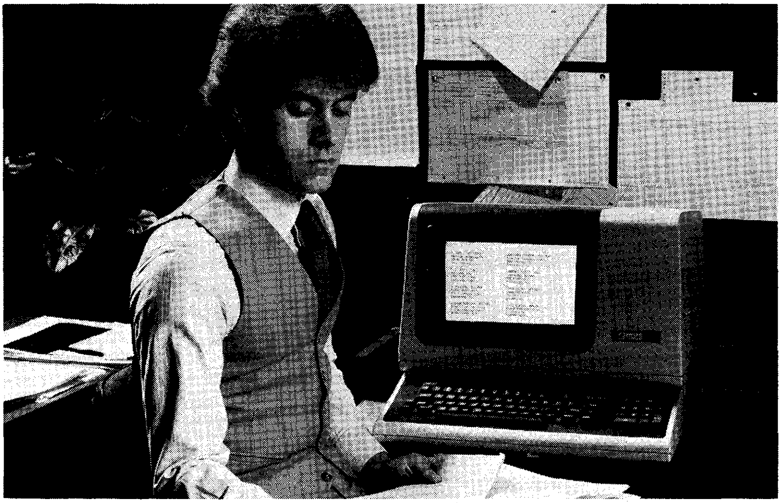
The VAX architecture was also designed to enhance program performance. For example, all of the VAX language processors take advantage of the powerful variable-length instruction set and numerous datatypes. The result is compilers that generate compact and efficient code, and do it very quickly—so that your applications running sooner and performing better.

But of course there is more than the VAX architecture going for your investment; there is the software itself. This handbook describes the extraordinary capabilities we have engineered into our software. The VAX/VMS operating system is easy to use, so it is easy to learn; and it comes with a compliment of very powerful tools to assist and streamline program development. The VAX language processors lead the industry in performance and features, and programs written in one language can call procedures written in any other language. The VAX information management software provides an unprecedented, complete system for managing your data. The networking options will allow your application to spread and take just about any shape you need. In all, its an impressive offering, as you will see.

One final note. When we designed the VAX architecture and VAX/VMS software, we were not only committed to the future, we were recognizing our commitments to the thousands of customers who have invested—and are still investing—in our PDP-11 computers; so we designed PDP-11 compatibility into VAX. Your investment in PDP-11 is protected because VAX gives it room to grow. Even if you don't own a PDP-11, its nice to know that when you invest in one of our computers, *we stand behind your investment.*

PART I

INTRODUCTION



CHAPTER OVERVIEW

This chapter offers a survey of the VAX software, including its services, controls, and capabilities. Brief descriptions in each section give quick insight into VAX/VMS-specific aspects. All topics are expanded in greater detail in later chapters.

Topics include:

- System Introduction
- Management of Virtual and Physical Memory
- Definition of a Process
- Scheduling and Swapping
- System Services, I/O Control, and I/O Devices
- Interprocess Communication
- Communications and Internets
- Realtime Capabilities
- Languages and Language Processors
- Data Management Facilities

CHAPTER 1

INTRODUCTION TO VAX SOFTWARE

SYSTEM INTRODUCTION

VAX is a family of high-performance multiprograming computer systems which combine a 32-bit architecture, efficient memory management, and a virtual memory operating system to provide essentially unlimited program address space.

The architecture's variable length instruction set and variety of data types, including decimal and character string, provide high bit efficiency. The instruction set specifically implements many high-level language constructs and operating system functions.

Each member of the VAX family is a multiuser system for both program development and application system execution. Each is a priority-scheduled, event-driven system: the assigned priority and activities of the processes in the system determine the level of service they need. Realtime jobs receive service according to their priority and ability to execute, while the system manages allocation of CPU time and memory residency for normal executing processes.

VAX systems are highly reliable. Built-in protection mechanisms in both the hardware and software ensure data integrity and system availability. On-line diagnostics and error detecting and logging verify system integrity. Many hardware and software features provide rapid diagnosis and automatic recovery should the power, hardware, or software fail.

The systems are both flexible and extendable. The virtual memory operating system enables the programmer to write large programs that can execute in both small and large memory configurations without requiring the programmer to define overlays or later modify the program to take advantage of additional memory. The DIGITAL Command Language enables users to modify or extend their command repertoire easily, and allows applications to present their own command interface to users.

To understand how the operating system functions, as described in this Handbook, a few definitions of some basic terms will be valuable. The user must first understand the concepts of program image and process, and know the difference between them. Please note that nearly all of the concepts and features introduced in this chapter are examined in greater detail in subsequent sections or chapters.

USER PROCESS

A program image is an executable program, created by translating source language modules into object modules, and linking the object modules together. An image is normally stored in a file on disk. When a user runs an image, the operating system reads from a copy of the image file into physical memory to execute it.

A procedure is a description of the logic to be performed to solve a problem; that is, it is a static definition of an algorithm. An image consists of procedures and data that have been bound together by the linker. Linking refers to the resolution of cross linkages among modules and the assignment of virtual address space.

The environment in which an image executes is its context. The complete context of an image includes not only the state of its execution at any one time (known as its hardware context), but also the definition of its resource allocation privileges and quotas, such as device ownership, file access, and maximum physical memory allocation. Certain software information, including some key addresses and some software data structures to be described later, comprise the software context. An image context and the image executing in the context are called a process.

Working Set

When a process executes, only a subset of its pages need be in physical memory. (A page contains 512 bytes, which is also the size of a physical page of memory and a disk block.) This subset of pages is referred to as the process's working set. The remaining pages of the process reside on secondary storage. Before a process is allowed to compete for central processor resources, its working set must reside in memory.

Balance Set

The set of processes that reside in physical memory is termed the balance set. This set of processes has memory requirements that balance with the available memory of the system. At any time during the execution of a process, its entire working set can be written to secondary storage, thereby freeing physical memory for another use. This is called swapping.

Software Process Control

The VAX/VMS operating system provides each process with software definitions used to control the process, and its working set. The operating system provides two key data structures to define a process, the software process control block (PCB) and the process header. Through **process identification**, the system also provides each process with a unique identifier.

VIRTUAL MEMORY

The VAX/VMS virtual address space consists of 2^{32} bytes, divided into system and process address spaces, each of which has 2^{31} bytes. The VAX/VMS system distinguishes between the physical memory required by a process and the virtual address space that the process defines. A process's virtual address space is the range of memory locations that the process can address.

Process virtual address space is divided into a program region and control region. The program region contains the image currently being executed. The control region contains information maintained on behalf of the process by the system, and it contains the user stack and the kernel, executive, and supervisor mode stacks. Only a small portion of the control region is reserved for context maintained by the system; the remainder is available to the user.

A process's virtual memory is subdivided into pages. System and user virtual space are described in a data structure called the **system page table** (SPT), which contains one page table entry (PTE) for each page of system virtual memory. When a virtual page is in memory, the page table entry contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary or disk storage.

A process's virtual address space is described in two page tables: the P0 page table for the program region and the P1 page table for the control region. Process page tables reside in system virtual memory. They are virtually contiguous, but not necessarily physically contiguous, nor necessarily in memory. The system page table resides in system virtual memory, but is physically based and physically contiguous.

The hardware system base register (SBR) and system length register (SLR) provide the physical address and the length in longwords of the system page table. Given the contents of SBR and SLR, it is possible to locate all other system virtual pages. From the process page tables contained in system virtual space, it is possible to locate all process virtual pages.

MEMORY MANAGEMENT

Memory management code maintains a database (the page frame number database) describing the status of all physical pages of memory and the status and location of all virtual pages of processes in the system. For example, a physical page could be part of a working set, or it could be available on the free page list for a process virtual page to be loaded into it.

Memory management uses page tables as the database to contain the status and location of virtual pages of processes. Each page of a process has a page table entry in the appropriate process page table to describe that page and its location. For example, a virtual page of a process could be in its image file, in its working set, in an in-memory cache, or on the modified page list.

Image Activator and Pager

Memory management is divided into two logically separate functions to control the pages of a process:

- Image activation
- Paging

The image activator is responsible for making an image capable of running in the context of the requesting process. The image activator locates the file containing the image and sets up the page table entries for it.

As page faults occur for pages in the process, the pager receives the faults, obtains a physical page, and brings the virtual page into the working set. If the limit on the number of pages in the process's working set has been reached, the pager selects a page to be removed from the working set. The pager selects the page to be deleted using information in the working set list portion of the process header.

Global Sections

Memory management uses widely available image sections, called global sections, to provide a mechanism for sharing code and data. A global section can be either of the following:

- A shareable image file produced by the linker and identified to the system by the system manager
- The result of a process's issuing a Create and Map Section system service

Global sections made from shareable images are permanent; they remain known to the system until explicitly deleted by the system manager. Global sections made as the result of a Create and Map Section system service are temporary or permanent; the system deletes temporary global sections automatically when no processes are using them.

Global sections are defined by a database that is similar in structure to that used to describe processes. Global sections consist of a number of pages. A page of a global section can be mapped into one or more process working sets. The one copy is shared among many processes. Both read-only and read/write global sections can be defined.

WORKING SET SWAPPER

The working set swapper is a small process that moves process working sets into and out of the balance set. The main function of the working set swapper is to provide memory residency for the highest priority executable processes so that they can be scheduled for execution.

Working set swapping occurs in two phases:

- The outswapping of nonexecutable or low priority processes from the balance set to free memory for inswap candidates
- Inswapping of processes from the executable nonresident state to the executable resident state

The working set swapper also performs initial process creation. Because process creation is accomplished using a shell (prototype) process that is swapped into memory, process creation requires little additional effort by the swapper. The shell process establishes the initial context and virtual memory of a new process.

PROCESS SCHEDULING

The VAX/VMS operating system defines 32 levels of software priority for scheduling processes. The lower 16 priorities (0 through 15) are reserved for normal processes, while the higher 16 priorities (16 through 31) are reserved for realtime processes. The highest priority executable resident process is always selected for execution. Realtime process priorities are established by the user and are not altered by the system. Normal process priorities are altered by the system to optimize responsiveness.

The process scheduler makes scheduling decisions by:

- Maintaining a queue for each state that a process can attain
- Reacting to system events

System events are occurrences that cause the status of one or more processes in the system to change. The scheduler reflects the change by removing the process's software Process Control Block (PCB) from one state queue and queuing it to the appropriate one.

SYSTEM PROCESSES

All VAX/VMS system functions are implemented as processes or as procedures that are called by user processes or by many system processes. A system process can be one of three types:

- Full process
- Small process
- Fork process

Full processes are user processes.

Small processes have no program region in their virtual address space and have an abbreviated context. They are scheduled in the same manner as full processes but must remain resident. For example, the working set swapper is a small process.

Fork processes have minimal context; they are defined by an abbreviated control block called a fork block. Fork processes execute at software interrupt levels and are dispatched for execution immediately. Fork processes execute until they are preempted by higher priority forks or they terminate. Device driver routines are examples of fork processes.

SYSTEM SERVICES

System services are procedures in the executive that can be called by user processes to provide controlled sharing of system resources. Because the system performs a service on behalf of the user, functions that require access to privileged databases are controlled.

Requests for system services are honored only if the requesting process has sufficient privilege and if protection is not violated.

Event Flag Services

Event-related system services are those services that allow a process or a group of cooperating processes to read, wait for, and manipulate event flags. The software Process Control Block (PCB) of each process contains two clusters of 32 event flags each that are local to the process. In addition, groups of cooperating processes can create and associate with two additional event flag clusters. These clusters are common to all associated processes with the same group number.

Lock Management Services

The lock management services provide a mechanism for synchronizing access to a common resource by cooperating processes. There are six choices of lock mode, each providing a different level of sharing. Resources are defined by a tree-structured nametable. The depth of hierarchy employed in the nametable determines the degree of granularity in defining and controlling access to the resource.

Asynchronous System Trap (AST) Services

Process execution can be interrupted by events (such as I/O completion) for the execution of designated subroutines. These software interrupts are called asynchronous system traps (ASTs) because they occur asynchronously to process execution. System services are provided so that a process can control the handling of ASTs.

Logical Name Services

Logical name services provide a generalized technique for maintaining and accessing character string logical name and equivalence name pairs. Logical names can provide device-independence for system and application program input and output operations. Logical name re-assignment is also the most convenient and flexible facility for moving an application from a single-CPU system to a multiple-CPU system.

I/O System Services

I/O services perform input and output operations directly, rather than through the file handling provided by the VAX/VMS Record Management Services (RMS). I/O services:

- perform physical, logical, and virtual input /output operations
- Format output lines converting binary numeric values to ASCII strings and substituting variable data in ASCII strings
- Perform network operations
- Queue messages to system processes
- Create mailboxes, which are virtual devices for interprocess communication.

Process Control Services

Process control system services allow the user to create, delete, and control the execution of processes.

Timer and Time Conversion Services

Timer services schedule program events for a particular time of day, or after a specified interval of time has elapsed. The time conversion services provide a way to set, obtain, and format binary time values for use with the timer services.

Condition Handling Services

Condition handlers are procedures that can be designated to receive control when a hardware or software condition occurs during image execution. Condition handling services designate condition handlers for special purposes.

Memory Management Services

Memory management system services allow a process to control its use of virtual and physical memory. Included are services that:

- Allow an image to increase or decrease the amount of virtual memory available
- Control the paging and swapping of virtual memory
- Create and access memory files that contain shareable code and data

Change Mode Services
Change mode services alter the access mode of a process to a more privileged mode to execute particular routines. Use of these services requires privilege.

INTERPROCESS COMMUNICATION AND SYNCHRONIZATION

The VAX/VMS operating system provides a variety of methods for processes to communicate with each other and synchronize their execution. The method selected for interprocess communication is affected by a number of variables, including: the level of explicit cooperation between the processes, the efficiency of communication, and the flexibility in a network environment.

Interprocess communication can be achieved using the following methodss:

1. Implicit communication using a shared database. This method is most efficient but requires explicit cooperation of the processes
2. Generalized communication using mailboxes or DECnet. Mailboxes are virtual devices to which processes can send and from which a process can read messages. DECnet can be employed for interprocess communication in a single node or multinode environment. These methods, however, incur the greatest overhead
3. Shared files

One method of interprocess synchronization is achieved using common event flag clusters. Each cluster contains 32 event flags. A process can wait for another process in the same group to set an event flag, thus indicating that the latter process had performed a function for which the former was waiting. A process can associate with up to two common event flag clusters.

Another method of synchronization is the use of the lock management sevices. Cooperating processes can synchronize access to a resource by queuing lock requests. There are six lock modes, each providing a different level of access-sharing.

VAX/VMS INPUT/OUTPUT

The I/O processing system consists of several interdependent components that enable programmers to choose the appropriate programming interface and processing method. The I/O request processing software takes advantage of the hardware's ability to overlap I/O transfers with computation, switch contexts rapidly, and generate interrupts on multiple priority levels to ensure the maximum possible data throughput and interrupt response.

I/O Interfaces

The I/O programming interfaces are: the record management services (RMS)—for general-purpose file and record processing—and the I/O system services—for direct I/O processing. RMS procedures can be invoked by a user program through high-level language statements such as OPEN, CLOSE, GET, and PUT, or, in VAX-11 MACRO assembler, by a CALL statement. The I/O system services are invoked using a CALL statement.

RMS procedures provide device-independent, file-structured access to all I/O peripherals, whether local or remote in a network. The most general purpose access enables programs to process logical records, where RMS automatically provides logical record blocking and unblocking. RMS users may also perform their own record blocking on file-structured volumes such as disk and magnetic tape, either to control buffer allocation or optimize special record processing.

The I/O system services provide both device-independent and device-dependent programming. Users perform their own record blocking on file-structured and non-file-structured devices. Both virtual block and logical block addressing are possible on file-structured volumes, though the latter requires either privilege or ownership of a private volume. In addition, users with sufficient privilege can perform direct I/O operations using logical block addressing for defining their own file structures and accessing methods on disk and magnetic tape devices.

Both RMS and the I/O system services use the same I/O control processes, called ancillary control processes (ACPs), for processing file-structured I/O requests. An ACP provides file structuring and volume access control for a particular type of device. There are three kinds of ACPs provided in the system: disk, magnetic tape, and network communications link.

I/O Request Processing

All I/O requests are generated by a Queue I/O (QIO) Request system service. If a program calls RMS procedures, RMS in turn calls the QIO system service on the program's behalf. Queue I/O Request processing is extremely rapid because the system can:

- Optimize device unit use by minimizing the code that must be executed to initiate requests and post request completion
- Optimize disk controller use by overlapping seeks with I/O transfers

The processor's many interrupt priority levels increase interrupt response because they enable the software to have the minimum amount of code executing at high priority levels by using low priority levels for code handling request verification and completion notification.

VAX/VMS REALTIME ENVIRONMENT

The VAX hardware and VAX/VMS software have been developed together to insure a superior realtime multitasking computational system. If realtime tasks are to be performed, the following inherent system attributes of the VAX system establish it as an extremely powerful system for the most demanding realtime applications:

- Highly efficient process scheduler providing 16 realtime process priorities
- Rapid process context switching
- Rapid hardware processing of interrupts
- Interrupts vectored to VAX/VMS device drivers
- VAX/VMS operating system support of PDP-11 system peripherals and facilities to enable customers to add support for their own devices
- Ease of use facilities to provide mapping to the I/O page and connection to an interrupt

Because realtime applications are performance sensitive, it is important to provide the application with a direct interface to the innermost core of the operating system services. Figure 1-1 illustrates in layered form the VAX/VMS operating system.

The outer layers of the VAX/VMS operating system are the more sophisticated general purpose features to ensure ease of use and functionality. These layers consist of command procedures, record management services, user programs, etc. The innermost layers constitute the realtime system described above.

I/O DRIVERS

A VAX/VMS device driver is a set of tables and routines that control I/O operations on a peripheral device interfacing to a VAX system. A device driver:

- Defines the peripheral device for the rest of the VAX/VMS operating system
- Defines itself for the operating system procedure that maps and loads the driver and its device database into system virtual memory
- Initializes the device (and/or its controller) at system startup time and after a power failure
- Translates software requests for I/O operations into device-specific commands
- Activates the device

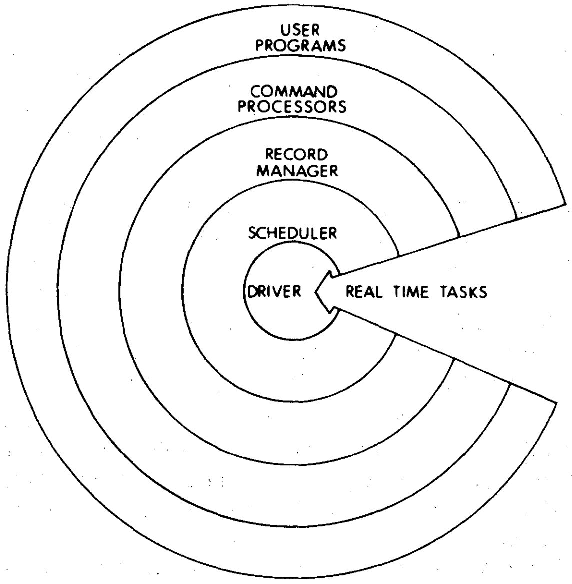


Figure 1-1 VAX/VMS Operating System

- Responds to hardware interrupts generated by the device
- Reports device errors
- Returns data and status from the device to software

When details of an I/O operation need to be translated into terms recognizable by a specific type of device, the operating system transfers control to a device driver. This is known as device-dependent processing. Because different peripheral devices expect different commands and setups, each type of device on a VAX system requires its own supporting driver. The device driver then performs all device-dependent processing. In addition to a wide range of peripherals supported by DIGITAL software, the customer may also develop application-specific device drivers.

COMMUNICATIONS SERVICES

DECnet is the family of DIGITAL's software products, protocols, interfaces, and support services that links DIGITAL computer systems into distributed processing networks. The VAX/VMS operating system offers the same interfaces for use on a single VAX system as DECnet/VAX communications software. Adding the DECnet/VAX software kit to VAX/VMS enables intersystem communication while preserving these interfaces. Therefore, a users application can grow from a single VAX system to a multiple node network, and an existing network can be reconfigured, without necessarily rewriting application programs. The network is transparent to the application programmer. In fact, the applications programmer may treat the networked computers as a common resource.

Using DECnet communications software, various kinds of computer system networks can be constructed to facilitate remote communications, resource sharing, and distributed computation. The DIGITAL Network Architecture (DNA) provides the common network structure upon which all DECnet software products are built. DECnet communications software is highly modular and flexible, and is designed to handle a broad range of application requirements.

DIGITAL's Internet family includes products for batch and interactive communications with computers built by other manufacturers. The Internet products on VAX systems emulate communication protocols recognized and supported by IBM and CDC host processors. Such coexistence features add flexibility to a VAX computer by increasing the number and variety of environments in which it can operate.

PROGRAMMING LANGUAGES

Many major languages are supported under the VAX/VMS operating system, including the FORTRAN, COBOL, BASIC, and PL/I languages. The compilers often offer enhancements to industry standards, while maintaining competitive compile and execution performance.

Applications need not rely on a single language: it is possible to combine several languages, as necessary, for the most efficient accomplishment of computer jobs. Because languages can call one another, the programmer may easily incorporate more than one language in an application program. This means that routines which can be most efficiently accomplished in a particular language can be written in that language and incorporated in applications as needed.

VAX languages available for the VAX/VMS operating system include:

VAX-11 BASIC	VAX-11 BLISS-32
VAX-11 COBOL	VAX-11 BLISS-16
VAX-11 FORTRAN	VAX-11 CORAL 66
VAX-11 PASCAL	VAX-11 DSM
VAX-11 PL/I	VAX-11 MACRO (assembly)
VAX-11 C	

In addition, there is the host development mode programming environment which includes support for PDP-11 FORTRAN IV/VAX to RSX, and MACRO-11 language processors. These language processors produce compatibility mode object code, allowing a VAX computer to "look like" a PDP-11 computer for many types of applications.

VAX PROGRAM DEVELOPMENT TOOLS

The VAX program development tools include text editors, compilers, a librarian, a linker, and the VAX symbolic debugger (DEBUG). Also included are the PATCH, ANALYZ, MESSAGE, and MAIL utilities. All program development utilities can be used either interactively or in batch mode, including the editors and DEBUG.

Libraries may be used extensively for building executable program images. In the native mode programming environment, the programmer can create libraries of assembler macro definitions, of object modules, and of shareable images. The system also includes the common Run-Time Library which provides library functions common to all VAX programming languages.

All program interfaces to the operating system and its utilities have uniform calling standards. System programmers can add new library procedures to the Run-Time Library, installing them online without modifying existing programs and utilities, since all arguments are passed using standard data structures.

User programs can be written to be completely device independent through the system service and command language logical naming facilities. All files and devices can be identified using arbitrarily defined logical names that can be assigned values at runtime.

The program development utilities, with the exception of the editors and the Mail utility, are not available in the host development environment. Many of these utilities are described in more detail in Chapter 4 of this book.

Editors

The programmer can use any or all of the three text editors: EDT, SOS, and SLP. EDT, the DIGITAL Standard Editor, is an interactive editor that enables the programmer to create and modify text files using commands entered from either a hardcopy or video terminal. It allows efficient and powerful character, word, line, and buffer editing. In addition, EDT supports a keypad editor for users of VT100 and VT52 video terminals. A window into the text, coupled with a full range of insertion, deletion, change, and relocation commands, and the capability to move whole text buffers (editing files) into one another make this a very attractive editing tool. Editing procedures (macros and programs) can be written to establish a specialized environment in any editing session. An audit trail file protects the session against accidental loss.

SOS is also an interactive text editor. The user can insert, delete, and replace lines, find and substitute strings, or modify the text a character at a time. Lines can be identified by line number, relative position, or by contents. An adjacent group of lines can be copied or transferred from one place to another. Editing can be done in any order in the file. Editor parameters can be set to user-specified values and the current values can be shown. User-specific parameters can be set automatically at editor startup.

SLP is a programmed text editor that enables a user to modify an existing file by supplying a command file that contains a list of the modifications to be made. The command file provides a reliable way to duplicate the changes made to a file at a later time or on another system. SLP provides a formal record of changes made to files, both in the source file and in an audit trail listing, a feature useful in tracking the stages of large programming projects.

Linker

The VAX/VMS linker accepts one or more native-mode object modules produced by an assembler or compiler, resolves the symbol and procedure references between them, allocates virtual memory, and produces an executable program image.

Unlike many other linkers, however, the VAX/VMS linker also enables a programmer to create shareable images that can be linked subsequently with other modules. Because shareable images are allocated virtual memory by the image activator at runtime, they offer tremendous economy in program development; the shareable image can be modified without having to relink all of the programs that use it.

The linker accepts not only object modules and shareable images as input, but also object module and shareable image libraries.

Librarian

The librarian enables a programmer to create, update, modify, list and maintain library files. A library file can be a collection of object modules, shareable images, macros, or help text. A programmer can request the linker to use one or more library files from which the linker can obtain modules to resolve references during linking.

Common Run-Time Library

The Run-Time Library is a collection of general-purpose and language-specific libraries available to any native program, regardless of the source language in which the program was written. The Run-Time Library allows:

- The choice of incorporating procedures from the library into an executable image, or mapping the global sections into a process virtual address space at runtime
- A single copy of the library to be shared by all processes
- Installation of a new shareable library without the need to relink existing programs

The Run-Time Library includes:

- Mathematical routines (single and double precision trigonometric, logarithmic, and exponential functions)
- Resource allocation routines (virtual memory and dynamic string functions)
- General utility routines (data type conversions)
- Condition handling routines (signaling exception conditions and declaring condition handlers)
- Language-independent support routines (error handling and record management services support functions)
- Several higher-level language-specific support routines (file handling support functions)

Symbolic Debugger

The VAX symbolic debugger (DEBUG) can be linked with a program image to control image execution. DEBUG can be used interactively or it can be controlled from a command procedure file. The debugging language is similar to the VAX/VMS command language. Expressions and data references are generally similar to those of the source language used to create the image being debugged. DEBUG commands allow starting and interrupting program execution, stepping through instruction sequences, calling routines, setting break or trace points, setting default modes, defining symbols, and depositing, examining, or evaluating virtual memory locations.

The symbolic debugger is discussed in more detail in Chpater 8 of this handbook.

PATCH Utility

The image file patch utility (PATCH) provides an extensive set of commands that lets the user make changes directly to the image file and then run the new version without recompiling, reassembling and relinking. PATCH creates a journal file in which all PATCH commands used are recorded. This file provides an easy way to keep track of the changes and attempted changes made to an image file.

PATCH features symbolic referencing of locations, a patch area to store additional data and instructions, and entry and display modes to control the environment in which PATCH accepts commands and displays output.

Object Analyzer Utility

The object module analysis utility checks an object module (or a concatenated file containing several object modules) to see if it is in the correct format for input to the linker. It is a diagnostic tool for writers of compilers or assemblers that generate VAX object code. The program, invoked by the DIGITAL Command Language (DCL) command ANALYZE/OBJECT, can analyze the entire module or only specified types of records. It checks the record type, contents, and sequence of each object module record it examines. The program creates an output file containing a record-by-record analysis of the object module, including identification of any errors in the module.

MESSAGE Utility

The MESSAGE utility allows programmers to construct informational, warning, or error messages in standard VAX/VMS format. First, using a text editor, the programmer creates a source file that specifies the information used in messages, message codes, and message symbols. The MESSAGE command can then be used to compile the source file.

The text displayed can be modified at runtime by using the SET MESSAGE command.

MAIL Utility

The personal mail utility (MAIL) allows users to send messages to each other within the same system or between VAX systems connected via DECnet communications software. With MAIL, users can also file, forward, delete, print, and reply to received messages.

MAIL is invoked with the DCL command MAIL. Messages received are stored in a mail file in a user's default login directory, and new messages are appended to the end of the file. A user can file messages into user-named files with the FILE command, SEARCH for a message containing a specified text string, and request a directory of messages in any of their mail files with the DIRECTORY command.

MAIL broadcasts to a user's terminal when a new mail message has arrived, and indicates who the message is from. Often, users will find MAIL to be a more efficient way reach another user than the telephone.

Command Language Editor (CLE)
The command language editor allows users to modify commands in, or add new commands to, the Command Language Interpreter (CLI) command tables.

CLE is invoked by the DCL command, SET COMMAND.

DATA AND FILE MANAGEMENT UTILITIES

A number of utilities are provided to manage data in files and the files themselves. Included are utilities for manipulation of RMS (Record Management Services) files and verification, manipulation, and back-up of disk volumes.

RMS Utilities
RMS provides the programmer with a File Definition Language (FDL) for defining the attributes of an RMS data file and a number of utilities, including:

ANALYZE /RMS_FILE	Allows the user to check for structure errors in the data file; also can generate a report on data file usage
CONVERT	Copies records from a source data file to a second data file, which can be of a different file organization. It can create a data file from an FDL file
CONVERT /RECLAIM	Reclaims empty buckets in indexed files
CREATE/FDL	Creates an empty data file from an FDL file
EDIT/FDL	Creates and modifies FDL files, and can be used to create an empty data file

RMSSHARE Utility
The RMSSHARE utility performs the following functions:

- It enables the VAX-11 RMS file sharing capability by initializing file sharing structures in system paged dynamic memory, and sets the maximum number of pages that the structures can occupy. The VAX-11 RMS file sharing capability must be enabled each time the operating system is booted
- If VAX-11 RMS file sharing has already been enabled, RMSSHARE displays figures on allowable and actual usage, and permits the resetting of the maximum number of pages that the file sharing structures can occupy

File Transfer Utility (FLX)
The File Transfer utility (FLX) is a utility program that tranfers files from one volume to another. FLX can be used on DOS-11, RT-11, and Files-11 (the file system used on the VAX/VMS operating system) formatted volumes. It converts the format of the files, as appropriate, when transferring files between volumes with different formats. For example, when transferring DOS-11 files to Files-11 volumes, FLX converts the DOS-11 files to Files-11 format.

Bad Block Locator Utility (BAD)
The Bad Block Locator utility (BAD) determines and records the logical block numbers and location of faulty blocks that cannot reliably store data. Usually, BAD is used to test block-structured volumes that have not been initialized. After BAD locates and records the bad blocks, the user issues the DIGITAL Command Language (DCL) command INITIALIZE so that the operating system will allocate the faulty blocks to a special file. This prevents users from accessing these faulty blocks for their files.

File Structure Verification Utility (VERIFY)
This utility is called by the DCL command ANALYZE/DISK-STRUCTURE. It will analyze Files-11 disk structures (both level 1 and level 2) and report errors and inconsistancies. Also, optionally, VERIFY can 1) provide a listing of files in the index file; 2) repair errors it detects in the file structure; 3) selectively repair errors; 4) read check all allocated blocks on the file structure.

SORT/MERGE Utility
The SORT utility rearranges and reformats records in any VAX-11 RMS (Record Management Services) file organization. MERGE is used to combine sorted files.

BACKUP Utility
The BACKUP utility allows users to create back-up copies of files and directories and to restore them. It can back up entire volume sets in one operation or perform selective back-ups by file or date. Wildcarding and several command qualifiers are available for flexible file selection. BACKUP can be used to perform incremental backups of volume sets — a particularly valuable feature for users with large, fixed-media disks.

Other Useful Commands
In addition to the utilities already mentioned, several DIGITAL Command Language (DCL) commands, listed below, aid in data and file management. See Chapter 3 for more information on these commands.

- The COPY command creates a new file from one or more existing files. It can: copy one file to another file, concatenate more than one file into a single output file, and copy a group of files to another group of files
- The CREATE command creates one or more sequential files from records that follow the command in the input stream
- The DELETE command deletes one or more files from a mass storage disk volume
- The DIFFERENCES command compares the contents of two disk files and creates a listing of the records that do not match
- The DIRECTORY command provides a list of files or information about a file or group of files
- The TYPE command displays the contents of a file or group of files on the current output device

SYSTEM MANAGEMENT UTILITIES

At the time a VAX/VMS system is installed, several utility functions are provided to tailor the system for a particular application environment. In addition, once the system is operational, facilities are provided to modify the environment and to upgrade/update the system with new software versions or optional software products.

System Bootstrap Program (SYSBOOT)

In a VAX/VMS system, system generation and start-up occur automatically when the system is bootstrapped. The system manager provides the information needed for system generation and start-up by supplying to SYSBOOT the name of the file that contains the system parameter values and start-up commands.

The SYSBOOT prompt can be requested for commands during the bootstrap operation. If this is done, the system manager can perform the following functions:

- Designate the name of a file that contains system parameter values
- Set and show individual parameter values
- Specify an alternate site-independent start-up command procedure

System Generation Utility (SYSGEN)

The System Generation utility (SYSGEN) allows the system manager to perform a "tailoring" function at system start-up (or later, if required). With SYSGEN, the system manager can:

- Create and modify system parameter files for subsequent bootstrap
- Dynamically modify many current system parameter values
- Create swap, page, and dump files

- Initialize multiport memory
- Dynamically connect devices and load device drivers
- Specify the start-up command procedure

AUTHORIZE Utility
The AUTHORIZE utility is run by the system manager to modify the existing UAF (User Authorization File) or to create a new one. It also allows specification of who may log into the system and permits controls on user's activities.

DISKQUOTA Utility
The DISKQUOTA utility controls the usage of disk volumes. It can be run by the system manager or any user maintaining a volume, and it allows them to create and maintain quota files and set quotas on a per volume basis.

The DISKQUOTA Utility has the following utility functions:

ADD	MODIFY
CREATE	REBUILD
DISABLE	REMOVE
ENABLE	SHOW

INSTALL Utility
The system manager runs the INSTALL utility to install and maintain known images. This enhances performance and permits the sharing of selected executable and shareable images. Another useful function is the ability to install an image with amplified privileges so that the system manager need not give the required privileges to all users of the program.

MONITOR Utility
With the MONITOR utility, the system manager can monitor activities indicative of system performance. Information can be displayed on:

- Network activity
- Use of the lock management services
- Principal users of CPU paging and I/O resources
- File primitive statistics
- I/O system rates
- Time in processor modes
- Page management statistics
- Nonpaged pool statistics

- Number of processes in each scheduler state
- VAX/VMS processes

Upgrade/update
The VMSUPDATE command procedure is used for:

- System upgrade (major releases)
- Maintenance updates
- Installation of optional software

An upgrade/update may only be done by the system manager on a system where there are no users or batch jobs running.

SYE Utility
The SYE utility allows the system manager to selectively report the contents of an error log file. It reports the following information:

- Errors—Device errors, bus errors, synchronous backplane interconnect (SBI) alerts, soft error correcting code (ECC) errors, machine checks, asynchronous write errors, and hard ECC errors
- Configuration changes—Volume mounts and dismounts
- System events—Cold start-up, warm start-up, crash start-up, message from Send Message to Error Logger system service, and time stamp

The types of reports are as follows:

- Totals by category
- Device errors—Contents of device registers
- Brief and standard reports

System Dump Analyzer
The System Dump Analyzer is run by the system manager to aid in determining the cause of operating system failure. It examines and formats contents of system dump files and displays various system data:

- Device data structures
- Memory management data structures
- Process information

CHAPTER OVERVIEW

The programmer and interactive user can find in this chapter how to get the system's attention, how to use some of the command language commands, and how to do program development using the VAX/VMS facilities. In addition, establishing files and assigning logical names for files, devices, and programs are explained. Formats used in later chapters on commands and system services are given here.

Topics include:

- Logging On
- Files and Logical Names
- Program Development Procedures

CHAPTER 2
THE SYSTEM USER

INTRODUCTION

The following sections will discuss basic user-oriented information. These sections include system access, files, logical file names, and program development.

Note that the symbol < > indicates that the user has pressed an action key at the terminal keyboard. For example, <RET> means that the return key is pressed; <DEL> is the delete key; <↑C> is the control/C (CTRL/C) combination.

SYSTEM ACCESS

The user gains the system's attention by pressing the <RET> or CTRL/C. The system responds by prompting for the user's name. Upon entry of a correct user name followed by <RET>, the system prompts for a user password. As the user enters the password, it is not echoed; that is, the password is not displayed on the terminal.

The login sequence appears for a user named GING as follows:

```
<RET>
User_name: GING <RET>
Password: <RET>

Welcome to VAX/VMS Version V3.0
$
```

The $ is a system prompting symbol: when this character appears on the far left of the terminal, the system is ready for command entry.

A default is the user's omission of certain information when entering commands. In the case of a default the system may assume that the omitted names, parameters, and qualifiers have certain values called default values. For example, the system will assume that all of a user's files reside on the default disk unless the user specifies otherwise. Similarly, a user will have a default working set size unless the manager specifically changes it. The use of defaults can simplify and speed up the processes of entering commands, running jobs, and editing files.

Entering Commands

All commands to the system are English-language verbs that describe the functions they perform. For example, the user enters the SHOW TIME command:

```
$ SHOW TIME <RET>
```

The system responds by displaying the current date and time, as follows:

```
22-JUL-1981 10:25:30
```

Commands can be entered using either uppercase or lowercase letters, or a combination of both.

Most commands have parameters and qualifiers. A parameter is the object of a command verb. In the SHOW TIME command above, TIME is a keyword parameter for the SHOW command. Keywords are words that the system recognizes.

As another example, in the following command:

```
$ PRINT MYFILE.LIS<RET>
```

MYFILE.LIS is a parameter for the PRINT command; the command requires the name of a file (MYFILE) and a filetype (.LIS), as explained below.

The user does not have to include the entire command on one line. If a command is entered without required parameters, the system will prompt for additional data. As an example, the print command is entered without the file name qualifiers:

```
$ PRINT<RET>
_file:              MYFILE.DAT<RET>
```

In this example, the filename parameter was omitted; therefore, the system prompted for a file specification parameter.

Qualifiers are keywords that restrict or modify the function of a command. For example, in the following command:

```
$ PRINT/COPIES=2 MYFILE.LIS<RET>
```

/COPIES=2 is a qualifier indicating how many copies of a file to print. Each qualifier in a command must be preceded by a slash character (/).

If the user introduces errors during command input, they may be corrected interactively. The basic line editing functions are:

- <DEL> The delete key deletes and backspaces over characters that have been typed on the current line. In the following example, the first line illustrates user input, while the second line illustrates system echo of the first line (that is, what the user actualy sees typed at hardcopy and some video terminals).

  ```
  $ PRO<DEL>INT MYDA<DEL><DEL>FILE.LIS<RET>
  $ PRO\O\INT MYDA\AD\FILE.LIS
  ```

As far as the command processor is concerned, the line reads perfectly:

```
$ PRINT MYFILE.LIS
```

On some terminals, the key that performs the delete function is marked RUBOUT

- <↑U> The CTRL/U key deletes the current line and performs a carriage return, enabling the user to reenter an entire line
- <↑R> The CTRL/R key performs a carriage return and displays the current line, leaving the print element or cursor at the end of the line permitting continued entry

```
$ PRON\NO\INT MU\U\Y<↑R>
$ PRINT MY
```

- <↑C> The CTRL/C key combination cancels an entire command that was entered on more than one line

  CTRL/C may also be used to interrupt the system during command execution. To terminate an unwanted command during execution, press the CTRL/C or CTRL/Y key and issue the EXIT command as follows:

```
$ TYPE MYFILE.LIS<RET>
      .
      .
      .
<↑C>
$ EXIT <RET>
$
```

The HELP Command

The HELP facility provides information about specific DCL commands. It is accessed interactively from the terminal, which makes it a particular benefit for users who do not have convenient access to a reference manual.

HELP can be used in one of two ways:

1. **query/response mode**. The user may type simply:

   ```
   $ HELP <RET>
   ```

   This will invoke the HELP facility which then displays on the user's terminal a table of all of the primary DCL commands, organized alphabetically, and followed by the querry

   ```
   Topic?
   ```

The user can then select a command from the table—for instance, the PRINT command—and respond

```
Topic? PRINT <RET>
```

The HELP facility will then display information about the PRINT command, what it does and how to invoke it, followed by a list of subtopics including a list of PRINT qualifiers with the defaults indicated by a "(D)". HELP then queries

```
PRINT subtopic?
```

to which the user can respond with one of the listed PRINT subtopics; for example:

```
PRINT subtopic? /AFTER <RET>
```

In response, HELP displays information about the /AFTER qualifier, followed by another query for a PRINT subtopic. The user may then either request another subtopic description or respond with a <RET>.

```
PRINT subtopic? <RET>
```

In this case, HELP returns to the first stage and queries for another HELP topic. Another <RET> response brings the user back to the command level and the dollar sign ($) prompt.

2. **Direct mode.** The experienced user with a specific question might prefer this more direct approach. To find out about a specific topic or subtopic, the entire command can be entered on one line. For example, if the user types

```
$ HELP PRINT/AFTER
```

the resulting display is the same as given for /AFTER response to the "PRINT subtopic" query.

LOGOUT

Upon completing an interactive session, the user must enter the LOGOUT command as follows:

```
$ LOGOUT<RET>
```

The system responds:

```
Username       logged out at      22-APR-1982   11:30:50
```

FILES

A file is a collection of logically related data stored on a medium, such as a disk, tape, or card deck. Many system commands require input files or produce output files. To access files that already exist, or to give names to files that are being created with system commands, the user must know how to identify files.

The system uniquely identifies a file by its file specification (abbreviated "file-spec").

The file is first identified by its location, that is, the actual or physical device on which it is stored.

Because a disk can contain files belonging to many different users, each disk has a set of files called directories. A directory is simply a catalog of a related set of files on that disk.

A complete file specification contains all the information the system needs to know to locate and identify a file. It has the format:

```
device: [directory]filename.filetype;version
```

For example, DMA3:[HANDLE]JEANNE.LIS is a file specification for the directory HANDLE located on an RK07 disk, controller A, unit 3. The file name is JEANNE and the file type is .LIS. See details below.

The punctuation marks (colon, brackets, period, semicolon) are required syntax that separate the various components of the file specification.

When the user logs onto the system, the system assumes all of that user's files reside on a specific disk, alloted to the user by default, called the default disk. The user can determine the current default disk and directory by issuing the SHOW DEFAULT command as follows:

```
$ SHOW DEFAULT
DBA2:[TINKER]
```

This response indicates that the default disk device is DBA2 (an RP06 disk) and the default directory is named TINKER. Often the user's directory name and user name are the same.

File Name and File Type

The user can specify a file uniquely by its file name and file type (or *extension*) as follows:

```
filename.filetype
```

The file name can be from one to nine alphanumeric characters. The alphanumeric characters are A through Z, 0 through 9.

The file type (sometimes called the file extension) can be from one to three alphanumeric characters in length; it must be preceded by a period. By convention, the file type describes more specifically the kind of data in the file. The system recognizes several default file types used for special purposes. For example, among them are:

File Type	**Contents**
.BAS	Input source file for the VAX-11 BASIC compiler
.B32 or BLI	Input source file for the VAX-11 BLISS-32 compiler
.CMD	Compatibility mode command procedure.
.COB	Input source file for the VAX-11 COBOL compiler
.COM	Command procedure file to be executed with the @(Execute Procedure) command, or to be submitted for batch execution with the SUBMIT command
.COR	Input source file for the PDP-11 CORAL 66/VAX compiler
.DAT	Input or Output data file
.DIR	Directory file
.DMP	Output listing created by the DUMP command
.EXE	Executable program image
.FOR	Input file containing source statements for the VAX-11 FORTRAN compiler
.L32	VAX-11 BLISS-32 precompiled library
.LIS	Listing file created by a language compiler or assembler; default input file type for PRINT and TYPE commands
.LOG	Batch job output file
.LST	Compatibility mode listing file
.MAC	MACRO-11 source file
.MAP	Memory allocation map created by the linker
.MAR	VAX-11 MACRO source file
.MLB	Macro library

.OBJ	Object file created by a language compiler or assembler
.OLB	Object module library
.PAS	Input source file for the VAX-11 PASCAL compiler
.R32 or .REQ	VAX-11 BLISS-32 source files required for compilation

Version Numbers
Every file has a version number associated with it, distinguishing different versions of the same file. Each time a file is accessed and modified, the version number is increased by one. The version number is placed after the file type preceded by a semicolon (;) or period (.) as follows:

filename.filetype.version number

or

filename.filetype; version number

Physical Devices
A device name identifies the physical device on which a file is stored. A device name is specified in the format:

dvcu:

where dv is the two-character code for the device type, c is the controller designation, and u is the unit number

Some examples of device names are:

Name	Device
DBA2	RP06 disk on controller A, unit 2
MTA0	TE16 magnetic tape on controller A, unit 0
TTB3	Terminal on controller B, unit 3

If the device name is omitted from a file specification, the system assumes it to be the default disk device.

Among the physical device mnemonics are:

Table 2-1 Device Names

Mnemonic	**Device Type**
CR	Card Reader
CS	Console Device
DB	RP06 Disk
DD	TU58 Tape Cartridge
DL	RL02 Disk
DM	RK07 Disk
DQ	RB02 and RB80 Disks
DR	RM03, RM05, RM80,and RP07 Disks
DY	RX02 Floppy Disk
LA	LPA11-K
LP	Lineprinter
MB	Mailbox
MF	TU78 Magnetic Tape
MS	TS11 Magnetic Tape
MT	TE16, TU77 Magnetic Tapes
OP	Console Terminal
TT	Interactive terminal
XA	DR11-W
XM	DMC-11 Network Link Module

Directories

If the user specifies a file and omits the directory name, the system assumes the file to be in the user's default directory. However, the user may, with privilege, access files in other directories (including directories that catalog files belonging to other users) by specifying the directory name in a file specificaton.

The user may access a file called CUBIT.FOR whose directory name is PERSON by issuing the TYPE command as follows:

```
        $TYPE [PERSON]CUBIT.FOR <RET>
```

This file specification, however, does not contain a device name. Therefore, the system assumes the directory PERSON to be located on the accessing user's default device.

If PERSON's directory were located on disk DBB2, the accessing user would issue the TYPE command as follows:

```
        $TYPE DBB2:[PERSON]CUBIT.FOR <RET>.
```

It is assumed, however, in both cases, that PERSON permitted access to files in the directory by other users. If not, a protection violation error would be returned to the command.

Subdirectories, down to many levels, are possible in the VAX/VMS operating system. This useful feature allows a user to organize a tree structure of subdirectories and catalog files in functional groups.

LOGICAL NAMES

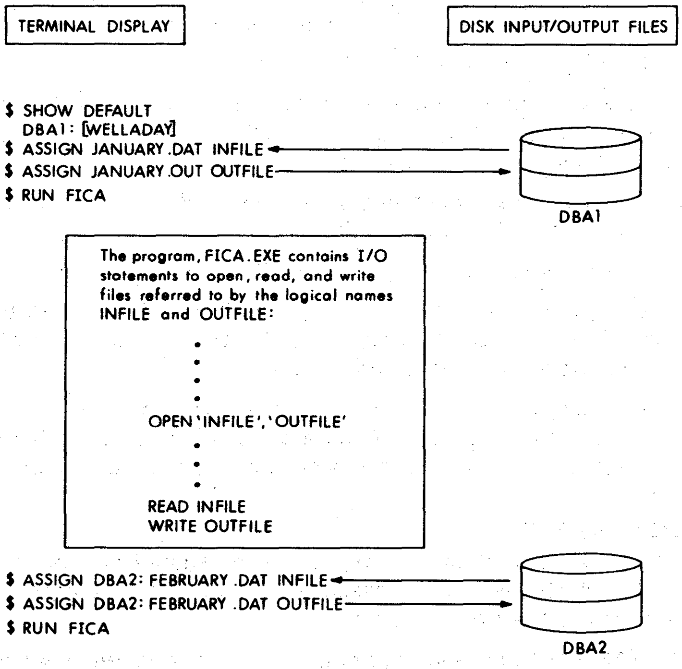
The VAX/VMS operating system provides a generalized logical name capability which permits the association of an arbitrary equivalence string to an arbitrary logical name.

In the VAX/VMS operating system, device independence is accomplished through the use of logical names. During the coding of a program, the user might refer to input and output as INFILE and OUTFILE respectively. INFILE and OUTFILE are logical names. Prior to program execution, the user must associate logical names used in the program with actual files and devices required to run the program.

The ASSIGN command makes this connection: it establishes the correspondence between a logical name (that is, the name used in the program) and an equivalence name (that is, the actual file or device to use).

Figure 2-1 shows how logical names are used. The program FICA contains OPEN, READ, and WRITE statements in a general form; the program reads from a file referred to by the logical name INFILE, and writes to a file referred to by the logical name OUTFILE.

For different runs of the program, the ASSIGN command establishes different equivalence names for INFILE and OUTFILE. In the first example, the program reads the file JANUARY.DAT from the device DBA1 and writes to the file JANUARY.OUT on the same disk device. In the second example, it reads the file FEBRUARY.DAT from the device DBA2 and writes the file FEBRUARY.OUT to that device.



```
$ SHOW DEFAULT
  DBA1: [WELLADAY]
$ ASSIGN JANUARY.DAT INFILE
$ ASSIGN JANUARY.OUT OUTFILE
$ RUN FICA
```

```
The program, FICA.EXE contains I/O
statements to open, read, and write
files referred to by the logical names
INFILE and OUTFILE:
          •
          •
          •
          •
      OPEN 'INFILE','OUTFILE'
          •
          •
          •
      READ INFILE
      WRITE OUTFILE
```

```
$ ASSIGN DBA2: FEBRUARY .DAT INFILE
$ ASSIGN DBA2: FEBRUARY .DAT OUTFILE
$ RUN FICA
```

Figure 2-1   Using Logical Names

System Defined Logical Names
Certain logical names are predefined by the VAX/VMS operating system to provide access to commonly used resources. The major logical names are:

Logical Name	**Equivalence Name**
SYS$INPUT	Default input stream for the process. For an interactive user, SYS$INPUT is equated to the terminal. In a command procedure or batch job, SYS$INPUT is equated to the input file or batch input stream

SYS$OUTPUT	Default output stream for the process. For an interactive user, SYS$OUTPUT is equated to the terminal. In a batch job, SYS$OUTPUT is equated to the batch job log file.
SYS$ERROR	Default device to which the system writes error and event messages. For an interactive user, SYS$ERROR is equated to the terminal. In a batch job, SYS$ERROR is equated to the batch job log file
SYS$COMMAND	Original SYS$INPUT device for an interactive user or batch job
SYS$DISK	Default device established at login, or changed by the SET DEFAULT command

PROGRAM DEVELOPMENT

Four basic steps are required during the course of program development. They are:

- Creating the source program
- Compiling or assembling the source program
- Linking the object module output of a compiler or assembler
- Executing and debugging the program

These steps are common to all of the languages that are available on the VAX/VMS operating system. Figure 2-2 illustrates the necessary steps of program development.

Use the editor to create a disk file containing the source program statements. Specify the name of this file when invoking the compiler or assembler.

The various commands invoke the different language compilers, assemblers, and interpreters that check syntax, create object modules, and if requested, generate program listings.

If a compiler signals any errors, use the editor to correct the source program.

The linker searches the system libraries to resolve references in the object module and create an executable image. Optionally, private libraries can be specified to search, and request the linker to create a storage map of the program.

The RUN command executes a program image. While the program is running, the system may detect errors and issue messages. To determine if the program is error-free, check its output.

If there is a bug in the program, determine the cause of error and correct the source program.

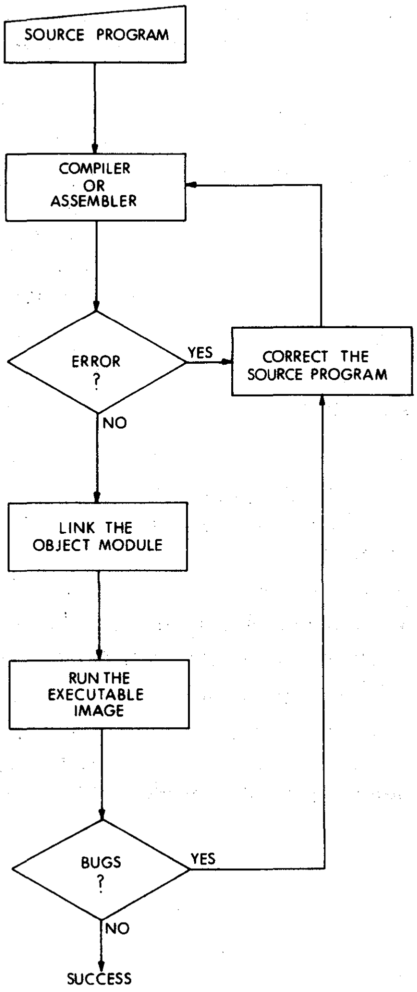


Figure 2-2 Steps in Program Development

Use the editor to create a disk file containing your source program statements. Specify the name of this file when you invoke the compiler or assembler.	SOURCE PROGRAM	
Various commands invoke the different language compilers and assemblers that check syntax, create object modules, and if requested, generate program listings.	COMPILER OR ASSEMBLER	
If a compiler signals any errors, use the editor to correct the source program.	ERRORS? YES	CORRECT THE SOURCE PROGRAM
The linker searches the system libraries to resolve references in the object module and create an executable image. Optionally, you can specify private libraries to search, and request the linker to create a storage map of your program.	NOW LINK THE OBJECT MODULE	

The RUN command executes a program image. While your program is running, the system may detect errors and issue messages. To determine if your program is error-free, check its output.	RUN THE EXECUTABLE IMAGE	
If there is a bug in your program, determine the cause of error and correct the source program.	BUGS	YES
	NO	
	SUCCESS	

Creating the Program
The user must create a file to contain the source program statements. The editor is used to create a file.

Compiling or Assembling the Program
The user must first invoke the compiler or assembler via a command language command.

The compilers check the source program for syntax and programming errors, and then translate the input source file into a binary form that can be interpreted by the computer. The translated code, that is, the object module, is written into a file called an object module file.

Linking the Object Module
An object module is not, in itself, executable; generally, an object module contains references to other programs or routines that must be bound with the object module so that it can be executed. This is the function of the linker.

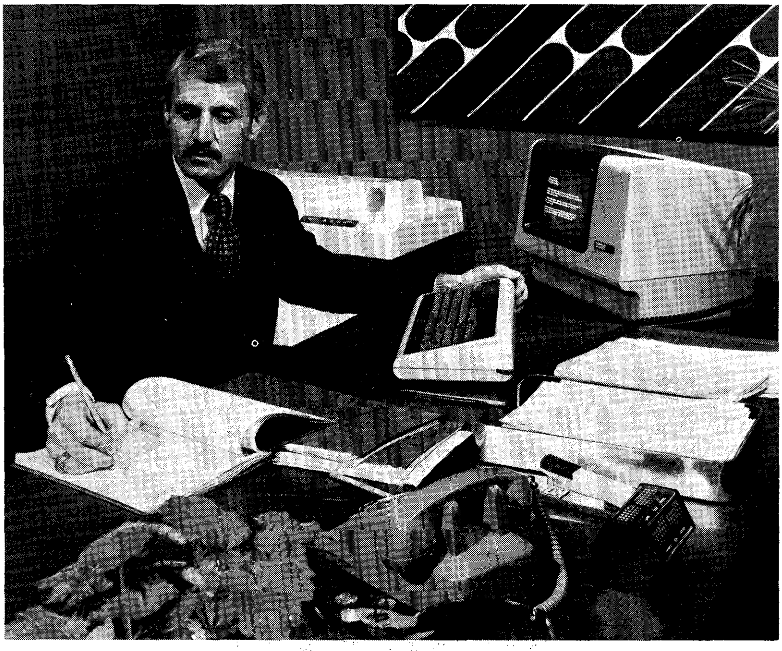
The LINK command invokes the linker. The linker uses system libraries to resolve references to routines or symbols that are not defined within the object modules it is linking. Also, the user can request the linker to include more than one object module as input, or specify user libraries of object modules or shareable images for it to search.

The linker creates an image, which is a file containing the user program in an executable format.

Executing the Program

The RUN command executes an image, that is, it places the image created by the linker into virtual memory so that it can be run.

PART II
PROGRAM DEVELOPMENT



CHAPTER OVERVIEW

The DIGITAL Command Language (called DCL) is a useful tool for establishing and controlling the environment in which a process executes. A command is a request directed to the operating system for a specific action. Frequently used strings of commands can be built into command procedures. This chapter introduces the idea of a command and a command procedure, and shows in some detail how each is used. The formats of many of the DCL commands are listed alphabetically, and examples of some are included. The user will find this chapter helpful when approaching the terminal. Particular attention is paid to the SHOW command.

Topics include:

- Language Name Command Conventions
- Command Procedures
- Commands
- Terminal Function Keys

CHAPTER 3
COMMAND LANGUAGE

INTRODUCTION

A single command language, The DIGITAL Command Language (DCL), provides VAX/VMS users with an extensive set of commands for:

- Interactive program development
- Device and data file manipulation
- Interactive and batch program execution and control

Commands exist for program development and execution, for resource allocation, for environmental control, for job control, for file maintenance, for utilities, and for operational control. Program development and execution commands include commands to invoke each compiler, the assembler, the editor, and the linker, as well as to run any pre-linked program. Resource allocation commands include the ability to allocate and deallocate devices and mount and dismount volumes. Environmental commands include assign and deassign logical names and set and show parameters such as job status, terminal type, and default directory. Job control commands include the ability to continue and stop execution, a GOTO command to transfer control, and IF and ON commands to specify error handling. The VAX/VMS operating system also includes commands to login and logout, to submit batch jobs, to send messages to the operator, and to prompt the user for input. File maintenance commands include append to files, copy, create, and delete files, list directories, initialize volumes, print and type files, and rename files.

COMMAND FORMAT

Commands are composed of English words. Any file name can be given a logical name for mnemonic reference. Command parameters can be supplied on the same line as the command verb. Missing parameters will be prompted for by the VAX/VMS command interpreter. To make it easier to learn the VAX/VMS system, an extensive HELP facility is provided that gives guidance on the use of commands and the meaning of system messages. Typical VAX/VMS commands are brief because of the extensive use of defaults. The user also has the ability to define additional commands and use them just as the system-defined commands are used. In addition, all command verbs and qualifiers can be abbreviated to the shortest unique form.

File specifications can be as simple as the user-given name of the file only, or as complex as a full specification of network node, device (including type, controller, and unit), directory, file name, file type, and version number. Logical names can be defined for complex file specifications so that repetitive typing can be avoided.

The general format of a command is:

$[label:]command-name[qualifiers][parameter-1]...[parameter-n]

where the following rules apply:

1. Dollar Sign $ — The dollar sign [$] must appear in position 1 of a command to be executed in a command procedure. Optionally, it may appear in a command executed in interactive mode.
2. Brackets — In the description of commands in this specification, brackets ([and]) are used to surround optional values. For example:

   ```
   COPY[qualifiers]
   ```

   indicates that the user does not need to supply any qualifiers to issue a valid COPY command.
3. Labels — Any command may be labeled. Labels are used to transfer flow of control via the GOTO command. They may also be used for documentation purposes. The maximum length of a label is 15 characters. A label precedes the command name and is separated from it by a colon (:).
4. Command Names — The command name indicates the action the command is to perform.
5. Qualifiers — A qualifier is used to modify the default action of a command. There are defaults for all qualifiers, i.e., qualifiers are never required. A qualifier always begins with a slash (/). Both command names and parameters can be qualified.

   Examples:

   ```
   PRINT/DELETE     MYFILE.DAT
   SET TERMINAL/LOWERCASE
   ```

   Many qualifiers have associated qualifier values. The qualifier is separated from the qualifier value by an equal sign (=) or a colon (:), e.g., /COPIES=3. Whenever a qualifier requires a list of values, that list must be enclosed in parentheses:

   ```
   /BLOCK=(5,6)
   ```

   A qualifier may not contain any blanks; however, blanks are allowed in qualifier values following left parenthesis, preceding right parenthesis, and before or after a comma. No other blanks are permitted in qualifier values.

Some qualifiers may be negated. When this is permitted, the letters NO prefix the qualifier name.

Example:

```
/OBJECT        produce an object file
/NOOBJECT      do not produce an object file
```

6. Parameters — A parameter either specifies a value that a command is to use when executing, or further defines the action a command is to take. At least one space or tab must separate the first parameter from the command name; parameters are then separated from each other by one or more spaces and/or tabs. Interactive users may supply parameters in response to prompts.
7. Commas and Ellipsis — Some commands permit the user to replace a single parameter by a list of values. When this is done, the items in the list are separated by commas. The commas may, optionally, be surrounded by blanks.

   Examples:

   ```
   DELETE    A,B,C
   ```

   Delete files A, B, and C.

   ```
   COPY    A,B   C
   ```

   Copy files A and B into C.

   In the description of a command's format, ellipsis (three dots ...) indicate that a list of values of the same type may replace a single value.
8. Continuation Character — A hyphen (-), which may optionally be followed by blanks and/or a comment, is used to indicate that a command is to be continued on the next line.

   Example:

   ```
   COPY A.DAT -
   B.DAT
   ```
9. Comment Character — An exclamation mark (!) delimits the start of a comment. Comments can occur only after the last character of a command or after a hyphen. Comments are for the user's information only and do not affect the processing of the command.

Example:

```
COPY A.DAT B.DAT
!FILE A TO FILE B

!COMMAND PROCEDURE FOLLOWS
```

10. Concatenation Character — A plus sign (+) indicates concatenation, that is, the records in the file specified on the left of the plus sign are processed followed by the records in the file specified on the right of the plus sign.

    Example:

    ```
    FORTRAN      A+B
    ```

    The FORTRAN statements in file A.FOR followed by the FORTRAN statements in file B.FOR are read by the FORTRAN compiler to product a single object module, A.OBJ.
11. Lowercase Characters — Lowercase characters will be processed as their uppercase equivalents except for characters within a quoted string. The SET TERMINAL/ [NO]LOWER command controls conversion of characters entered interactively at the terminal; however, it has no effect on data entered via a command procedure.
12. Abbreviation Rule — All command names, qualifiers and parameter keywords can always be abbreviated to the first four letters. The implementation will recognize, in each case, the minimal unique abbreviation. Qualifiers and keywords must be unique only within the command containing them. Additional letters are acceptable, for example, LOGOUT, LOGOU, and LOGO are all correct.
13. End of Data — In interactive mode, CTRL/Z is used to terminate input to a command or a user program, i.e., CTRL/Z will generate an end-of-file.

CONVENTIONS FOR LANGUAGE NAME COMMANDS

1. When the input file specification in a language-name command consists of a list of concatenated files, e.g., A+B+C, then the language processor is invoked once and a single object file is produced. If this object file is not explicitly named, the leftmost file specification will be used for the default. (Note that not all language processors permit the specification of a concatenated list.)

2. When the input file specification in a language-name command consists of a list of file specifications separated by commas—e.g., A, B, C—then the language processor is invoked separately for each file specification and a separate object file is produced for each. If the object files are not explicitly named, the name of the corresponding input file specification is used for the default. A qualifier on a file specification overrides a corresponding qualifier on the command name for that file specification.

   Example:

   ```
   FORTRAN/LIST     A, B/NOLIST, C
   ```

3. In interactive mode, /OBJECT, i.e., produce an object file, and /NOLIST are the defaults. These defaults are also used when a command procedure file is invoked from interactive mode. In batch mode the defaults are /OBJECT and /LIST.

COMMAND PROCEDURES

A command procedure is a file containing VAX/VMS commands and, optionally, data. The commands in a command procedure file are executed when a reference to the command procedure file name appears in interactive mode or in another command procedure file. The syntax is:

```
@file specification
```

The following rules apply:

1. If no file type is given, the default is .COM.
2. Each command in a command procedure file must begin with a dollar sign ($), including further command procedure file references. Lines without the dollar sign leader are interpreted as data lines.
3. A reference to a command procedure must be the rightmost element of the command, and the entire contents of the file are inserted into the command at the point at which the reference was made.

   Examples:

   a. The user types the command:

      ```
      @MYJOB
      ```

      where the file MYJOB.COM contains:

      ```
      $FORTRAN      A
      $LINK         A
      $RUN          A
      ```

b. The user types the command:

```
LINK @LINK_OPT
```

where the file LINK_OPT.COM contains:

```
/IMAGE=JOB1 -
/MAP -
MYJOB, MYDATA
```

indicating that the default image type (.EXE) should be created, overriding the default name of MYJOB to JOB1. A map is explicitly requested with the default to MYJOB, and the object input files are MYJOB and MYDATA.

TERMINAL FUNCTION KEYS

<CR> or RETURN	Carriage return. Transmits the current line to the system for processing
CTRL/X	Cancels type-ahead. Discards any characters that have been typed but not yet read by a program. Also effects a CTRL/U
CTRL/C	Before terminal session, initiates login sequence. During command entry, cancels command processing. Note: Certain system and user programs may provide special routines to handle CTRL/C interrupts. If CTRL/C is pressed to interrupt a program that does not handle CTRL/C, CTRL/C has the same effect as CTRL/Y and echoes at Y.
CTRL/I	Duplicates the function of the TAB key
CTRL/K	Advances the current line to the next vertical tab stop
CTRL/L	Form feed
CTRL/O	Alternately suppresses and continues display of data at the terminal
CTRL/Q	Restarts terminal output that was suspended via CTRL/S
CTRL/R	Retypes the current line during input and leaves the cursor positioned at the end of the line
CTRL/S	Suspends terminal output until CTRL/Q is pressed

CTRL/U	Cancels the current line and discards it
CTRL/Y	Interrupts commands or program execution and returns control to the command interpreter
CTRL/Z	Terminates a file input from the terminal
DELETE or RUBOUT	Deletes the last character entered at the terminal and backspaces over it
ESCAPE or ALTMODE	Have uses pertinent to particular commands or programs

COMMANDS

For the convenience of the user, commands are listed and described below in alphabetical order. Some include detailed examples, particularly control commands for use in command procedures. The on-line HELP facility will provide more detail about most of these commands.

NOTE

This list is not exhaustive. See the VAX/VMS Command Language Users Guide for complete details of commands, options, and defaults.

ALLOCATE

Format:

```
    ALLOCATE      device-name [:]      [logical-name[:]]
```

Purpose:

The ALLOCATE command provides exclusive access to a device and optionally establishes a logical name for the device. Once a device has been allocated, other users cannot access the device until the user specifically deallocates it or logs out.

ANALYZE/CRASH_DUMP

Format:

```
    ANALYZE/CRASH_dUMP      file-spec
```

Purpose:

This command invokes the System Dump Analyzer (SDA) to examine the specified dump file.

ANALYZE/DISK_STRUCTURE

Format:

```
    ANALYZE/DISK_STRUCTURE      device-name
```

Purpose:

ANALYZE/DISK_STRUCTURE invokes the VAX-11 Verify Utility to check the readability and validity of a Files-11 structure disk volume, reporting errors and inconsistencies and, optionally, repairing them.

ANALYZE/OBJECT

Format:

```
ANALYZE/OBJECT file-spec [,...]
```

Purpose:

ANALYZE/OBJECT provides a description of the records comprising an object file or object module library. It also performs a partial error analysis on the file.

ANALYZE/RMS_FILE

Format:

```
ANALYZE/RMS_FILE file-spec [,...]
```

Purpose:

This invokes an RMS utility to inspect and analyze the internal structure of an RMS file. Refer to the description of RMS utilities in Chapter 12 for more details.

ANALYZE/SYSTEM

Format:

```
ANALYZE/SYSTEM
```

Purpose:

This will invoke the System Dump Analyzer (SDA) to examine a running system. In order use this command,you must have the Change Mode to Kernel (CMKRNL)privilege.

APPEND

Format:

```
APPEND   input-file-spec,... output-file-spec
```

Purpose:

The APPEND command adds the contents of one or more specified input files to the end of a specified output file.

ASSIGN

Format:

```
ASSIGN     device-name[:]     logical-name[:]
```

Purpose:

The ASSIGN command equates a logical name to a physical device name, to a complete file specification, or to another logical name, and places the equivalence name string in the process, group, or system logical name table.

BACKUP

Format:

```
BACKUP   input-spec      output-spec
```

The BACKUP command allows users to create backup copies of files and directories and to restore them. It can backup entire volume sets in one operation or perform selective backups by file or date. Wildcarding is available, as well as several file selection qualifiers. BACKUP can also be used to copy entire directory trees (directories and sub-directories).

CANCEL

Format:

```
CANCEL [process-name]
```

Purpose:

The CANCEL command cancels scheduled wakeup requests for a specified process. This includes wakeups scheduled with the RUN command and with the Schedule Wakeup ($SCHDWK) system service.

CLOSE

Format:

```
CLOSE      logical-name
```

Purpose:

The CLOSE command closes a file that was opened for input or output with the OPEN command and deassigns the logical name specified when the file was opened.

Example:

```
    $ OPEN/READ   INPUT_FILE   TEST.DAT
    $ READ_LOOP:
    $                                   READ/END_OF_FILE=
    NO_MORE      INPUT_FILE      DATA_LINE
          .
          .
    $ GOTO READ_LOOP
    $ NO_MORE :
    $ CLOSE INPUT_FILE
```

The OPEN command opens the file TEST.DAT and assigns it the logical name of INPUT_FILE. The /END_OF_FILE qualifier on the READ command requests that when the end of file is reached, the command interpreter transfer control to the line at the label NO_MORE. The CLOSE command closes the input file.

CONTINUE

Format:

```
    CONTINUE
```

Purpose:

The CONTINUE command resumes execution of a DIGITAL Command Language (DCL) command, a program, or a command procedure that was interrupted by pressing CTRL/Y or CTRL/C. The CONTINUE command can also serve as the target command of an IF or ON command in a command procedure, or following a label that is the target of a GOTO command.

Example:

```
    $ RUN     MYPROG
    ↑Y
    $ SHOW TIME
      19-MAR-1980 13:40:12
    $ CONTINUE
```

Note that only a limited set of commands may be executed before a continue command and that a privileged program cannot be continued. The RUN command executes the program MYPROG. While the program is running, pressing CTRL/Y interrupts the image. The SHOW TIME command requests a display of the current date and time. The CONTINUE command resumes the image.

CONVERT

Format:

CONVERT input-file-spec [,...] output-file-spec

Purpose:

This invokes the RMS CONVERT Utility, which copies from one file to another, changing the organization and record format to that of the second file. Refer to the section on RMS Utilties in Chapter 12 for more details.

CONVERT/RECLAIM

Format:

CONVERT/RECLAIM file-spec

Purpose:

This command invokes an RMS Utility to make empty buckets in Pro-log 3 indexed files available so that new records can be written into them. If all the records in the bucket have been deleted, that bucket is locked until CONVERT/RECLAIM makes it available.

COPY

Format:

COPY input-file-spec,... output-file-spec

Purpose:

The COPY command creates a new file from one or more existing files. The COPY command can:

- Copy one file to another file
- Concatenate more than one file into a single output file
- Copy a group of files to another group of files

CREATE

Format:

CREATE file-spec

Purpose:

The CREATE command creates a sequential disk file from records that follow the command in the input stream, or creates a directory file.

CREATE/FDL

Format:

```
        CREATE/FDL=fdl-file-spec [file-spec]
```

Purpose:

This command invokes the RMS utility for creating a new, empty data file based on the specification of an FDL file. This is helpful when creating more than one data file from a particular FDL specification. Refer to Chapter 12 for more details.

DEALLOCATE

Format:

```
        DEALLOCATE      [device-name[:]]
```

Purpose:

The DEALLOCATE command returns a device that was reserved for private use to the pool of available devices in the system.

DEASSIGN

Format:

```
        DEASSIGN      [logical-name[:]]
```

Purpose:

The DEASSIGN command cancels logical name assignments made with the ASSIGN, DEFINE, ALLOCATE, or MOUNT commands.

DEBUG

Format:

```
        DEBUG
```

Purpose:

The DEBUG command invokes VAX DEBUG after program execution is interrupted by CTRL/C or CTRL/Y. The program image being interrupted must contain the debugger; that is, the image was linked with the /DEBUG qualifier and/or run with the /DEBUG qualifier. Notice that DBG> is the DEBUG prompt for a command.

DECK

Format:

```
        DECK
```

Purpose:

The DECK command marks the beginning of an input stream for a command or program. The DECK command is required in command procedures when the first non-blank character in any data record in the input stream is a dollar sign ($).

The DECK command must be preceded by a $. Input records may or may not start with a $.

Example:

```
$ FORTRAN     CERISE
$ LINK     CERISE
$ RUN     CERISE
$ DECK
Input line one...
Input line two...
$ Input line...
    .
    .
    .
$ EOD
$ PRINT SUMMARY.DAT
```

The FORTRAN and LINK commands compile and link program CERISE. When the program is run, any data the program reads from the logical device SYS$INPUT is read from the command stream. The $DECK command indicates that the input stream may contain dollar signs. The $EOD command signals end-of-file for the data.

DEFINE

Format:

```
DEFINE     logical-name     equivalence-name
```

Purpose:

The DEFINE command creates a logical name table entry and assigns an equivalence name string to the specified logical name. The DEFINE command is similar in function to the ASSIGN command; however, its primary purpose is to assign logical name/equivalence name pairs for application-specific uses other than for logical file specification assignments.

DELETE

Format:

```
DELETE     file-spec,...
```

Purpose:

The DELETE command deletes one or more files from a mass storage disk volume.

DELETE/ENTRY

Format:

```
    DELETE/ENTRY=job number,...       queue-name
```

Purpose:

The DELETE/ENTRY command deletes one or more entries from a printer or batch job queue. The /ENTRY qualifier is required.

DELETE/SYMBOL

Format:

```
    DELETE/SYMBOL       symbol-name
```

Purpose:

The DELETE/SYMBOL command deletes a symbol definition from a local symbol table or from the global symbol table, or deletes all symbol definitions in a symbol table. The /SYMBOL qualifier is required.

DEPOSIT

Format:

```
    $ DEPOSIT      location=data,...
```

Purpose:

The DEPOSIT command replaces the contents of a specified location or locations in virtual memory.

The DEPOSIT command, together with the EXAMINE command, aids in debugging programs interactively. The DEPOSIT command is similar to the DEPOSIT command of the VAX-11 Symbolic Debugger.

DIFFERENCES

Format:

```
    DIFFERENCES      input-file-spec      [compare-file-spec]
```

Purpose:

The DIFFERENCES command compares the contents of two disk files and creates a listing of the records that do not match. If no specification for a compare-file is entered, the command uses the next lower version of the input file.

DIRECTORY

Format:

```
    DIRECTORY      [file-spec,...]
```

Purpose:

The DIRECTORY command provides a lists of files or information about a file or group of files.

DISMOUNT

Format:

```
DISMOUNT      device-name[:]
```

Purpose:

The DISMOUNT command releases a volume previously accessed with a MOUNT command.

DUMP

Format:

```
DUMP      file-spec
```

Purpose:

The DUMP command displays or prints the contents of a file or volume in ASCII and a choice of decimal, hexadecimal, or octal data format. The default format is hexadecimal.

EDIT

Format:

```
EDIT/editor      file-spec
```

Purpose:

The EDIT command invokes one of the following VAX/VMS editors:

- EDT
- SOS
- SLP

The default editor is EDT.

EDIT/FDL

Format:

```
EDIT/FDL file-spec
```

Purpose:

EDIT/FDL invokes the file Definition Language (FDL) editor, which allows the user to create and modify FDL files. FDL files provide the specifications for RMS data files. For more information on FDL, refer to Chapter 12 in this book.

EOD

Format:

```
EOD
```

Purpose:

The EOD command signals the end of a data stream when a command or program is reading data from an input device other than an interactive terminal. This command is required only if the DECK command preceded input data in the command stream, or if multiple input files are contained in the command stream without intervening commands. The program or command reading the data receives an end-of-file condition when the EOD command is read.

The EOD command must be preceded by a $; the $ must be in the first character position (column 1) of the input record.

Example:

```
$ RUN     MYPROG
          .
          first data file to be read by the program
          .
$ EOD
          .
          second data file to be read by the program
          .
$ PRINT   TESTDATA.OUT
```

The program MYPROG requires two input files; these are read from the logical device SYS$INPUT. The EOD command signals the end of the first data file and the beginning of the second. The next line that begins with a dollar sign (a PRINT command in this example) signals the end of the second data file.

EOJ

Format:

```
EOJ
```

Purpose:

The EOJ command marks the end of a batch job submitted through the system card reader. An EOJ card is not required; however, if present, the first non-blank character in the command line must be a dollar sign ($). The EOJ command performs the same functions as the LOGOUT command.

EXAMINE

Format:

```
EXAMINE     location[:location]
```

Purpose:

The EXAMINE command displays the contents of virtual memory at the terminal.

Example:

```
$ RUN        MYPROG
↑Y
$ EXAMINE     2678
00002678:     1F4C5026
$ CONTINUE
```

The RUN command begins execution of the image MYPROG.EXE. While MYPROG is running, the CTRL/Y interrupts its execution, and the EXAMINE command requests a display of the contents of virtual memory location hexadecimal 2678.

EXIT

Format:

```
EXIT     [status-code]
```

Purpose:

The EXIT command terminates processing of the current command procedure. If the command procedure was executed from within another command procedure, control returns to the calling procedure.

When typed interactively, the EXIT command may be used to terminate an image interrupted by CTRL/C or CTRL/Y. (See also STOP command.)

In the EXIT command, the image's termination handlers are called, whereas in STOP they are not. The EXIT command is the preferred method of terminating an image interrupted by CTRL/C/CTRL/Y.

Example:

```
$ @SUBTEST
$ IF $STATUS .EQ. 7 THEN GOTO PROCESS
      .
      .
      .
$ EXIT
$ PROCESS:
```

```
              .
              .
```

This procedure executes a second procedure, named SUBTEST.COM. When SUBTEST.COM completes, the outer procedure tests the value of the symbol $STATUS which may be set by SUBTEST as follows:

```
$ PATH1:
    .
    .
$ EXIT 7
$ PATH2:
    .
    .
$ EXIT 9
```

GOTO

Format:

```
GOTO      label
```

Purpose:

The GOTO command transfers control to a labeled statement in a command procedure.

Example:

```
$ IF P1.EQS. "HELP" THEN GOTO TELL
$ IF P1.EQS. "THEN" GOTO TELL
    .
    .
    .
$ EXIT
$ TELL:
$ TYPE SYS$INPUT
    .
    .
    .
```

The IF command checks the first parameter passed to the command procedure; if this parameter is the string HELP or is not specified, the GOTO command is executed, and control is passed to the line labeled TELL. Otherwise, the procedure continues executing until the EXIT command is encountered. At the label TELL, a TYPE command displays data in the input stream that documents how to use the procedure.

HELP

Format:

```
HELP      [keyword [keyword]...]
```

Purpose:

The HELP command displays on the terminal information available in the system HELP files; most notably about how to use DCL commands. See Chapter 2 for more details about the HELP command.

IF

Format:

```
IF   expression   THEN [$]   command
```

Purpose:

The IF command tests the value of an expression and executes a command if the test is true. Any arithmetic or logical expression is considered true if the result of the expression is an odd numeric value; an expression is false if the result is an even value.

Example:

```
$ COUNT = 0
$ LOOP:
$ COUNT = COUNT + 1
      .
      .
      .
$ IF COUNT.LE.10   THEN   GOTO   LOOP
$ EXIT
```

This example shows how to establish a loop in a command procedure using a symbol named COUNT and an IF statement that checks the value of COUNT and performs an EXIT command when the value of COUNT is greater than 10.

INITIALIZE

Format:

```
INITIALIZE   device-name[:]   volume-label
```

Purpose:

The INITIALIZE command formats and writes a label on a mass storage volume.

INQUIRE

Format:

```
INQUIRE     symbol-name     [prompt-string]
```

Purpose:

The INQUIRE command requests interactive assignment of a value for a local or global symbol during the execution of a command procedure.

Example:

```
$ INQUIRE    CHECK "Enter Y[ES] to continue"
$ IF .NOT.CHECK    THEN    EXIT
```

The INQUIRE command displays the following prompting message at the terminal:

```
Enter Y[ES] to continue:
```

The IF command tests the value entered. If it is an odd numeric value or any non-quoted character string that begins with either a "T" or a "Y," the symbol CHECK will be considered true and the procedure will continue executing. If it is an even numeric value, any nonquoted character string that begins with either an "N" or an "F," or a null string, the symbol will be considered false and the procedure will exit.

JOB

Format:

```
JOB     user-name
```

Purpose:

The JOB command identifies the beginning of a batch job submitted through a system card reader.

Example:

```
$ JOB HIGGINS
$ PASSWORD HENRY
$ ON WARNING THEN EXIT
$ FORTRAN SYS$INPUT:AVERAGE
input statements for FORTRAN compiler
      .
      .
      .
$ LINK AVERAGE
$ RUN AVERAGE
data records for program average
      .
      .
      .
$ PRINT AVERAGE
$ EOJ
```

The JOB and PASSWORD cards identify and authorize the user HIGGINS to enter batch jobs. The command stream consists of a FORTRAN command and FORTRAN source statements to be compiled. The file name AVERAGE following the device name SYS$INPUT provides the compiler with a file name for the object and listing files. The output files are cataloged in the user HIGGINS' default directory.

If the compilation is successful, the LINK command creates an executable image, and the RUN command executes it. Input for the program follows the RUN command in the command stream. The last command in the job prints the program listing.

LIBRARY

Format:

```
LIBRARY    library [file-spec,...]
```

Purpose:

The LIBRARY command creates or modifies an object module library or a macro library, or inserts, deletes, replaces, or lists modules, macros, or global symbol names in a library.

LINK

Format:

```
LINK       file-spec,...
```

Purpose:

The LINK command invokes the VAX-11 linker to link one or more object modules into a program image and defines execution characteristics of the image. See Chapter 4 for details about the linker.

LINK/RSX11

Format:

```
LINK/RSX11      file-spec,...
```

Purpose:

The LINK/RSX11 command invokes the RSX-11M task builder to build an RSX-11M image.

MAIL

Format:

```
        MAIL [file-spec]
```

Purpose:

The MAIL command invokes the VAX/VMS personal mail utility. MAIL can be used to correspond with other users on a system or on other VAX systems via DECnet. With the services MAIL provides, the user can:

- Send text, either messages created using MAIL or previously created text files
- Select mail to read
- Delete mail
- File mail in user-named mail files
- Forward mail
- Print mail
- Reply to a mail message
- Peruse a directory of mail messages in the default mail file or one of the user-named mail files
- Edit mail messages with the VAX/VMS editor of the users choice

Also, MAIL will broadcast a message on the receiving user's terminal indicating that new mail has arrived and who it is from.

MCR

Format:

```
        MCR      [component[command-string]]
```

Purpose:

The MCR command provides a means of running RSX-11M components in a manner that is compatible with the RSX-11M operating system.

Examples:

1.
```
$ MCR      DSP      MYFILE.DAT
```
   The MCR command precedes a single RSX-11M command. When the command finishes, DCL prompts for another command.

2.
```
$ MCR
MCR>PIP    MYFILE.DAT/SP
MCR>↑Z
$
```

The MCR command requests activation of MCR command mode. The MCR> prompt indicates that the MCR command interpreter is ready to accept commands. After the PIP command executes, MCR continues prompting until CTRL/Z is used to return to DCL.

MONITOR

Format:

```
MONITOR class-name
```

Purpose:

MONITOR is a VAX/VMS utility for monitoring operating system performance. It collects system performance data by class and produces two forms of optional output:

- a recording file
- statistical terminal display

For more information about the MONITOR Utility, refer to Chapter 16 of this book.

MOUNT

Format:

```
MOUNT  device-name,...  [volume-label,...][logical-name[:]]
```

Purpose:

The MOUNT command makes a volume and the files or data it contains available for processing by system commands or user programs.

ON

Formats:

```
ON     severity-level     THEN     [$] command
ON     CONTROL_Y          THEN     [$] command
```

Purpose:

The ON command defines the default courses of action when a command or program executed within a command procedure 1) encounters an error condition or 2) is interrupted by CTRL/Y. The specified actions are taken only if the command interpreter is enabled for error checking or CTRL/Y interrupts; these are the default conditions.

Examples:

1.
```
$ ON    ERROR    THEN    GOTO    BYPASS
$ RUN A
$ RUN B
    .
    .
    .
$ EXIT
$ BYPASS:      RUN C
```

If either program A or program B returns a status code with severity level of error or severe error, control is transferred to the statement labeled BYPASS.

2.
```
$ ON    CONTROL_Y    THEN    GOTO    CRTL_EXIT
        .
        .
        .
$ CTRL_EXIT
$ CLOSE INFILE
$ CLOSE OUTFILE
$ EXIT
```

The ON command specifies action to be taken when CTRL/Y is pressed during the execution of this procedure. When CTRL/Y is pressed, the GOTO command that transfers control to the line labeled CTRL_EXIT is executed. At this label, the procedure performs clean-up operations, in this example, closes files and exits.

OPEN

Format:

```
OPEN      logical-name      file-spec
```

Purpose:

The OPEN command opens a file for reading or writing at the command level.

Example:

```
$ OPEN    INPUT_FILE    AVERAGE.DAT
$ READ_LOOP:
$ READ/END_OF_FILE=ENDIT      INPUT_FILE      NUM
        .
        .
        .
$ GOTO    READ_LOOP
$ ENDIT:
$ CLOSE    INPUT_FILE
```

The OPEN command opens the file named AVERAGE.DAT as an input file and assigns it the logical name INPUT_FILE. The READ command reads a record from the logical file INPUT_FILE into the symbol named NUM. The procedure executes the lines between the labels READ_LOOP and ENDIT until the end of the file is reached. At the end of the file, the CLOSE command closes the file.

PASSWORD

Format:

```
PASSWORD      password
```

Purpose:

The PASSWORD command specifies the password associated with the user name specified on a JOB card for a batch job submitted through the system card reader.

Example:

```
$ JOB      JOHN
$ PASSWORD      BYRON
     .
     .
     .
$ EOJ
```

The JOB and PASSWORD commands precede a batch job submitted from the card reader. An EOJ command marks the end of the job.

PHONE

Format:

```
PHONE [phone-comand]
```

This invokes the VAX/VMS PHONE Utility, which allows users on a system to "talk" via their terminals to one another or to any user on another VAX System connected by DECnet/VAX.

PRINT

Format:

```
PRINT      file-spec,...
```

Purpose:

The PRINT command queues one or more files for printing, either on a default system printer or on a specified device.

PURGE

Format:

```
PURGE        file-spec,...
```

Purpose:

The PURGE command deletes all but the highest numbered version or versions of a specified file or files.

READ

Format:

```
READ       logical-name       symbol-name
```

Purpose:

The READ command reads a single record from a specified input file and assigns the contents of the record to a specified symbol name.

Example:

```
$ OPEN IN NAMES.DAT
$ LOOP:
$ READ/END_OF_FILE=ENDIT IN NAME
      .

      .
$ GOTO LOOP
$ ENDIT:
$ CLOSE IN
```

The OPEN command opens the file NAMES.DAT for input and assigns it the logical name of IN. The READ command specifies the label ENDIT to receive control when the last record in the file has been read. The procedure loops until all records in the file have been processed.

RENAME

Format:

```
RENAME      input-file-spec      output-file-spec
```

Purpose:

The RENAME command changes the directory name, file name, file type, or file version of an existing disk file.

REQUEST

Format:

```
REQUEST      message-text
```

Purpose:

The REQUEST command displays a message at a system operator's terminal, and optionally requests a reply. System operators are identified by the function(s) they perform; if more than one operator is designated for a particular function, all receive the specified message.

RUN (Image)

Format:

```
RUN      file-spec
```

Purpose:

The RUN command places an image into execution in the process.

SEARCH

Format:

```
SEARCH file-spec [,...] string
```

Purpose:

The SEARCH command allows users to search one or more file for an occurance of a specified string. It will list all occurances on the user's terinal or, optionally, in an output file.

SET

Format:

```
SET      option
```

where the options are

```
CARD_READER
COMMAND
[NO]CONTROL_Y
DEFAULT
MAGTAPE
MESSAGE
[NO]ON
PROCESS
PROTECTION
QUEUE
RMS_DEFAULT
TERMINAL
[NO]VERIFY
WORKING_SET
```

Purpose:

The SET command defines or changes, for the current terminal session or batch job, characteristics associated with files and devices owned by the process.

1) SET CARD_READER

Format:

```
SET CARD_READER        device-name
```

Purpose:

The SET CARD_READER command defines the default translation mode for cards read into a system card reader. All subsequent input read into the specified card reader will be converted using the specified mode.

SET COMMAND

Format:

```
SET COMMAND file-spec [,...]
```

Purpose:

SET COMMAND invokes the VAX-11 Command Definition Utility to add commands that are defined in the specified command description file to your process command set or a command tables file.

2) SET CONTROL_Y

Format:

```
SET [NO]CONTROL_Y
```

Purpose:

The SET CONTROL_Y command controls whether the command interpreter receives control when CTRL/Y is pressed.

3) SET DEFAULT

Format:

```
SET DEFAULT        device-name
```

Purpose:

The SET DEFAULT command changes the default device and/or directory name for the current process. The new default is applied to all subsequent file specifications that do not explicitly give a device or directory name.

When the default device assignment is changed, the system equates the specified device with the logical name SYS$DISK.

4) SET MAGTAPE

Format:

```
SET MAGTAPE       device-name[:]
```

Purpose:

The SET MAGTAPE command defines the default characteristics associated with a specific magnetic tape device for subsequent file operations. The SET MAGTAPE command is valid for tape devices that do not currently have volumes mounted on them, or on which foreign volumes are mounted.

5) SET MESSAGE

FORMAT:

```
SET MESSAGE
```

Purpose:

The SET MESSAGE command allows selection of which fields of the message get printed.

6) SET ON

FORMAT:

```
SET [NO]ON
```

Purpose:

The SET ON command controls whether the command interpreter performs error checking following the execution of commands in command procedures.

Example:

```
$ SET NOON
$ DELETE   *.SAV;*
$ SET ON
$ COPY    *.OBJ *.SAV
```

This command procedure routinely copies all object modules into new files with file types of .SAV. The DELETE command deletes all existing files with that file type, if any. The SET NOON command ensures that the procedure will continue execution if there are not currently any files with that file type. Following the DELETE command, the SET ON command restores error checking. Then the COPY command makes copies of all existing files with file types of .OBJ.

7) SET PROCESS

Format:

```
SET PROCESS        [process-name]
```

Purpose:

The SET PROCESS command changes execution characteristics associated with a process for the current terminal session or job.

8) SET PROTECTION

Format:

```
SET PROTECTION[=code]       [file-spec,...]
```

Purpose:

The SET PROTECTION command establishes the protection to be applied to a particular file or a group of files, or establishes the default protection for all files subsequently created during the terminal session or batch job. The protection for a file limits the type of access available to other system users.

9) SET QUEUE/ENTRY

Format:

```
SET QUEUE/ENTRY=jobid       [queue-name]
```

Purpose:

The SET QUEUE command changes the current status or attributes of a file that is queued for printing or for batch job execution but not yet processed by the system.

10) SET RMS_DEFAULT

Format:

```
SET RMS_DEFAULT
```

Purpose:

The SET RMS_DEFAULT command defines default values for the multiblock and multibuffer counts used by the VAX-11 RMS (Record Management Services) for file operations. Defaults can be set for sequential or relative files on a process-only or system-wide basis.

11) SET TERMINAL

Format:

```
SET TERMINAL       [device-name]
```

Purpose:

The SET TERMINAL command changes the characteristics of a specified terminal.

12) SET VERIFY

Format:

```
SET [NO]VERIFY
```

Purpose:

The SET VERIFY command controls whether or not command lines in command procedures are displayed at the terminal or printed in a batch job log.

Example:

```
$ SET VERIFY
      .
      .
      .
$ SET NOVERIFY
$ EXIT
```

The verification setting is turned on for the execution of a command procedure. The system displays all the lines in the procedure, including command lines, as it reads them. At the end of the procedure, the SET NOVERIFY command restores the system default.

13) SET WORKING_SET

Format:

```
SET WORKING_SET
```

Purpose:

The SET WORKING_SET command redefines the default working set size for the process or sets an upper limit to which the working set size can be changed by an image that the process executes.

SHOW

Format:

```
SHOW     option
```

Options
[DAY]TIME
DEFAULT
DEVICES
LOGICAL

```
        MAGTAPE
        NETWORK
        PRINTER
        PROCESS
        PROTECTION
        QUEUE
        RMS_DEFAULT
        STATUS
        SYMBOL
        SYSTEM
        TERMINAL
        TERMINAL PERMANENT
        TRANSLATION
        WORKING_SET
```

Purpose:

The SHOW command displays information about the current status of the process, the system, or devices in the system.

1) SHOW DAYTIME

Format:

```
        SHOW      [DAY]TIME
```

Purpose:

The SHOW DAYTIME command displays the current date and time in the default output stream.

2) SHOW DEFAULT

Format:

```
        SHOW DEFAULT
```

Purpose:

The SHOW DEFAULT command displays the current default device and directory name. These defaults are applied whenever a device and/or directory name from a file specification is omitted.

The default disk and directory are established in the User Authorization File. They can be changed during a terminal session or in a batch job with the SET DEFAULT command, or by reassigning the logical name SYS$DISK.

3) SHOW DEVICES

Format:

```
        SHOW DEVICES
```

Purpose:

The SHOW DEVICES command displays the status of all devices in the system, the status of a particular device, or lists the devices that currently have volumes mounted on them and/or are allocated to processes.

The information displayed includes:

- Device name
- Device status (indicates whether the device is online)
- Device characteristics (indicates whether the device is allocated or spooled, has a volume mounted on it or has a foreign volume mounted on it)
- Error count
- Volume label (for disk and tape volumes only)
- Number of free blocks on the volume
- Transaction count
- Number of mount requests issued for the volume (disk devices only)

4) SHOW LOGICAL

Format:

```
        SHOW LOGICAL      [logical-name[:]
```

Purpose:

The SHOW LOGICAL command displays all logical names in one or more logical name tables; or displays the current equivalence name assigned to a specified logical name by the ASSIGN, ALLOCATE, DEFINE, or MOUNT commands.

5) SHOW MAGTAPE

Format:

```
        SHOW MAGTAPE      device-name[:]
```

Purpose:

The SHOW MAGTAPE command displays the current characteristics and status of a specified magnetic tape device, including device type, density, and format.

6) SHOW NETWORK

Format:

```
        SHOW NETWORK
```

Purpose:

The SHOW NETWORK command displays the availability of the local node as a member of the network and the names of all nodes that are currently accessible by the local node.

7) SHOW PRINTER

Format:

```
        SHOW PRINTER        [device-name[:]]
```

Purpose:

The SHOW PRINTER command displays the default characteristics currently defined for a system printer; for example, tthe device type, column-width, lines per page, and if it is currently spooled to any device.

8) SHOW PROCESS

Format:

```
        SHOW PROCESS
```

Purpose:

The SHOW PROCESS command displays information about the current process, including:

- Date and time the SHOW PROCESS command is issued
- Device name of the current SYS$INPUT device
- User name
- Process identification number
- Process name
- User identification code (UIC)
- Base execution priority
- Default device
- Default directory
- Devices allocated to the process and volumes mounted, if any

9) SHOW PROTECTION

Format:

```
        SHOW PROTECTION
```

Purpose:

The SHOW PROTECTION command displays the current file protection to be applied to all new files created during the terminal session or batch job. The default protection can be changed at any time with the SET PROTECTION command.

10) SHOW QUEUE

Format:

```
SHOW QUEUE      [queue-name[:]]
```

Purpose:

The SHOW QUEUE command displays the current status of entries in the printer and/or batch job queues.

11) SHOW QUOTA

Format:

```
SHOW QUOTA
```

Purpose:

Displays the current disk quota that is authorized and used by a specific user on a specific disk.

12) SHOW RMS_DEFAULT

Format:

```
SHOW RMS_DEFAULT
```

Purpose:

The SHOW RMS_DEFAULT command displays the current default multiblock count and multibuffer count that VAX-11 RMS (Record Management Services) uses for file operations.

13) SHOW STATUS

Format:

```
SHOW STATUS
```

Purpose:

The SHOW STATUS command displays the status of the image currently executing in the process, if any. The SHOW STATUS command is issued after execution has been interrupted by entering a CTRL/C. It does not affect the image; execution of the image can be continued after displaying its status.

The information displayed by the SHOW STATUS command includes:

- Current time and date
- Elapsed CPU time used by the current process
- Number of page faults
- Open file count
- Buffered I/O count
- Direct I/O count

- Current working set size
- Current amount of physical memory occupied

14) SHOW SYMBOL

Format:

```
SHOW SYMBOL       symbol-name
```

Purpose:

The SHOW SYMBOL command displays the current value of a local or global symbol. Symbols are defined with assignment statements (= command), by passing parameters to a command procedure file, or by the INQUIRE or READ commands.

15) SHOW SYSTEM

Format:

```
SHOW SYSTEM
```

Purpose:

The SHOW SYSTEM command displays a list of processes in the system and the following information about the status of each:

- Process identification
- Process name
- User identification code
- Process state
- Current priority
- Direct I/O count*
- Elapsed CPU time*
- Number of page faults*
- Physical memory occupied*
- Process indicator**

* This information is displayed only if the process is currently in the balance set; if the process is not in the balance set, these columns contain the message:

```
-- swapped out --
```

** The letter B indicates a batch job; the letter S indicates a subprocess; the letter N indicates a network process.

16) SHOW TERMINAL

Format:

```
SHOW TERMINAL      [device-name]
```

Purpose:

The SHOW TERMINAL command displays the current characteristics of a specific terminal. Each of these characteristics can be changed with a corresponding option of the SET TERMINAL command.

17) SHOW TRANSLATION

Format:

```
    SHOW TRANSLATION      logical-name
```

Purpose:

The SHOW TRANSLATION command searches the process, group, and system logical name tables, in that order, for a specified logical name and returns the equivalence name of the first match found.

18) SHOW WORKING_SET

Format:

```
    SHOW WORKING_SET
```

Purpose:

The SHOW WORKING_SET command displays the working set quota and limit assigned to the current process.

SORT

Format:

```
   SORT        input-file-spec        output-file-spec
```

Purpose:

The SORT command invokes the VAX SORT/MERGE utility to reorder the records in a file into a predefined sequence, and to create either a new file of the reordered records or an address file by which they can be accessed.

If SORT/RSX11 is used, the PDP-11 SORT utility is invoked.

STOP

Format:

```
STOP     [process-name]
```

Purpose:

The STOP command terminates execution of:

- A command, image, or command procedure that was interrupted by CTRL/Y

- A command procedure
- A subprocess or a detached process

See also EXIT command.

Example:

```
0.  $ ON ERROR THEN STOP
         .
         .
         .
```

In a command procedure, the ON command establishes a default action when any error occurs as a result of a command or program execution. The STOP command stops all command levels: if this ON command is executed in a command procedure that is executed from within another procedure, control does not return to the outer procedure, but to the command interpreter.

SUBMIT

Format:

```
SUBMIT      file-spec,...
```

Purpose:

The SUBMIT command enters a command procedure in the batch job queue.

SYNCHRONIZE

Format:

```
SYNCHRONIZE      [job-name]
```

Purpose:

The SYNCHRONIZE command places the process issuing this command in a wait state until a specified batch job completes execution.

Example:

```
$    SUBMIT/NAME=PREP    FORMAT/PARAMETERS=
(SORT,PURGE)
$ SUBMIT PHASER
```

The first SUBMIT command submits the command procedure FORMAT.COM for execution and gives the job the job name PREP. The second SUBMIT command queues the procedure PHASER.COM. The procedure PHASER.COM contains the line:

```
$ SYNCHRONIZE    PREP
```

When this line is processed, the system verifies whether the batch job name PREP is currently executing. If it is, the procedure PHASER is forced to wait until PREP completes execution.

TYPE

Format:

```
        TYPE      file-spec,...
```

Purpose:

The TYPE command displays the contents of a file or group of files on the current output device.

UNLOCK

Format:

```
        UNLOCK      file-spec,...
```

Purpose:

The UNLOCK command makes accessible a file that became inaccessible as a result of being improperly closed. This can only happen with compatibility mode images.

WAIT

Format:

```
        WAIT      delta-time
```

Purpose:

The WAIT command places the current process in a wait state until a specified period of time has elapsed. The WAIT command is provided for use in command procedures to delay processing of the procedure or of a set of commands in a procedure for a specific amount of time.

Example:

```
        $ LOOP:
        $ RUN    ALPHA
        $ WAIT 00:10
        $ GOTO    LOOP
```

The command procedure executes the program image ALPHA. After the RUN command executes the program, the WAIT command delays execution of the next command for 10 minutes. After 10 minutes, the GOTO command executes the program again. The procedure loops until interrupted or terminated.

If the procedure is executed interactively, it can be terminated by pressing CTRL/C or CTRL/Y and issuing the EXIT command or another DIGITAL Command Language command that runs a new image in the process. If the procedure is executed in a batch job, it can be terminated with the DELETE/ENTRY command.

WRITE

Format:

```
        WRITE      logical-name      data,...
```

Purpose:

The WRITE command writes a record to a specified output file.

Example:

```
        $ WRITE    SYS$OUTPUT "Beginning second phase of tests"
```

The WRITE command writes a single line of text to the current output device. This command is particularly useful for displaying information on the terminal from a command procedure.

CHAPTER OVERVIEW

Special notice needs to be given to some of the important programming support facilities of a VAX/VMS operating system. Three text editors—interactive and batch—are described with examples in this chapter. The linker, a crucial VAX/VMS utility, is explained. VAX DEBUG can help programmers step through their code to detect and correct logical and coding errors. The extensive VAX Run-Time Library, which holds coded algorithms ready for linking into user processes, is treated, as is the DIGITAL Standard RUNOFF utility.

Topics include:

- Text Editors
- The Linker
- VAX DEBUG
- The VAX Run-Time Library
- The DIGITAL Standard RUNOFF Utility
- VAX-11 SORT/MERGE
- Optional Code Management System

CHAPTER 4
PROGRAMMING SUPPORT FACILITIES

INTRODUCTION

The VAX/VMS operating system provides a complete program development environment for the user. Development tools supporting this environment are interactive and batch text editors, a linker, a librarian, an interactive program debugger, and the differences utility. These tools, as well as program language compilers, are available to the programmer via the command language facility.

The text editors can be used to create memos, documentation, and text and data files, as well as source program modules for any language processor. The linker, librarian, debugger, and Run-Time Library are used only in conjunction with the language processors that produce native code. Each of the support utilities is described below, with the exception of the librarian, which is discussed in Chapter 3, The DIGITAL Command Language.

TEXT EDITORS

The VAX/VMS operating system supports three text editors: two interactive text editors (EDT and SOS) and a batch-oriented text editor (SLP). Text editing refers to the process of creating, modifying, and maintaining files.

The user invokes the EDT and SOS text editors interactively with the computer system. That is, the user creates and processes files online. The SLP text editor, on the other hand, allows direct modification to an information file via an instruction file prepared by the user. In addition, SLP generates a formal and complete record of changes to a file including time of occurrence. SOS is often used to create SLP command files. All editors are invoked by the command language command EDIT. The default editor is EDT. Therefore, to invoke EDT, enter EDIT or EDIT/EDT; to invoke SOS, enter the command EDIT/SOS; to invoke SLP, enter EDIT/SLP.

THE EDT EDITOR

EDT, the DIGITAL standard Editor, lets users enter and manipulate text and programs. It is used to view and modify a file directly.

With its extensive HELP facility, the EDT editor is designed to be learned by novices. In addition, it provides many capabilities that will appeal to advanced users.

What EDT Does

EDT is a powerful text editor that provides:

- Both line and character editing facilities
- Screen editing using the keypad on VT52 and VT100 video terminals
- The ability to work on multiple files simultaneously
- A journaling facility which protects against loss of edits due to system crashes
- An extensive HELP facility
- A default start-up command file, which allows a choice of editing options to be set automatically
- A window into a file (on VT52 and VT100 terminals only) that lets users view changes in buffer contents immediately
- Shareable installation for multiple users

EDT is also supported on hardcopy terminals, but it does not provide the window on a file.

Buffers

All editing with EDT is done using 'buffers'. A buffer is a part of EDT's memory that can hold an essentially unlimited amount of text. When a user begins editing, the input file is read into the MAIN buffer, and when editing is complete, the MAIN buffer is written onto the disk as a file. Thus, editing in the MAIN buffer is like editing a file directly.

Editing with a Window

'Window editing' is a valuable feature that lets the user edit one 22-line window (screenful) at a time. This feature allows the user to see immediately how the edits made affect the buffer. The window moves through the buffer so that the cursor is always visible.

Start-Up File

When a user starts EDT, the editor checks to see if the user created a start-up file. Editing options, such as SET MODE CHANGE and DEFINE KEY, can be inserted in the start-up file. These options take effect automatically when an editing session begins.

HELP Facilities

The HELP facilities on EDT are extensive. A user can get help on general EDT operations by typing HELP. If help is needed while in keypad mode, pressing the help key displays information that is specific to keypad editing. The help information is tree-structured, so that more specific help can be obtained on a general topic.

Redefining Keypad Keys

A user can redefine any of the keypad keys and most of the control (CTRL) keys on VT52 and VT100 terminals. With this feature, a series of commands can be assigned to a key. EDT then performs these commands when the key is pressed.

The SET and SHOW Commands

The SET command, with a variety of qualifiers, affects EDT's editing capabilities. SET controls screen parameters such as line width. SET also lets the user determine the appearance of text, such as changing the window size to less that 22 lines. The SHOW command provides information on the current state of the editor, such as terminal parameters, definitions of keypad keys, and the names of buffers in use during an editing session.

Journal Processing

Journal processing protects the user's work against system crashes. During an editing session, EDT saves all the input from a terminal in a journal file. After a crash and recovery, the user can retrieve and execute commands in this saved file with the /RECOVER option. In this way, a file can be recovered to nearly the time of the crash.

EDT MODES OF OPERATION

Keypad and Line Editing

With EDT there is a choice of keypad or line mode editing. They allow the user to:

- Display a range of lines
- Find, substitute, insert, and delete text
- Move, copy, and renumber lines
- Copy text into a buffer and write it on files
- Define the functions of keys

Keypad editing is available on VT52 and VT100 terminals. The group of keys at the right of the keyboard are used to enter keypad functions.

Keypad editing is powerful and versatile, yet it is easy to learn and use. In keypad editing, the active buffer is displayed on the screen as the user edits. There is a wide variety of keypad editing functions, each of which requires that only one or two keypad keys be pressed to perform a function. The user enters commands, inserts text, and performs CONTROL functions from the keyboard.

Line editing is useful for those users who have hardcopy terminals or who prefer editing by numbered lines. In line editing, all entries are made from the keyboard. As the user makes changes to the contents of the buffer, EDT displays one or more lines at a time.

Keypad Layout

Keypad functions allow the user to perform a variety of operations with a single keystroke. Using the DEFINE KEY command, the function of any keypad key can be changed. The Figure 4-1 shows the default keypad for the VT100:

Gold	Help	Fndnxt Find	Del L Und L
Page Command	Sect Fill	Append Replace	Del W Und W
Advance Bottom	Backup Top	Cut Paste	Del C Und C
Word Chngcase	Eol Del Eol	Char Specins	Enter Subs
Line Open Line		Select Reset	

Figure 4-1 VT100 Keypad

Backspace	Go to the beginning of line
Delete	Delete character
Linefeed	Delete to start of word
CTRL/A	Compute tab level
CTRL/D	Decrease tab level
CTRL/E	Increase tab level
CTRL/K	Define key
CTRL/T	Adjust tabs
CTRL/U	Delete to start of line
CTRL/W	Refresh screen
CTRL/Z	Return to line mode

The commands in keypad editing let the user alter or change the cursor position in the buffer. Some of the keypad functions let the user advance or back up the cursor to the top or bottom of the text. The cursor may also be moved any number of characters, words, lines, or pages at a time.

Keypad keys let the user select a string of text and move it elsewhere in any of the user's buffers. The next occurence of a certain piece of text can be found and deleted or replaced. There is also a key to press for help messages.

A SAMPLE SESSION WITH EDT

To begin an editing session with EDT, the user logs in and types EDIT or EDIT/EDT. A prompt appears to let the user start the editor:

```
$ EDIT <RET>
$-File:
```

Creating a File

To create a file, type the file name after the prompt:

```
$-File: TEST<RET>
```

EDT notifies the user that no file with that name exists by responding:

```
Input file does not exist
[EOB]
*
```

[EOB] means the "End of the Buffer" in the MAIN buffer. The asterisk prompt indicates that EDT is in line mode. When EDT is in line mode, the buffer can be edited by individual lines.

Entering Text in Line Mode

The first entry in the buffer is an insertion. When "i" is typed to insert and then the RETURN key is pressed, any text entered is indented two tab spaces.

```
*i<RET>
                    'Twas brillig, and the slithy toves<RET>
                    Did gyre and gimble in the wabe;<RET>
                    All mimsy were the borogoves,<RET>
                    And the mome raths outgrabe<RET>
                    ↑Z
[EOB]
*
```

↑Z (CTRL/Z) is typed to save insertions.

Range Specifications in Line Mode

A range specification expresses the part of the buffer on which a command is to operate. There are various ways of expressing range specifications in line mode. Some examples follow:

1. Type the whole buffer.

```
*t w<RET>
  1                 'Twas brillig, and the slithy toves
  2                  Did gyre and gimble in the wabe;
  3                 All mimsy were the borogoves,
  4                 And the mome raths outgrabe.
[EOB]
*
```

2. Type the second line.

```
*2<RET>
  2                  Did gyre and gimble in the wabe;
*
```

3. Type the rest of the buffer.

```
*t r<RET>
  2                  Did gyre and gimble in the wabe;
  3                  All mimsy were the borogoves,
  4                  And the mome raths outgrabe.
[EOB]
```

4. Type every line in the buffer that contains the word "and."

```
*t all 'and'<RET>
  1                  'Twas brillig, and the slithy toves
  2                  Did gyre and gimble in the wabe;
  4                  And the mome raths outprabe.
```

Deleting and Replacing Text

Range specifications are useful not only for displaying lines but also for manipulating text. The following examples show how to delete and replace text in the buffer.

The /QUERY option (/Q) can be used to decide whether or not to change individual lines. EDT responds to the /QUERY option with a ? prompt. A carriage return after this prompt causes EDT to print help information.

1. Suppose the user wants to delete either line 2 or 3. The /QUERY option can be used to read them first.

```
*D2:3/Q<RET>
  2                  Did gyre and gimble in the wabe;
?<RET>
Please answer Y(es), N(o), Q(uit), or A(ll)
?N
  3                  All mimsy were the borogoves,
?Y<RET>
1 line deleted
  4                  And the mome raths outgrabe.
*
```

Notice that EDT displays the next line in the text. The file now looks like this:

```
1.  *t w<RET>
      1             'Twas brillig, and the slithy toves
      2              Did gyre and gimble in the wabe;
      4              And the mome raths outgrabe.
    [EOB]
    *
```

The line numbers can be resequenced in the buffer with the following command:

```
1.  *res 1:4<RET>
    3 lines resequenced
```

EDT checks lines 1 through 4 and renumbers them in increments of 1. In this case, simply typing res<RET> would have done as well, since the resequence command defaults to the whole buffer.

2. Replace the new line 3 with two more lines.

```
    *re 3<RET>
    1 line deleted
                    While grimply at her terminal
                    The snofu mumped agrabe.
    ↑Z
    [EOB]
    *
```

Notice that the REPLACE command deletes the line you specify and puts EDT in the insert level of line mode. Exiting with CTRL/Z confirms that the text is to be inserted. The file now looks like this:

```
1.  *t w
      1             'Twas brillig, and the slithy toves
      2              Did gyre and gimble in the wabe;
      3             While grimply at her terminal
      4              The snofu mumped agrabe.
    [EOB]
    *
```

Entering Change Mode

For a VT52 or VT100 terminal, the easiest way to edit a file is with keypad functions. The default mode can be reset for video terminals with the SET KEYPAD command. When the user types an abbreviation for change mode (C, CH, or CHA) he or she automatically enters the keypad submode of change mode:

```
  *CH<RET>
```

The screen clears, and then the contents of the buffer appear in the upper left of the screen. The cursor appears as an underscore under the first character in the file. The cursor appears on the first character in the buffer. Everything typed at this point is inserted directly into the buffer.

Using the Keypad

To move the cursor to the bottom of the buffer, the GOLD key is pressed and then the BOTTOM key. The buffer appears as shown:

```
                                    'Twas brillig, and the slithy toves
                                    Did gyre and gimble in the wabe;
                                    While grimply at her terminal
                                    The snofu mumped agrabe.
This is Line 1.
[EOB]
```

Any characters that typed on the main keyboard are inserted before the cursor:

```
                                    But none was more beguiling<RET>
                                    Than keypad EDT.<RET>
                                    [EOB]
```

If the up-arrow key is pressed twice, the cursor will be moved up two lines. The screen would then look like this:

```
                                    'Twas brillig, and the slithy toves
                                    Did gyre and gimble in the wabe;
                                    While grimply at her terminal
                                    The snofu mumped agrabe.
                                    But none was more beguiling
                                    Than keypad EDT.
                                    [EOB]
```

A section of text can be moved about in the buffer with the CUT and PASTE commands. The following shows how to move the last two lines to the beginning of the buffer:

1. Mark the start of the lines by pressing the SELECT key
2. Move the cursor to the end of the lines (just above [EOB]) by pressing the down arrow key twice
3. Press the CUT key to insert the two lines into the paste buffer. The lines will disappear from the screen
4. Move the cursor to the top of the file by pressing the GOLD key and then the TOP key

5. Press the GOLD key and then the PASTE key to place the contents of the paste buffer at the start of the buffer being edited

The file now looks like this:

```
                    But none was more beguiling
                    Than keypad EDT.
                    'Twas brillig, and the slithy toves
                    Did gyre and gimble in the wabe;
                    While grimply at her terminal
                    The snofu mumped agrabe.

                    [EOB]
```

The GOLD and PASTE keys can be pressed for as many times as these two lines are to be duplicated.

Returning to Command Level
To write out the buffer and exit change mode, the user enters a CTRL/Z. This returns the user to the asterisk prompt in line mode. Next, the user types "EX" (for EXIT) after the prompt:

```
    *EX<RET>
```

Typing EXIT displays a message on the status of the file just edited:

```
    DBA1:[USER]TEST.;1 7 lines
$
```

If the file is a practice file, the user types "QUIT" instead:

```
*QUIT<RET>
```

QUIT returns the user to the operating system's command level prompt without writing out the MAIN buffer.

THE SOS EDITOR

The SOS editor is a line-oriented, interactive text-editing program. It has features that allow examination and modification of text, character by character. The SOS editor can be used to perform the following functions:

- Examine, create, and modify ASCII files
- Search for and/or change one or more arbitrary text strings, with the option to verify each change before it is made
- Merge parts of one file into another
- Create a file that is a subset of another file

Because the SOS editor is line-oriented, it operates with line-numbered text files. If a file is edited that does not contain line numbers, the editor adds line numbers to the text lines. The SOS editor requires the maintenance of line numbers within the file. For most SOS commands, a line number or range of line numbers specifies the text to be operated on. When commanded to insert, delete, move, or copy text, The SOS editor maintains line numbers in ascending order within each page of text.

In certain modes of operation, the SOS editor responds on a character-by-character basis. For example, one SOS feature that exhibits this character-by-character interactivity is the Alter mode. This special mode permits interactive changes within a line of text. Alter mode has its own commands and syntax; it functions essentially as an editor within an editor.

Advanced features of SOS allow considerable flexibility in searching for a string of text and allow specification of blocks of text by content, instead of by line number. SOS features many user-controlled default values.

Initiating and Terminating SOS

The SOS editor is initiated by entering one of the following commands in response to the command language prompt:

```
$ EDIT/SOS file-spec <RET>
```

To terminate SOS, enter the command E (EXIT) after the SOS editor's prompt
(*):

```
*E<RET>
[file-spec]

$
```

Upon terminating, the editor writes an output file containing all the modifications made in editing the file. The original file is not changed. The specifier that the SOS editor uses for the output file has a version number higher by 1 than the latest version of the original file.

SOS Modes of Operation

The SOS editor is capable of operating in various modes. A mode of operation is a state in which the editor interprets terminal input in a distinctive way. Edit mode is the foundation of SOS, from which the other modes can be accessed. The SOS editor can be initiated as follows:

- Input mode—allows the insertion of one or more new lines of text into a file. Input mode is entered either directly via the command language or via the Edit mode
- Edit mode—allows extensive modification, additions to, and deletions from an existing file

The SOS editor includes many advanced features. These features allow the user to search through files without editing, copy parts of files, alter individual lines interactively, and decide on text substitutions interactively.

Input Mode
The SOS Input mode is used for creating new files or adding lines to a file. Input may be entered either directly via the command language or via the Edit mode. SOS Input mode is invoked if the file being referenced does not exist. Therefore, SOS creates a file with the specified name and waits to process input entry as illustrated below:

```
$ EDIT/SOS NEW.FOR <RET>
SOS VERSION V02.02D2

INPUT:SYS0:[TERRY]NEW.FOR.1
00100
```

The SOS editor prints the word INPUT before the file-spec, indicating that it is creating a new file and operating within the Input mode. While in the Input mode, SOS prompts the user by printing the line number of the line to be entered. The user must terminate each new line of text with a carriage return character, <RET>. To correct typing errors while entering text, use the terminal control characters described in Chapter 3.

After completing the input process, switch to Edit mode by entering an escape character, <ESC>. The escape character on other terminals may appear as either ALTmode or SELect. The escape character may be entered either at the end of a line of input or after SOS prompts with the next line number. The SOS editor follows the user-entered escape character by printing an asterisk (*), indicating Edit mode.

While in Edit mode, modifications may be made to the new file by using other Edit mode commands or Alter mode commands. Upon completion of all modifications, SOS can be terminated by entering the E command. If it is necessary to enter lines of text into an existing file, use either the Input or Replace commands in the Edit mode.

Edit Mode
The Edit mode constitutes the major part of the SOS editor. With the exception of the Read-only mode, the user is able to switch to any of

the other modes of operation from the Edit mode and return. SOS accepts 24 commands in Edit mode, many of which can be represented by a single character. Table 4-1 describes each of the Edit mode commands.

To initiate SOS in Edit mode, enter the file-spec of an existing file either on the Edit command line or in response to the SOS prompt as illustrated below:

```
$ EDIT/SOS <RET>
SOS VERSION V02.02D2
File:PROG2.COB<RET>

EDIT:SY0:[EMILY]PROG2.COB.4
```

Table 4-1 Common Edit Mode Commands

Form	Command	Description
C	Copy	Copy a range of lines to another place within a file, or from another file
D	Delete	Delete a range of lines
E	End	Terminate SOS, return to command language monitor
F	Find	Search for the occurrence of one or more specified strings of text
H	Help	Print HELP facility on terminal
I	Input	Enter Input mode to insert lines of text
N	reNumber	Renumber a range of lines
P	Print	Print a range of lines on the terminal
R	Replace	Delete a range of lines and enter Input mode

Table 4-1 Common Edit Mode Commands

Form	Command	Description
S	Substitute	Replace one or more text strings with other string(s) in a range of lines
T	Transfer	Copy a range of lines to a new location and delete the original lines
W	Save World	Write a new file containing all the changes made so far and continue editing
<RET>		Print next line
<ESC>		Print previous line

SOS Examples

Copy command

1) C300,9000:95000
   Make a copy of lines numbered 9000-9500 and insert the lines after line 300.

Delete command

1) D1700:1750
   Delete lines numbered 1700 through 1750.
2) D400
   Delete line numbered 400.

Find command

1) Fmore<ESC>
   Search for "more" from the current point in the file.
2) Fmore<ESC>,1:1000
   Search for the first occurrence of "more" in the range of lines from 1 through 1000.

Input command

1) I1200,5

   Insert lines following line 1200 with new lines being numbered with increment 5.

Print command

1) P500:800

   Print lines 500 through 800.

2) P1800

   Print line numbered 1800.

Replace command

1) R1700:1750,5

   Delete lines 1700 through 1750 and insert starting at 1700 with line increment of 5.

Substitute command

1) Smore<ESC>less<ESC>,500:800

   Change all occurrences of "more" into "less" on lines numbered 500 through 800.

Transfer command

1) T300,9000:9500

   Move all lines numbered 9000 through 9500 to a point following line 300, deleting the lines in the old location.

THE SLP EDITOR

The SLP editor is the batch-oriented editing program used for source file maintenance. It allows updating (deletion, replacement, addition) of lines in an existing file. Furthermore, the SLP editor generates a record of editing modifications. The SLP command file provides a reliable method of duplicating the changes made to a file, at a later time or on another computer system.

Input to the SLP editor consists of a correction input file that is to be updated, and command input containing text lines and edit command lines that specify the update operations to be performed. SLP locates lines to be changed by means of locators (sequence numbers or character strings). Command input normally enters through an indirect file that contains commands and text input lines to be inserted into the file. Alternatively, commands can be entered from the terminal.

SLP output is a listing file and an updated copy of the corrected input file. SLP provides an audit trail that helps keep track of the update status of each line in the file. The audit trail is provided in the listing and is included permanently in the output file. When a given file is updated with successive versions of an SLP command file, different audit trails may be used to differentiate between changes made at various times.

SLP output qualifiers permit the user to create or suppress an audit trail, eliminate an existing audit trail, specify the length and beginning position of the audit trail, or generate a double-spaced listing.

Initiating and Terminating SLP

SLP is initiated via the command language EDIT command. The normal way to use the SLP editor is to specify an indirect command file that informs SLP what files to process, and indicates what editing changes are to be made to the correction input file. The indirect file can be specified on the same line with the EDIT command, or on a separate line. The indirect file must be created before running SLP. The interactive text editor SOS is normally used to create SLP indirect command files. If both new and old versions of the file exist, the differences utility (see Chapter 3) can be used to create a SLP correction file that will change the old file into the new one.

SLP Input and Output Files

The SLP editor requires two types of input: a correction input file and command input. The correction input file is the source file to be updated using SLP. Command input consists of an initialization line, followed by SLP edit commands that indicate how the file is to be changed.

SLP output consists of a listing file and an output file. The listing file is a copy of the output file with sequence numbers added; it shows the changes SLP makes to the correction input file. The output file is the permanently updated copy of the input file that resides on the system.

The Correction Input File

The correction input file is the file to be updated by SLP. It can contain any number of lines of text. When SLP processes the correction input file, it makes the changes specified by SLP edit commands with an audit trail in the output file.

Command Input

The SLP editor uses command input to update files. Normally, SLP reads command input from an indirect file; alternatively, the user can enter commands from the terminal. Command input consists of:

- An initialization line that informs SLP what files to process
- SLP edit command lines that define changes to the input file
- New lines of text to insert into the output file
- A command terminator—a single slash in Column 1

The SLP Listing File

The SLP listing file shows the updates made to the source file. Each line in the listing file is numbered in sequence. Updates are marked by means of an audit trail (unless the qualifier that suppresses audit trail generation is specified).

The SLP Output File

The SLP output file is the updated input file. All of the updates specified by the command input are inserted in this file. A default audit trail, unless suppressed, is applied to lines changed by the update. The numbers generated by SLP for the listing file do not appear in the output file.

Specifying SLP Edit Commands

The SLP edit commands permit updating source files by adding, deleting, and replacing lines in a file. SLP edit commands are marked by certain characters that SLP interprets as operators.

SLP Operators

The SLP editor interprets each of the following characters, when entered as the first character of an input file, as special operators: the minus sign (−), the backslash (\), the percent sign (%), the at sign (@), the slash (/), and the less-than character (<). Table 4-2 lists each of these operators and the functions they perform.

The less-than character (<) is the escape character that allows characters that SLP otherwise would interpret as operators to be entered in the command input (in column 1). For example, </ hides the slash character from SLP, thereby enabling slash entry into the output file without terminating the SLP edit session. The less-than character can be used as an escape character for all SLP operators listed in Table 4-2 (including itself).

Table 4-2 SLP Operators

Operator	Function
–	First character of an SLP edit command
\	Suppress audit-trail processing
%	Re-enable audit-trail processing
@	Invoke an indirect file for SLP processing
/	Terminate the edit session and return to SLP command level
<	Escape character

General Form of the Edit Command

The general form of the SLP command is as follows:

```
    –locator1[,locator2][,/audittrail/][;comment]
    inputline
    .
    .
    .
```

where:

–(minus)	Specifies that this is an SLP edit command line
locator1	A line locator that causes SLP to move the current line pointer to a specified line. If only locator1 is specified, the current line pointer is moved to that line and SLP reads the next line in the edit command file
locator2	A line locator that defines a range of lines (that is, the range beginning with locator1 and ending with locator2) to be deleted or replaced
/audittrail/	A character string used to keep track of the update status of each line in the file. This audit trail is used to mark new or replaced lines in the file until the audit trail is either changed or suppressed. This argument must be delimited by a slash (/)
inputline	A line of new text to be inserted into the file immediately following the current line. Any number of input lines can be entered

;comment	An optional comment. SLP ignores any text after a semicolon

All fields in the command line are position-dependent; commas must be specified.

The locator fields can take one of the following forms:

```
(/string[...string]/)
(   number   )   [+n]
(      .     )
```

String, number, n, and period (.)are defined as follows:

string	A string of ASCII characters. SLP locates the next line in which the string exists and moves the current line pointer to that line. If the locator is specified in the form /string...string/ (that is, two different strings of characters separated by three periods), SLP locates the line in which the two character strings delimit a larger character string
number	Specifies a sequence number to which the current line pointer is to be moved. The largest sequence number that can be specified is 9999
n	Specifies a decimal value used as an offset from the line specified by the locator
. (period)	Indicates the current line

All forms of the line locator can be specified interchangeably in a command line.

SLP can edit files sequentially only. Once the current line pointer moves past a given line in the file, it can not be returned. The file must be closed by typing CTRL/Z, and another SLP edit session invoked.

SLP Examples

1) −350
   Performs insert following the 350th line
2) −17,23
   Deletes the 17th through 23rd lines

LINKER

Before a source-language program can be run on VAX/VMS, it must be assembled or compiled by a language processor and then linked. The Language processors translate user-written source programs into object modules. The VAX-11 Linker binds these object modules into an image that can be executed by the VAX system.

Not all computer systems employ a linker; in some, the work of the linker is assumed by the language processors and what is called a "loader". But the linker offers programmers on VAX/VMS greater flexibility in choosing and mixing languages, and simplifies and extends the modern approach of modular programming.

Input to the Linker

There are two basic forms of input processed by the linker: object modules and shareable images. They are introduced to the linker as part of the input files specified in the LINK command. The linker will accept one or more of the following kinds of input files:

- Object file
- Shareable image file
- Symbol table file
- Library file
- Options file

The object file can contain one or more object modules. This file has file-type OBJ. It is the fundamental input to the linker and at least one object file must be specified with any LINK command.

The shareable image file is the product of a previous linking operation, but one which is by itself not executable. It can serve only as input to another linking operation. The shareable image file can only be specified in the options file and is indicated there by the /SHAREABLE qualifier.

The symbol table file is also a product of a previous linking operation. It may be specified when linking so that the linker can use the symbol values to resolve undefined symbols in other object modules. A symbol table file has the file type STB.

There are two kinds of library files: object and shareable image. Of these there are both system libraries (maintained by VMS) and user Libraries (created by the DCL command, LIBRARY). Library files are used by the linker either to resolve undefined symbols, or as a source for particular object modules or shareable images specified with the /INCLUDE qualifer in the LINK command.

The options file is not really input to the linker, in the same sense the other files mentioned are input; rather, it is a tool for managing the linking operation and for simplifying the use of complex and often-repeated linker operations. (This is, in a way, analogous to the use of DCL command procedures for complex or commonly used command sequences.) A linker options file can contain one or more input file specifications, including qualifiers or special linker options that cannot be specified in the DCL LINK command line.

Output of the Linker

The linker will generate one of three types of images:

- Executable
- Shareable
- System

and an optional image map and/or symbol table.

The most common output of the linker is the executable image. It is the end product of program development. It has the file type EXE and can be run by the DCL command, RUN.

A shareable image, on the other hand, cannot be executed directly. It must be linked with one or more object modules to produce an executable image. It contains an image header, one or more image sections, and a symbol table that defines universal symbols in the shareable image.

A system image is one that does not run under the control of the operating system, but is intended to run stand-alone on a VAX. VAX/VMS is a system image.

If the /MAP qualifier is specified in the LINK command, the linker will generate one or more of the following (at the user's option) in an image map file:

- An object module synopsis
- A module relocatable reference synopsis
- An image section synopsis
- A program section synopsis
- A list of symbols by name
- A list of symbols by value
- Link run statistics

If the /SYMBOL_TABLE qualifier is specified, the linker will generate a symbol table file that can serve as input to a subsequent linking operation.

Action of the Linker

In the process of creating an image the linker performs three major tasks:

- resolution of symbolic references
- allocation of virtual memory
- image initialization

The following sections describe these processes in some detail.

Resolution of Symbolic References
A symbol is a name associated with a program location or a value. Any reference to a symbol, other than the definitive reference, must be resolved. For example:

```
        JMP SYMBOL_1                 (Jump to where?)
or
        ADD SYMBOL_A,SYMBOL_B        (Add what to what?)
```

Somewhere, SYMBOL_1 must be defined as a location of an instruction or the beginning of a subroutine. Similarly, SYMBOL_A and SYMBOL_B must have had values assigned to them.

References to local symbols (that is, symbols defined and used entirely within the module) are resolved by the language processor, but references to global symbols (those that can be referred to by modules other than the defining module) and universal symbols (those referenced outside of a shareable image) must be resolved by the linker.

Since universal symbols are in fact global symbols that are available to modules outside of a shareable image, the process whereby the linker resolves global and universal symbols is the same. During its first pass through the linking operation, the linker records each symbol reference and definition in a global symbol table. When the linker seeks to resolve a symbol reference, it first searches modules named in the command line (with /INCLUDE), then user libraries, and finally system default libraries.

Memory Allocation
By the end of its first pass, the linker has processed all the input modules and library modules needed to resolve undefined symbols, and knows how large the final image will be, but it still needs to organize the image and allocate virtual memory.

The linker organizes the image on three levels: cluster, image section, and program section.

Clusters are determined in three ways:

- The default cluster (generated by the linker)
- User defined clusters (generated by the CLUSTERS= option)
- Shareable image clusters (one for each shareable image)

Image sections are created by gathering program sections (psects) with similar attributes. Those attributes include writeability, executeability, shareability, position-independence, and protected vector.

Program sections and their attributes are determined by the language and, optionally, by the user, either through directives to the language processor (for example, .PSECT in MACRO) or by the PSEC_ATTR option in the linker options file.

The linker processes each cluster, one at a time—with the exception of non-based, or position-independent, shareable images, which are allocated virtual memory by the image activator at runtime. In processing all other clusters, the linker organizes the psects within each cluster into image sections. Then the clusters are assigned virtual address space and the image section descriptor (ISD) of each image section is updated to include the starting virtual address of the image section.

Image Initialization

After resolving references and allocating memory, the linker fills in the actual contents of the image. Primarily, initialization consists of copying all data and code into a single image; but the linker performs two other functions at this stage:

- Computes values that depend on externally defined fields
- Inserts these values into the referencing location

Fix-up Image Section

After it has initialized the image, the linker will generate a special image section, called the fix-up image section. This image section contains the code that makes otherwise position-dependent shareable images position independent.

The general addressing mode is used to reference routines and data contained in a shareable image. The linker converts general addressing mode directives into longword deferred addressing mode, with indirecion going through the fix-up image section. Failure to use general addressing mode when referencing a shareable image will elicit a warning message.

All DIGITAL VAX-11 high-level languages generate position-independent code.

Shareable Images

An important benefit of the linker—perhaps the most important benefit—is that it allows the use of shareable images. An effective application of shareable images can help to conserve valuable resources in the users operating environment. For example, physical memory requirements would be reduced if global sections (one for each image section of a shareable image) used commonly among processes could be resident in memory and mapped into their address space, thus the same physical pages satisfy a number of processes, reducing duplication. So, too, can the user conserve disk storage and reduce paging I/O, when sharing replaces duplication.

One of the reasons modular programming is so attractive, is that a commonly used routine or function can be developed or modified once, then incorporated into any number of programs. The use of shareable images carries this efficient practice a step further. The modules that make up a shareable image are linked only once, so the overhead of resolving undefined symbols (within the image) and generating image sections—the bulk of the linkers work—is incurred only once, facilitating another level of modular hierachy. Furthermore, since a position-independent shareable image is allocated virtual memory by the image activator at runtime, the code it comprises can be modified and updated without having to re-link every program that uses that image.

The LINK Command
The linker is run by the DCL command:

```
        $ LINK [/command-qualifier...] file-spec [/file-qualifier...]...
```

At least one input file must be specified. There can be multiple command qualifiers, multiple file specifications, and multiple qualifiers for each file specified.

VAX DEBUG

The VAX DEBUG program is a language-independent, interactive, symbolic debugger that works with programs written in most of the languages supported by the VAX/VMS operating system. Current languages for which the debugger works are: VAX-11 FORTRAN, VAX-11 BASIC, VAX-11 COBOL, VAX-11 PASCAL, VAX-11 PL/I, VAX-11 BLISS-32, and the VAX-11 MACRO assembly language.

DEBUG enables dynamic examination and modification of the contents of memory locations, which is useful for finding errors in programs. Breakpoints may be set to stop program execution at specific points, and critical code sections can be single-stepped— line by line or instruction by instruction—to verify correct execution. Since user program execution is controlled by DEBUG once it is invoked, modifications may be made to the program while it is executing.

Linking DEBUG with the User Program
Before DEBUG can process the user program, it must be linked to it. This can be accomplished by specifying the /DEBUG qualifier in the LINK command as follows:

```
        $LINK/DEBUG PROG1
```

This links the debugger to the user process called PROG1. Subsequent execution of the program is controlled by the debugger.

DEBUG Execution

Once linked, the process begins execution under the control of DEBUG after the RUN command has been entered:

```
$ RUN PROG1
```

As a response to this command, DEBUG will issue an identification message that verifies its control of the program; that is followed by a prompt for additional DEBUG commands:

```
$RUN PROG1
                VAX-11 DEBUG Version 3.0-3
DBG>
```

(Note: DBG> is the DEBUG prompt symbol.)

The programmer may now enter a series of DEBUG commands to manipulate the execution according to program needs.

DEBUG Commands

DEBUG commands direct the execution of the program and can be used to aid the programmer in the debugging process. The DEBUG commands can:

1. Specify points at which execution will be suspended, when and if they are encountered, by using the SET BREAK command
2. Trace the sequence of program execution by means of the SET TRACE command. This command establishes tracepoints in the program
3. Display before-and-after values of a location whenever that location is stored into, by means of the SET WATCH command
4. Initiate or resume execution, by means of the GO command
5. Execute a single line or instruction of the program by means of the STEP command
6. Determine the location of breakpoints, tracepoints, and watchpoints by means of the commands SHOW BREAK, SHOW TRACE, SHOW WATCH, respectively
7. Erase breakpoints, tracepoints, and watchpoints in the program, through use of the CANCEL command
8. Display the contents of variables or memory locations, by using the EXAMINE command
9. Change the contents of variables or memory locations, by using the DEPOSIT command

10. Obtain the value of an expression or the current address of a symbol by using the EVALUATE command
11. Call a subroutine at DEBUG time, by means of the CALL command
12. Change values of parameters for LANGUAGE, SCOPE, MODE, and TYPE
13. Specify an arbitrary file name for the DEBUG log file by means of the SET LOG command
14. Control DEBUG I/O at debug time, via the SET OUTPUT command. This includes normal terminal output, log file output, and command file verification
15. Find all current output attributes (VERIFY, TERMINAL and LOG) by using the SHOW OUTPUT command. For more limited needs, a SHOW LOG command is available that displays only the LOG data
16. Instruct DEBUG to take commands from a specified file by means of @Filespec
17. Display source lines with compiler-assigned listing line numbers (for some languages only)

SET Command

The SET command is used in a variety of ways to establish one or more conditions pertinent to DEBUG. It has the form:

```
SET keyword parameter
```

Table 4-3 summarizes the values that may be used for keyword and parameter.

Table 4-3 SET Command Summary

Keyword	**Parameter**	**Function**
LANGUAGE	Language-name	Specifies the language characteristics to be used by DEBUG DBG> SET LANGUAGE FORTRAN
BREAK	address [DO(DEBUG commands)]	Sets a breakpoint at a location in the program; optionally specifies commands to be performed when program execution reaches that point DBG> SET BREAK SUB2 DO(EXAMINE K)
TRACE	address	Lets the user follow the program's execution sequence, to ensure that instructions are being executed in proper order DBG> SET TRACE %LINE 25 (see note below) DBG> SET TRACE %LABEL 99 (see note below)
WATCH	address	Sets a watchpoint at the specified address DBG> SET WATCH SYM

MODULE	module-name	Makes all local symbols in the specified module available to DEBUG DBG> SET MODULE FOO
SCOPE	A list whose elements may be: pathname nonnegative integer "\\"	Establishes an ordered list of scopes which DEBUG searches when looking up the definitions of symbols
MODE	Radix: DECIMAL HEXADECIMAL OCTAL Display: [NO]SYMBOLIC	Alters the defaults used by DEBUG for radix and symbolic representation of addresses
TYPE	BYTE WORD LONG QUAD, OCTAWORD D-FLOAT F-FLOAT G, H-FLOAT ASCII:length INSTRUCTION	Establishes a data type to be used to interpret those addresses for which DEBUG cannot infer a type from the data definition

NOTE

The %LINE and %LABEL modifiers are used to indicate line numbers (%LINE) and numeric statement labels (%LABEL).

LOG	file name	Specifies that the DEBUG log can be called something other than the default name, "DEBUG.LOG" DBG> SET LOG NEW.LOG

OUTPUT	[NO]LOG, [NO]TERMINAL, [NO]VERIFY	Tailors output modes of DEBUG to suit particular applications DBG> SET OUTPUT NOLOG VERIFY TERMINAL
SOURCE	directry...	Specifies which directories are to be searched for source files DBG> SET SOURCE [MYDIR],[MAST.SCR]

EVALUATE Command
The EVALUATE command allows the user to check the value of an expression or the definition of a symbol. It has the form:

```
EVALUATE expression
```

where the evaluation follows the rules of the host language.

To illustrate, if the element to be evaluated is a FORTRAN expression (for example, (2*K-1)+A*B), the precedence of operations follows the FORTRAN standard: parenthetical operations, followed by exponentiation, followed by multiplication and division, followed by addition and subtraction, from left to right.

The value is displayed according to the source language rules for data types. That is, if a FORTRAN expression contains both real and integer elements, the value will be expressed as a real value.

CALL Command
CALL is used to execute a subroutine while under the control of DEBUG. The subroutine may be one that was included specifically for debugging use, or one that was used by the application program during normal execution. The CALL command has the form:

```
CALL s(a,...,a)
```

where

s	subroutine name
a,...,a	actual arguments

SHOW Command
The SHOW command allows the user to check the status of DEBUG settings, such as the location of breakpoints. The SHOW command has the form:

```
SHOW keyword
```

Table 4-4 SHOW Command Summary

Keyword	**Function**
BREAK	Displays, in symbolic form, the location of each breakpoint in the program
TRACE	Displays, in symbolic form, the location of each tracepoint in the program
WATCH	Displays, in symbolic form, the location of each watchpoint in the program
MODE	Displays the current modes
SCOPE	Displays the current ordered list of scopes
TYPE	Displays current default type
OUTPUT	Displays output attributes
SOURCE	Displays current directory search list

CANCEL Command
The CANCEL command is used to nullify conditions established by earlier SET commands, such as eliminating breakpoints. The CANCEL command has the form:

```
CANCEL keyword [parameter]
```

Table 4-5 lists the keywords and parameters that can be specified with CANCEL.

Table 4-5 CANCEL Command Summary

Keyword	**Parameter**	**Function**
BREAK	address	Eliminates the breakpoint at the specified location
TRACE	address	Eliminates the tracepoint at the specified location
WATCH	address	C
ied location		
MODULE	module-name	Removes all local symbols in the specified module from DEBUG symbol table and releases their symbol table space
SCOPE	none	Restores the default value of SCOPE
MODE	none	Cancels the current mode for radix, length, and data type and restores the default values
ALL	none	Cancels all parameters previously set for DEBUG
SOURCE	none	Cancels SET SOURCE command

GO command
Use the GO command to start or resume program execution. The GO command has the form:

```
GO [address]
```

If the user types the GO command, but does not include an address as specified, its effect is either to start program execution at the beginning, or to resume execution from the point where it stopped (such as a breakpoint). If an address is specified, DEBUG restarts program execution at that address.

Example:

```
GO %LINE 12
```

DEBUG resumes program execution at line 12 of the program.

NOTE

Attempting to restart a program from the beginning will yield unpredictable and unreliable results.

STEP Command

The STEP command allows the user to specify that a specific number of instructions or statements are to be executed in the user program. Thereafter, execution will again stop. The user may specify instruction or statement stepping (assuming the language supports statement numbers). The STEP command has the form:

```
STEP [n]
```

where the value of n is a decimal integer indicating how many instructions or statements to execute. If n is omitted, one instruction or statement is executed. This command allows the user to suspend program execution prior to reaching a breakpoint or a tracepoint, so the user can examine the result of program execution on an instruction-by-instruction or statement-by-statement basis.

If the program has not begun to execute, the STEP command causes n instructions or statements to be executed, starting with the first executable instruction or statement in the program. If program execution has started and been suspended, the STEP command causes n instructions or statements to be executed starting from the point of suspension.

EXAMINE Command

To determine the current contents of locations in the user program, use the EXAMINE command. The EXAMINE command has the form:

```
EXAMINE [address]
```

To specify an address, enter the symbolic variable name defined in the source program. DEBUG displays the contents in the format:

```
address:     contents
```

Both the address and its contents are displayed in a form appropriate to the host language. That is, the user will not have to translate from hexadecimal to ASCII in order to determine the value of a location that contains character data.

The address will, if possible, be displayed symbolically when the mode is SYMBOLIC. Otherwise, it will be displayed numerically.

If an address is not specified, the next location's contents are displayed. To display a range of locations, specify the EXAMINE command as follows:

```
EXAMINE     address1:address2
```

The current contents of the locations from address1 to address2 will be displayed.

DEPOSIT Command

To change the contents of a location while debugging, use the DEPOSIT command, which has the form:

```
DEPOSIT  address = data
```

For example:

```
DEPOSIT  LOC = 100
```

places the value 100 into the location symbolized by LOC.

TYPE Command

The TYPE command is used to display lines of source code. The format of the TYPE command is:

```
TYPE  moduleline-number1:line-number2
```

To display source lines in the current scope, this may be abbreviated to:

```
TYPE  line-number1:line-number2
```

or just

```
TYPE  line-number
```

EXIT Command

To terminate the DEBUG session and return to the DIGITAL Command Language level, use the EXIT command.

THE VAX RUN-TIME LIBRARY

The VAX Run-Time Library (RTL) is composed of a set of language-independent and language-specific VAX procedures which establish a common runtime environment for user programs written in any native mode language. Because all of the language support procedures follow the same programming standards, a user program can be composed of modules written in different languages, including assembly language. Because of the VAX procedure calling standard, each native mode user module can call any other native mode user module or any of the procedures in the Run-Time Library.

Most of the VAX Run-Time Library is constructed as a separate shareable image which interacts with the rest of the operating system via an entry point vector. This allows:

1. Installation of a new library without the need of relinking all user programs
2. Implementation of new internal algorithms without relinking all user programs
3. A single copy of the library to be shared by all processes

NOTE

A portion of the Run-Time Library is not shareable.

Each procedure entry point in the shareable image has a storage location in the area known as the entry vector. Each entry vector contains the starting address of an associated procedure to be executed when a user program calls the library. Use of entry vectors permits a single position-independent copy of the library to be bound to different virtual addresses in processes which are sharing it. Use of entry vectors and address binding at image activation also permits a new release of the library to be installed without requiring that user images be relinked.

The VAX Run-Time Library comprises several sections which are grouped by function or calling sequence. They include:

- A resource allocation section
- A condition handling section
- A general utility section
- A mathematical section
- A language-independent support section
- Language-specific support sections
- A string handling section

The Run-Time Library is designed as a set of modular re-entrant procedures that run in user mode.

Resource Allocation Section (LIB$)

The Resource Allocation Section includes all procedures that allow allocation of process-wide resources. Such resources include the following:

1. Virtual Memory—one procedure to allocate and one to deallocate arbitrarily sized blocks of process virtual memory
2. Logical Unit Numbers—allow logical unit numbers to be allocated in a modular manner
3. Event flags—same as logical unit numbers

In most cases, the resource allocation procedures must be used to allocate process-wide resources in order for all library, DIGITAL, and customer-written procedures to work together properly within an image.

Signaling And Condition Handling
The VAX condition handling facility is a collection of library procedures and system services that provides a unified and standardized mechanism for handling errors internally in the operating system, the Run-Time Library, and user programs. In some cases, the mechanism is also used to communicate errors across these interfaces. In particular, all error messages are printed using this mechanism. Where an error condition is signaled, the process stack is scanned in reverse order. Establishing a handler provides the programmer with some control over fix-up, reporting, and flow of control on errors. It provides the system and library messages in order to give a more suitable application-oriented user interface.

Error Processing Procedures
Errors detected by the Run-Time Library are indicated by returning an error completion status wherever possible. This is especially true for the general utility library (LIB$). However, the math library and the language support libraries indicate most errors by calling the VAX LIB$SIGNAL or LIB$STOP procedures. The LIB$SIGNAL procedures use a condition value as an argument which has an associated error message stored in a system error message file. The condition is signaled to successive procedure activations in the process stack. These procedures may have established handlers to handle the conditions or change the error message. Thus an application can tailor its error messages to its own needs.

The Run-Time Library provides routines to perform the following condition handling functions:

- Establish and delete user condition handlers (LIB$ESTABLISH, LIB$REVERT)
- Enable and disable the detection of the hardware and conditions decimal overflow, floating-point underflow, and integer overflow
- Signal exception conditions and stop execution by means of the signaling mechanism (LIB$SIGNAL, LIB$STOP)
- Emulate VAX instructions that are not implemented on the host processor (LIB$EMULATE)
- Convert floating faults to floating traps (LIB$SIM-TRAP)
- Find the reserved operand of any F-, D-, G-, or H- floating instruction after a reserved operand fault has been signaled (LIB$FIXUP-FLT)

General Utility (LIB$)

General utility procedures are not mandatory in order to use the rest of the library successfully. They provide a wide range of functions for the convenience of the user:

- Common I/O procedures—These procedures perform such functions as getting records from the current input device (LIB$GET-INPUT) and putting them to the output device (LIB$PUT-OUTPUT), executing DCL commands from a running program (LIB$DO-COMMAND), getting the command line from a 'foreign command' (LIB$GET-FOREIGN), and copying strings to and from the process's common storage are (LIB$PUT-COMMON, LIB$GET-COMMON). I/O control procedures are also available to customize printer output and translate logical names
- Terminal independent screen procedures—These procedures provide a high-level language interface to the video terminal. They put text to the screen, mover the cursosr to the desired position, erase text from the screen, and manipulate the screen buffer
- Data type conversion procedures—These procedures perform conversions between one VAX data type and another (e.g., text to D-floating, decimal to binary)
- Variable bit field instruction procedures—LIB$INSV, LIB$EXTV, and LIB$EXTZV insert and extend variable bit fields. LIB$FFC searches a bit field for the first set bit
- Performance measurement procedures—These procedures provide a facility for timing, counting I/O operations, and counting page faults
- Date/time utility procedures—These procedures return the system data or time in several forms
- CRC procedures—LIB$CRC and LIB$CRC-TABLE permit the user to calculate the cyclic redundancy check for a data stream
- Multiple precision arithmetic procedures—LIB$ADDX and LIB$SUBX add and subtract signed two's-complement integers of arbitrary length

Run-Time Library procedures also permit the high-level language programs to use the following VAX hardware instructions:

- Extended multiply and integerize—EMODF, EMODD, EMODG,EMODH
- Evaluate polynomials—POLYF, POLYD, POLYG, POLYH
- Insert and remove queue entry—INSQHI, INSQTI, REMQHI, REMQTI

Mathematical Functions (MTH$)

The mathematical library consists of over 200 standard procedures to perform common mathematical functions. These functions include:

- Floating-point mathematical functions: trigonometric, logarithmic, and square root
- Complex functions: absolute values, conjugation, trigonometric, arithmetic, exponetiation, return imaginary part of complex number, return real part of complex number, make complex from floating-point, logarithmic, and square root
- Exponentiation on floating-point, word, longword, and complex data
- Random number generators
- Processor-defined mathematical procedures including both the intrinsic and basic external functions defined in ANSI FORTRAN, whcih are treated in a uniform manner. They include routines that perform conversions between floating-point and integer data and a large number of miscellaneous procedures

Language-Independent Support (OTS$)

The language support libraries support the code generated inline by compilers. As such, most of the procedures are called implicitly as a consequence of a language construct specified by the user, rather than being called explicitly by the user with a CALL statement. Those language support procedures which are independent of higher level language use the facility prefix OTS$. They include:

- Language-independent initialization and termination
- Error and exception-condition processing procedures
- Datatype conversion

Language-Specific Support (FOR$, BAS$, COB$, PAS$)

Each of the language-specific support libraries is generally composed of:

- I/O processing procedures
- File processing procedures
- Compiled-code support procedures
- Compatibility procedures
- System procedures

String Processing (STR$)

The string processing procedures allocate and deallocate dynamic strings. They also perform a wide variety of string manipulation functions, such as comparing, locating a character, concatenating, extracting a substring, performing arithmetic operations on decimal strings, and translating ASCII to EBCDIC code.

System Procedures

VAX programs written in the higher-level languages may call the operating system directly. However, since some languages cannot easily pass arguments in the form that system services require, and some languages use data types that system services cannot properly handle (i.e., dynamic strings), some LIB$ routines have been provided to handle the input and output arguments correctly.

VAX SORT/MERGE

The VAX SORT/MERGE utility may be run interactively, as a batch job, or called from a user-written VAX language program.

The SORT utility allows the user to reorder data from one to ten input files into a single output file in a sequence based upon user-specified key fields within the input data records. If the user does not wish to physically reorder the input, SORT can be used to extract key information and store the sorted information as a permanent file. That file can be used then to access the original input file in the order of the key information in the sorted file.

SORT provides four sorting techniques:

- **Record sort** produces a reordered data file by moving the entire contents of each record during the sort. A record sort can be used with any acceptable VMS input device and can process any valid VAX-11 RMS (Record Management Services) formatted file.
- **Tag sort** produces a reordered data file by moving only the record keys and Record File Addresses (RFAs) during the sort. SORT then randomly reaccesses the input file to create a resequenced output file according to those record keys. The tag sort method may conserve temporary storage, but can accept only input files residing on disk.
- **Address sort** produces an address file without reordering the input file. The address file contains RFAs (Record's File Address),a pointer to each record's location in a file which can later be used as an index to read the database in the desired sequence. Any number of address files may be created for the same database. A customer master file, for example, may be referenced by customer-number index or sales territory index for different reports. Address sort is the fastest of the four sorting methods.
- **Index sort** produces an address file containing the key field of each data record and a pointer to its location in the input file. The index file can be used to randomly access data from the original file in the desired sequence.

The MERGE utility permits the user to merge data from two to ten similarly sorted input files. It merges the data according to key field(s) defined by the user and generates a single output file. All input files to be merged must be in the same sorted sequence, i.e., the MERGE key fields must be a proper subset of the equivalent SORT key fields.

The following example illustrates the sorting of a sales record file by customer last name. The name of the initial file is SALES.DAT. Each record contains six fields: date of sale, department code, salesperson, account number, customer name, and amount of sale. The numerical ranges listed below the set of records indicate the position and size of each information field within the record.

DATE	DPT	SALESP	ACCT	CUST-NAME	AMT
091580	25	Fielding	980342	Coolidge Carol	24999
091580	25	Sanchez	643881	McKee Michael	2499
091580	25	Bradley	753735	Rice Anne	10875
091580	19	Arndt	166392	Wilson Brent	1298
091580	28	Meredith	272731	Karsten Jane	4000
091580	25	Bradley	829582	Olsen Allen	3350
091580	19	Erkkila	980342	Coolidge Carol	7200
1-7	8-10	11-21	22-28	29-58	59-65

The user may now rearrange the sales records in file SALES.DAT according to any of the file's information fields. For instance, to sort the file in alphabetic order of customer's last name, the user would type the following command sequence:

```
$ SORT/KEY=(POSITION=29,SIZE=30) SALES.DAT BILL-
ING.LIS<RET>
```

In this command sequence, the user is defining the SORT key to be the customer's last name and the output file to be BILLING.LIS.

The user may now obtain a listing of the sorted data file by using either the TYPE or PRINT commands.

```
$ TYPE BILLING.LIS<RET>
```

```
DATE     DPT  SALESP    ACCT    CUST-NAME         AMT

091580   19   Erkkila   980342  Coolidge Carol    7200
091580   25   Fielding  980342  Coolidge Carol   24999
091580   28   Meredith  272731  Karsten Jane      4000
091580   25   Sanchez   643881  McKee Michael     2499
091580   25   Bradley   829582  Olsen Allen       3350
091580   25   Bradley   753735  Rice Anne        10875
091580   19   Arndt     166392  Wilson Brent      1298
```

To perform the MERGE function, the MERGE utility expects presorted data files upon which to operate. In the following example, MERGE is operating upon two presorted (by alphabetic order) sales data files, STORE1.FIL and STORE2.FIL.

```
STORE1.FIL
DATE     DPT  SALESP    ACCT    CUST-NAME             AMT

091580   19   Erkkila   980342  Coolidge Carol        7200
091580   25   Fielding  980342  Coolidge Carol       24999
091580   28   Meredith  272731  Karsten Jane          4000
091580   25   Sanchez   643881  McKee Michael         2499
091580   25   Bradley   829582  Olsen Allen           3350
091580   25   Bradley   753735  Rice Anne            10875
091580   19   Arndt     166392  Wilson Brent          1298

091580   20   OConnor   358419  Beaulieu Ronald       1598
091580   04   Docus     980342  Coolidge Carol      575500
091580   25   Fielding  669011  Fernandez Felicia    12000
091580   35   Leith     848105  Kingsfield Stanley    5550
091580   04   Kramer    561903  Landsman Melissa    230000
091580   20   OConnor   643881  McKee Michael          995
091580   19   Erkkila   454389  VanDerling Julie      5480
```

To merge the two data files, the user must type the following command sequence:

```
$                    MERGE/KEY=(POSITION=29,SIZE=30)
STORE1.FIL,STORE2.FIL- CENTR.FIL<RET>
```

The user has indicated in the above command sequence that the files are to be merged via the alphabetic order of the customer's last name. The user can examine the output file via the PRINT or TYPE commands.

```
$ TYPE CENTR.FIL<RET>
```

```
DATE     DPT  SALESP     ACCT     CUST-NAME             AMT

091580   20   OConnor    358419   Beaulieu Ronald      1598
091580   19   Erkkila    980342   Coolidge Carol       7200
091580   25   Fielding   980342   Coolidge Carol      24999
091580   04   Docus      980342   Coolidge Carol     575500
091580   25   Fielding   669011   Fernandez Felicia   12000
091580   28   Meredith   272731   Karsten Jane         4000
091580   35   Leith      848105   Kingsfield Stanley   5550
091580   04   Kramer     561903   Landsman Melissa   230000
091580   25   Sanchez    643881   McKee Michael        2499
091580   20   OConnor    643881   McKee Michael         995
091580   25   Bradley    829582   Olsen Allen          3350
091580   25   Bradley    753735   Rice Anne           10875
091580   19   Erkkila    454389   VanDerling Julie     5480
091580   19   Arndt      166392   Wilson Brent         1298
```

VAX SORT/MERGE Features

The VAX SORT/MERGE utility can perform the following functions:

- Reorder data files into ascending or descending order by up to ten user-specified keys
- Merge as many as ten sorted input files into one sorted output file
- Create reordered address files of RFAs and keys for software use
- Supports fixed, variable, and VFC records
- Sort or Merge ASCII character keys in ASCII or EBCDIC sequence
- Supports sequential, relative, and indexed sequential files
- Supports character, decimal, binary, unsigned binary, F-, D-, G-, and H- floating data types
- Determines its own work file requirements based on input file RMS information received
- Can be controlled by a command string or specification file
- Efficiency may be tuned for a particular application
- Accepts input files from any VAX/VMS input device
- Will output to any VAX/VMS output device
- May be invoked by a single command string, or can prompt the operator for input and then output file specification
- Responds with unique SORT/MERGE error messages in VAX/VMS format
- Prints out statistics upon completion, when requested
- Optional sequence checking of input files on merge

VAX SORT/MERGE supports the following key formats:

- ASCII character data
- ASCII and EBCDIC collating sequence
- Binary, packed decimal, and zoned decimal data
- Unsigned binary and F-, D-, G-, and H- floating
- String decimal data format can be:

  leading separate sign
  leading overpunched sign
  trailing separate sign
  trailing overpunched sign

SORT/MERGE as a Set of Callable Subroutines

SORT and MERGE can be used as a set of callable subroutines from any native VAX language. This subroutine package provides two functional interfaces to choose from: a file I/O interface and a record I/O interface.

SORT and MERGE subroutines are callable from the VAX-11 COBOL language using the standard COBOL SORT and MERGE verbs. (SORT/MERGE is callable from any VAX supported language that supports the VAX Calling Standard.)

For either interface, the user can supply a key comparison routine. This feature allows the user additional flexibility.

File and Record I/O Interfaces

The file I/O interface allows the user to specify the input files and an output file to SORT or MERGE. SORT then reads the data from the input file(s) and sorts the data into the output file. MERGE also reads the data from the input file(s) and merges it into one output file.

The record I/O interface allows the user to pass each individual data record to SORT/MERGE, let SORT/MERGE order them and then receive each record back in the correct order, individually, from SORT/MERGE.

Any program can use either the SORT or MERGE subroutine interface with any of the VAX native mode languages that support the VAX Calling Standard.

DOCUMENT FORMATTING UTILITY

Designing and producing printed materials can be simplified through the use of the DIGITAL Standard RUNOFF (DSR) utility. DSR reduces the number of interactions needed for preparation of a document by allowing both textual correction and formatting change to be executed in the same pass over the file. And since text changes do not affect the basic design, documents can be updated without extensive retyping.

The input to DSR is a file containing the text of the document and the DSR instructions. The output file is the print-ready document. After the program has been run, the original file remains available for further editing.

Formatting instructions consist of commands and flags. Command lines are signalled by a period in position one and may contain one or more commands and text. Within the text are special characters—called flags—which specify character enhancements such as underlined text or boldface characters.

Filling and Justifying

DSR commands set left and right margins, so that the user may enter text without concern for line width or variable spacing between words. The DSR program will **fill** and **justify** the text when it is run. Filling is the successive addition of words to a line until one more word would exceed the right margin. DSR justifies the line by adding enough spaces between words to expand the line to the right margin.

DSR Default Modes

When an input file is processed by the DIGITAL Standard RUNOFF utility, certain default actions are performed that do not depend upon command or flag entries for their execution. These actions are similar to those performed during the preparation of a manually typed document.

DSR default modes provide:

- A standard typewriter page size of 8½″ × 11″
- Sequential page numbering
- Page width of 60 characters
- Single spacing
- Automatic tab settings for every eighth print position
- Automatic filling and justifying

Page Formatting

The page formatting commands control the appearance of each page of output. For example, there are page formatting commands to enable or suspend page numbering, produce and format titles and subtitles, or force the printer to advance to a new page.

Another page formatting command allows a conditional page advance, based on the number of lines left on the page. This capability can be used to guarantee that text which should appear on a single page (e.g., tables, lists) will not be broken up.

For example:

.LAYOUT 2,5	The 2 says page titles will be flush right on odd pages, flush left on even pages; pages will be numbered sequentially at center bottom with 4 blank lines after the body of text.

Title Formatting
Title formatting commands provide page, title, and subtitle information for all pages. Such actions as placing only the chapter heading on the first page of a chapter and printing any subtitles are provided for by the title formatting commands.

For example:

TITLE King Lear	Makes a title of King Lear.

Subject-Matter Formatting
Subject-matter formatting includes commands for managing the design and appearance of text, such as making a ragged right-hand margin, indenting a paragraph, skipping a number of lines, centering a line of text, underlining, hyphenating, and overstriking. Parts of the text may be formatted differently from one another, and commands may be combined. For example, a user has the option of having lists justified or having them with ragged margins.

For example:

.LM 5 .RM 58	Set the left margin at space 5 and the right margin at space 58
.NF	Disables filling: causes a new line in the input file to produce a new line in the ouput file
.NJ	Disables justifying: lines are ragged right
.BR	Causes a break: current line is output without being filled or justified
.S or .SK 2	Skips two blank lines
.PG	Causes a .BREAK, then starts the next page

.TP 25	Tests the page to see if 25 lines remain, so that certain material that needs to stay together (e.g., lists) will
.CENTER	Centers subsequent line of text on the next page
.TS 3,7,9,15,26,...	Sets up to 32 new tab stops to override the default tab stop values
.P 4,2,3	Formats paragraphs in which: first word is indented 4 spaces; there are 2 blank lines between paragraphs; there must be at least 3 lines remaining on the page for the paragraph to be started on the next page

Graphic, List, and Note Formatting
It often becomes necessary to accommodate graphics, lists, and tables, or to allow for special notes to be inserted. Footnotes also have to be prepared in such a way as to fit on the appropriate pages of the final document.

For example:

.FIG 24	Leaves 24 lines for a figure to be inserted
.FIG DEF 30	Leaves 30 lines, including at the top of the next page, for a figure
.LIST 1, '*'	Sets up a list with 1 blank line between items and an asterisk marking each item
.LE	Identifies the start of an element
.DLE"(",LL,")"	Establishes a user-specified display format for lists: in this case, sequential, lowercase letters will be enclosed in parentheses

.HL 1 Plays .HL 2 King Lear .HL 3 Tragic Flaw	These commands provide a properly numbered and formatted outline: 14 Plays 14.33 King Lear 14.3.2 Tragic FLaw--The Definition of Tragic Flaw...

Miscellaneous Formatting
Several useful DSR commands help the user to add nonprinted comments to the source file, to gather externally located files into the input, to exert conditional control, and to set or display time and date.

For example:

.IF complete	Processes the lines following only if /VARIANT:COMPLETE was given on the command line
!appendix C is 200 pages	DSR ignores comments
.REQ "APNDXC.RNO"	Processes all of APNDXC.RNO before continuing with next line
.ELSE complete	Marks the end of the line to process because of the IF, and starts the alternative
.F.J.SK10 or .S10; Contact the author...	Allows commands and text in one line
.ENDIF complete	Marks the end of a group of conditionally processed lines

Flags
Flags are special characters (e.g., an ampersand) that perform specific operations (e.g., underlining). The specified operation is invoked when the character is recognized as a flag by DSR. Certain special characters initially are recognized by default.

For example:

fix#some#space	The SPACE flag (#) fixes one nonexpandable space whenever it occurs

R-&D	The ACCEPT flag (-)prevents DSR from interpreting the ampersand in R&D as an underline flag

Index and Table of Contents

DSR has powerful facilities for creating indexes and tables of contents easily. There are commands to generate one-column or two-column indexes. The TOC program generates tables of contents.

For example:

.X Satire	Creates an index entry for Satire. DSR gives it the current page number
.ENTRY Parody>see Satire	Provides a cross reference to the index

Running the DSR Program

DSR is initiated by entering the following command:

```
RUNOFF filespec <RET>
```

After processing the file, DSR terminates.

For example:

$RUNOFF MYBOOK	Processes MYBOOK.RNO and produces MYBOOK.MEM as output

Various qualifiers can be placed on the command line. Examples are:

/FORMSIZE 55	Sets page to 55 lines rather than the default of 60 lines
/PAGES:"3-1: 3-16, 4-1: 4-16"	Prints only pages 3-1 through 3-16 and 4-1 through 4-16
/DEBUG:echo	Traces the operation of any DSR commands defined by the parameter by echoing each execution in the output file
/INDEX:drama.bix	Creates an index file called drama.bix
/CONTENTS: poems.btc	Creates a table of contents file called poems.btc
/OUTPUT:TT:	Puts the output on the terminal

DEC/CMS

The Digital Equipment Corporation Code Management System (DEC/CMS) is an optional software product that provides software developers with a tool to help manage files of an ongoing project.

DEC/CMS maintains a library of project elements, each consisting of one or more related files. A source file and the command file for compiling and linking that program could constitute an element of the library. Elements are stored in the library as a succession of generations; that is, each time an element is modified, a new generation of that element is added to the library. Historical generations of source and other text files are stored efficiently by storing only their differences. CMS figures out the differences. A history is kept of all movements of files into and out of the project library.

A generation, or line of decent, may be identified by a generation number or by a user-defined class name. A class may denote a base level, a release, the current stable version, a debugging version, or any other characteristic agreed upon by users for their project.

DEC/CMS enables users to:

- keep ASCII text files in a project library
- retrieve previous file generations
- get a report of who modified a file, including when and why the modification was made
- learn the origin of each line of a file, either as an annotated listing or as comments in a file
- manage concurrent modifications
- merge separately developed modifications
- keep related files together as a single element
- relate the generation of one element to the corresponding generations of other elements

COMMANDS

Each CMS command is invoked from the VAX/VMS command level to perform a specific function. Each command returns to the VAX/VMS command level where the user may edit, compile, and test in the usual manner. DEC/CMS provides the following commands:

ANNOTATE	Produces an annotated listing of any element in the library. The annotations describe the element and its ancestors, and indicate the origin of each line of the element (that is, the generation in which each line was inserted or most recently modified).
CREATE	Creates a new element or class.
FETCH	Similar to the RESERVE command described below, except that the element is not reserved for modification. The copy placed in the user's working directory may not be used to create a new generation of the same element.
INSERT	Puts an element generation into a class.
REPLACE	Creates a new generation of an element that the user has reserved. The files of the new generation are moved to the project library from the user's working directory. The new generation is a successor of the generation the user obtained when the element was reserved.
RESERVE	Places a copy of a generation in the user's working directory and notes that it is intended for modification. The entire element is reserved against the concurrent modification by another user. A user may have several elements reserved at the same time. Optionally, a remark may be inserted into each line to show the origin of the line (see ANNOTATE above). A qualifier allows another generation of the same element to be merged with the copy supplied to the user.

SET LIBRARY	Identifies the user's project library at the start of the session.
SHOW	Displays chronological and status information. For any generation, the command can give the author, creation date, creating commands, and author's remark. This information can be obtained for a generation's ancestors or decendents as well. Also, the SHOW command can list all elements of the library, all that are reserved, or all that have a generation in a given class. All or a portion of the project history can be displayed, and the display can be limited to unusual events.
UNRESERVE	Cancels an existing reservation.
VERIFY	Performs consistency checks on the library, and recovers from a malfunction by nullifying a partially completed transaction.

CHAPTER OVERVIEW

The large collection of language processors is described in this chapter. Included is information on language extensions beyond industry standards and special features of the VAX language environment. Some sample programs—particularly for COBOL—are printed in the text.

Topics include:

- VAX Common Language Environment
- High-Level VAX Languages
- Assembly Language
- Host Development Languages

CHAPTER 5
PROGRAMMING LANGUAGES

INTRODUCTION

The VAX/VMS operating system provides a complete program development environment. In addition to the assembly language, MACRO, it offers the optional higher-level languages commonly needed in engineering, scientific, commercial, instructional, and implementation applications—FORTRAN, COBOL, BASIC, PL/I, PASCAL, C, CORAL 66, BLISS-16, and BLISS-32. The VAX/VMS operating system provides the tools to write, assemble or compile, and link programs, as well as to build libraries of source, object, and image modules. User applications may employ more than one language, and the ability of languages to call one another allows concatenation of application segments written in a variety of languages, provided they satisfy certain criteria.

These VAX language processors produce native object code, and take advantage of the native instruction set and 32-bit architecture of the VAX hardware.

In addition, there is the host development mode programming environment which provides support for the PDP-11 FORTRAN IV/VAX to RSX and MACRO-11 languages. These produce compatibility mode object code.

VAX COMMON LANGUAGE ENVIRONMENT

An important feature provided by VAX is a "common language" environment. To put it another way, the VAX languages adhere to a specific set of standards, and these standards include:

- A symbolic debugger interface
- Use of the symbolic traceback facility
- Use of the Common Run Time Library
- Conformance to the VAX calling standard, which allows calls among any set of VAX languages and calls to VAX/VMS system services and to SORT and FMS subroutines
- Common handling of exceptions
- Use of VAX-11 RMS (Record Management Services) for record handling

The elements of the common language environment are briefly described here. For more detailed information, see the Index for the appropriate pages.

Symbolic Debugger Interface

The VAX/VMS operating system provides facilities to aid the debugging of programs written in native mode. It accomplishes this via a program known as the symbolic debugger (DEBUG). DEBUG can be linked with a program image to control image execution during development. It can be used interactively or can be controlled from a command procedure file. The debugging language is similar to the VAX/VMS command language. Expressions and data references are similar to those of the source language used to create the image being debugged.

Debugging commands include the ability to start and interrupt program execution, to step through instruction sequences, to call routines, to set break or trace points, to set default modes, to define symbols, and to deposit, examine, or evaluate virtual memory locations.

Symbolic Traceback Facility

The VAX/VMS operating system supports the Symbolic Traceback Facility. This is a runtime facility that aids programmers in finding errors by describing the call sequences that occurred prior to the error. The traceback facility is automatic and does not require that any special qualifiers be included with the FORTRAN or LINK commands (but it can be suppressed by specifying NOTRACE with the LINK command).

When an error condition is detected, the error message is displayed by the Run Time Library indicating the nature of the error and the address at which the error occurred (user program counter). This is followed by the traceback information, which is presented in inverse order to the calls. For each call frame, traceback lists module name, routine name, source program line, and absolute and relative PC. Using this information, the programmer can usually locate the source of the error in a relatively short period of time.

Common Run Time Library

The VAX-11 Common Run Time Procedure Library contains sets of general purpose and language-specific procedures. User programs call these procedures to perform specific tasks required for program execution. Both VAX-11 MACRO and VAX high-level language programmers can use any of the Run Time Library procedures in any combination. Because all procedures follow the same programming standards and make no conflicting execution assumptions, a language-independent common runtime environment is provided for user programs. Such an environment encourages a user program to be composed of procedures written in different languages, and thus increases programming flexibility.

VAX Calling Standard

The VAX procedure calling standard defines and supports the mechanism for passing arguments between modules of major VAX software subsystems such as languages, VAX-11 RMS (Record Management Services), and the VAX/VMS operating system. The standard facilitates the calling of a procedure written in one language from a program written in another language.

Exception Handling

The mechanism defined by the VAX calling standard are also used by the condition handling facility to signal the occurence of exceptions detected by hardware or software.

VAX-11 RMS

VAX-11 RMS (Record Management Services) is the technique programmers use to handle record I/O within programs. VAX-11 RMS routines are system routines that provide an efficient and flexible means of handling files and their data. Typically, VAX-11 RMS routines allow the programmer to create a file and:

- Accept new input
- Read or modify data
- Produce output in a meaningful form

High-level language programmers normally use the I/O facilities of their particular language to perform record and file operations. These operations are implemented using the VAX-11 RMS facilities. VAX-11 MACRO programmers can use the VAX-11 RMS routines directly within their programs.

VAX-11 RMS routines are an integral part of the operating system. The programmer need not perform any special linking or declaring of global entry points for the routines. Furthermore, calls to VAX-11 RMS routines are consistent with the VAX calling standard.

HIGH-LEVEL VAX LANGUAGES

VAX-11 BASIC

The VAX-11 BASIC product gives the VAX user the benefits of a highly interactive programming environment and a high-performance development language. It combines the features of the PDP-11 BASIC-PLUS-2 and RSTS/E BASIC-PLUS languages with the performance benefits provided by a VAX language that is fully integrated with the VMS environment.

The VAX-11 BASIC language is a highly extended implementation language. It provides powerful mathematic and string handling facilities, support for symbolic characters, and full RMS indexed, sequential, and relative I/O operations.

VAX-11 BASIC can be used as if it were either an interpreter or a compiler. A fast RUN command and support for direct execution of unnumbered statements (immediate mode) gives the VAX-11 BASIC user the "feel" of an interpreter. However, this product can also be used in compilation mode, where it generates native-mode object modules like the other VAX compilers. In either case, the VAX-11 BASIC system generates optimized VAX native mode instructions which have extremely fast execution times. Typical compilation speeds are up to 3,000 lines per minute and computations will generally execute up to five times faster than the same programs on a PDP-11 system.

Following is a brief overview of the general characteristics of the VAX-11 BASIC language.

General Characteristics

The VAX-11 BASIC system generates inline native VAX instructions in both its RUN and its compilation modes. The code produced takes advantage of VAX/VMS native mode capabilities, including:

- Direct calls on operating system service routines, even in immediate mode
- Transparent access to DECnet communications software
- Direct calls to the Common Run-Time Library and standard system utilities, including VAX-11 SORT/MERGE
- Direct calls to separately compiled native mode procedures written in any language that utilizes the VAX procedure calling standard
- Program sizes up to 2 billion bytes are allowed
- All modules are position-independent (PIC) and can be run as fully re-entrant code
- The VAX symbolic debugger has full support for the VAX-11 BASIC language

The code generated by the VAX-11 BASIC system uses the standard VAX/VMS traceback facility for determining the source of runtime errors. If a fatal program error should occur, an English message is printed identifying the module and line number where the error occurred. The English text, the traceback, and the integrated BASIC HELP utility provide a powerful program debugging environment.

Object modules produced by the VAX-11 BASIC system can be linked with native mode modules produced by other language processors including the BLISS, COBOL, FORTRAN, PASCAL, and MACRO processors.

Structured Programming

Structured programming constructs add some of the features of a block structured language (such as the PASCAL language) to the BASIC language to allow complex programs to be written without recourse to subroutines or obscure programming techniques. This makes programs easier to write and maintain.

Figure 5-1 illustrates a record defined by a MAP statement, successive retrievals by the use of a GET statement, and iteration controlled by a WHILE...NEXT statement block.

```
100    !   --------------------------------------------                                                        &
       !                             EMPLOYEE RECORD DEFINITION(S)                                            &
       !                                                                                                      &
       !                             LINE 100: THE "GENERAL DEFINITION"                                       &
       !                             LINE 200: THE "EXPANDED DEFINITION"                                      &
       !
           MAP (REC1)                STRING EMPLOYEE.RECORD = 36,                                             &
                                     REAL      RATE,                                                          &
                                     INTEGER ENDFLAG
       !
           MAP (REC1)                STRING LAST.NAME = 20,                                                   &
                                     STRING FIRST.NAME = 12,                                                  &
                                     STRING MID.INITS = 4,                                                    &
                                     REAL      FILL,                                                          &
                                     INTEGER FILL                                                             &
200    !   --------------------------------------------!
       !
       !
           FILE.NAME.1$ = "EMPLOYEE.DAT"
       !
           OPEN FILE.NAME.1$ AS FILE #1,SEQUENTIAL, ACCESS READ, MAP REC1
       !
           TOTAL.RATES = 000000.00
       !
       !
```

Figure 5-1 Sample Structured Basic Program

```
300    !    ---------------   COMPUTE SUM OF RATES IN FILE ---------------
       !
       !
            WHILE NOT ENDFLAG
       !
                                GET #1
                                TOTAL.RATE = TOTAL.RATE + RATE
       !
            NEXT
       !
       !
400    !    ---------------   REPORT CUMULATIVE SUM BELOW   ---------------
       !
       !
            PRINT "TOTAL.RATE: $";TOTAL.RATE
       !
       !
500    !    ---------------   REPORT COMPLETED: CLOSE FILE(S)   ---------------
       !
            CLOSE #1
       !
999         END
```

Figure 5-1 Sample Structured Basic Program cont'd

The SUBPROGRAM and FUNCTION constructs in the VAX-11 BASIC language have structured END and EXIT statements. In addition, it allows the use of statement modifiers which allow conditional or repetitive execution of the statement without requiring the user to construct unnecessary loops or blocks. Any non-declarative statement in the VAX-11 BASIC language can have one or more statement modifiers. The BASIC statement modifiers include FOR, IF, UNLESS, UNTIL, and WHILE constructs. Each of the statements in Figure 5-2 illustrates the use of a statement modifier:

```
100       A(I) = A(I) + 1               FOR      I = 1 TO 100
  !
200       PRINT SUMMARY.DATA            IF       OPTION.1 AND REPORT="MONTHLY"
  !
300       PRINT FNHOUSE.PAYMENT         UNTIL    RATE < 123.45
  !
400       GET #1                        WHILE    EMPLOYEE.NUMBER < 76000
  !
500       GOSUB 12300                   UNLESS   ERROR.FLAG
  !
600       PRINT "NORMAL EXIT"           IF       TOTAL > 1000   UNLESS ERRORS > 0
```

Figure 5-2   Statement Modifiers

Data Types

The VAX-11 BASIC language increases the number of data types available to the BASIC programmer by allowing the use of 32-bit integer and 64-bit floating point data values. Tabel 7-3 below describes the data types supported by the VAX-11 BASIC language.

Table 5-1 Data Types

Data Type	Meaning
REAL	Specifies that the variable or constant contains floating-point data. The precision depends on the COMPILE command qualifier used. COMPILE/SINGLE specifies 32-bit floating point numbers; COMPILE/DOUBLE specifies 64-bit floating point numbers
WORD	Specifies that the variable or constant contains word-length integer data, regardless of the COMPILE command qualifier used
LONG	Specifies that the variable or constant contains longword integer data, regardless of the COMPILE command qualifier used
INTEGER	Specifies that the variable or constant contains integer data. This data type defaults to the qualifier used at compile-time. If the program is compiled with the /WORD qualifier, integers are 16 bits long; with the /LONG qualifier, 32 bits long
STRING	Specifies that the variable or array contains string data

Declarations

The VAX-11 BASIC language allows implicit declaration of variables. Unless specifically named in a declaration statement, a variable will be declared by its first reference. The PDP-11 BASIC-PLUS-2 convention is to implicitly type a variable or value by the trailing character in its representation, e.g. ABC$ is a STRING variable; XYZ% and 123% are INTEGER; T12, 314159, and 3.14 are implicitly REAL.

Variables can be declared in COMMON, MAP, or DECLARE statements. Both COMMON and MAP statements are used to declare static storage areas—typically I/O records or shared data blocks. If a program has several named common statements with the same name, the common program sections (PSECTs) are stored one after the other. If several MAP statements have the same name, they overlay the same PSECT.

The DECLARE statement is used to explicitly type variables, functions, and constants. Note that the appearance of a variable name in a DECLARE statement implies that that variable will not be in static storage (see MAP, COMMON above).

Finally, the EXTERNAL statement is provided to let the BASIC programmer explicitly declare data types for symbols external to the current program unit, e.g. the name of a VMS system service module, an external BASIC function, or an external constant which is to be global in an application.

Figure 5-3 illustrates the use of COMMON, MAP, DECLARE, and EXTERNAL statements.

```
100  !   ----------------------   COMMON STATEMENTS   --------------------
     !
     COMMON (DATASET1) REAL A,B,C,D,E,F,G,H,O,P,Q,R,S,T,U,V,W,X,Y,Z,                              &
                                  INTEGER       I,J,K,L,M,N                                      &
                                  STRING        S1,S2,S3,S4
     !
     COMMON (DATASET1) LAST.NAME$=10, FIRST.NAME$=5
     !
200  !   ----------------------   MAP STATEMENTS   ---------------------
     !
     MAP (DATASET2)               REAL          PART.NUMBER, COST,                               &
                                  INTEGER       VENDOR.CODE, QA.INDEX,                           &
                                  STRING        VENDOR.ID=40
     !
     MAP (DATASET2)               REAL          FILL,       FILL,                                &
                                  INTEGER       FILL,       FILL,                                &
                                  STRING        VENDOR.NAME = 10, FILL,                          &
                                                VENDOR.TWX = 30
     !
300  !   ------------------------   DECLARE STATEMENTS   -----------------------
     !
     DECLARE                      INTEGER       COUNTER.1, COUNTER.2,
     DECLARE                      REAL          STANDARD.DEVIATION
     DECLARE                      LONG          A.32.BIT.VARIABLE
     DECLARE                      WORD          A.16.BIT.VARIABLE
     DECLARE                      STRING        LAB.NAME = 20
```

Figure 5-3   Declaration Statements

```
    !
        DECLARE              INTEGER    CONSTANT    DEBUG.MODE    = 0, MY.P = 3,
        DECLARE              REAL       CONSTANT    MY.PI         = 3.1416
        DECLARE              STRING     FUNCTION    CONCAT
400     DEF CONCAT( STRING Y, STRING Z )
401             CONCAT = Y + Z
402     FNEND
    !
500     PRINT CONCAT("THIS IS"," THE RESULT")
    !
600 !----------------------------  EXTERNAL STATEMENTS  -------------------------
    !
    !                                                       CAN BE USED FOR VMS SERVICES
        EXTERNAL             INTEGER FUNCTION SYS$ASSIGN
    !
        EXTERNAL             INTEGER FUNCTION SYS$TRNLOG    ! LOGICAL TRANSLATIONS
    !
        EXTERNAL             INTEGER FUNCTION SYS$QIOW      ! SYNCHRONOUS QIO CALL
    !
    !-------------------------------------------------------------
```

Figure 5-3 Declaration Statements cont'd

Files and Records
The VAX-11 BASIC language supports RMS (Record Management Services) sequential, indexed, and relative file organizations. In addition, BASIC applications can access virtual arrays (stored on files), terminal-format files, and block I/O files via RMS.

The OPEN statement in the VAX-11 BASIC language allows specification of file organization, access modes, file sharing, record formats, record size, and file allocation. At the record level, a BASIC program can FIND, GET, PUT, UPDATE, DELETE, or RESTORE any record in a file either sequentially or randomly.

The VAX-11 BASIC language can access files created by other native mode languages, assuming appropriate data representations are maintained with the records.

Symbolic Characters
The BASIC language supports references to symbolic characters—those characters in the 96-character ASCII set which do not print, e.g. NUL, SOH, FF, CR, etc. Figure 5-4 illustrates the use of symbolic characters in a BASIC program.

```
10          PRINT "PROGRAM STARTS...";LF;LF;"AT "+TIME$(0)
  !
            TITLE$ = "SUMMARY REPORT"
  !
            PRINT TITLE$;CR; FOR I = 1 TO 5
!     Bold copy by overprinting
  !
            PRINT
  !
            PRINT A(I) FOR I = 1 TO 10        !      Output report data
  !
            PRINT
  !
99          END
Ready
RUN
TEST5                     28-MAY-1980        17:20
PROGRAM STARTS...
                         AT 05:20 PM
SUMMARY REPORT
   0
   0
   0
   0
   0
   0
   0
   0
   0
   0
Ready
```

Figure 5-4 Symbolic Characters

CALL Facility

The CALL statement allows the BASIC programmer to invoke procedures and functions that are external to the current source module. By using the VAX-11 LINKER utility, procedures written in any of the VAX native mode languages can be invoked, i.e., BASIC routines can call or be called by procedures written in the COBOL, CORAL, FORTRAN, and PASCAL languages.

The CALL statement in the VAX-11 BASIC language has been extended to allow a procedure to pass parameters BY REFerence, BY VALUE, or BY DESCriptor. These mechanisms conform to the VAX procedure calling standard and allow BASIC programs to call VMS service routines and accept returning status values.

Shareable Programs

Applications written in the VAX-11 BASIC language can be made shareable images by the VAX-11 LINKER. The BASIC language generates fully re-entrant PIC code.

Developing BASIC Programs

The VAX-11 BASIC language delivers a high-productivity development environment. The key features of this environment include:

- Automatic line number generation via SEQUENCE command
- Integral line editing with EDIT
- A RUN command which allows a program to be placed directly into execution without requiring a separate LINK operation
- Direct execution of unnumbered BASIC statements, allowing quick verification of algorithms, inspection/change of data values, and invocation of subroutines or functions in a halted BASIC program
- An integral HELP facility helps program debug/development by providing online reference text from the BASIC manual set
- The VAX-11 BASIC system can produce source language listings with embedded diagnostics indicating the line and position of any errors. Fully descriptive diagnostic messages are provided at the point of an error. Many error conditions are caught at compile time. At the user's option, the VAX-11 BASIC system can also output a machine language listing and/or a cross-reference listing
- The VAX symbolic debugger (DEBUG) lets the programmer set breakpoints, and inspect or change the value of variables during execution of a program

Figure 5-5 illustrates the use of several of these features. The text appearing in bold in Figure 5-5 corresponds to user input, the remaining text is supplied by the BASIC system.

```
100     !-------------------INPUT A FILE NAME, COUNT NUMBER OF LINES IN IT-
        LINPUT "What file to be opened ",FILE.NAME$
        F.NAME$ = EDIT$(FILE.NAME$,32%)
        OPEN F.NAME$ FOR INPUT AS FILE #1
        ON ERROR GOTO 900
        LINPUT #1%,TEXT$      FOR I = 1 to 1000000
        STOP
900     LINE. = ERL
        NUMBER. = ERR
        MESSAGE$ = ERT$(NUMBER.)
        RESUME LINE 910
910     PRINT "*END, FROM LINE";LINE.;"WITH TEXT: ";MESSAGE$;
        PRINT " - AFTER ";I;"RECORDS"
991     STOP
995     PRINT "*** THE END ***"
999     END
Ready
RUNNH
%BASIC-E-SYNERR, syntax error
        at line 900 statement 4
            RESUME LINE 910
                        ↑
Ready
HELP RESUME
RESUME

The RESUME statement marks the end of an error handling routine,
and returns program control to a specified line number.
Format
        RESUME [<lin-num>]
Examples
        990 RESUME 300
        or
        990 RESUME


Ready
LIST 900
TEST6 28-MAY-1980 17:15
900     LINE. = ERL
        NUMBER. = ERR
        MESSGE$ = ERT$(NUMBER.)
        RESUME LINE 910
```

Figure 5-5 BASIC Program Development Features

```
Ready
EDIT 900 / LINE //
900     LINE. = ERL
        NUMBER. = ERR
        MESSAGE$ = ERT$(NUMBER)
        RESUME 910
Ready
RUN
TEST6 28-MAY-1980 17:16
What file to be opened          ? TEST6.BAS
*END, FROM LINE 200 WITH TEXT: ?End of file on device - AFTER 17 RECORDS
%BAS-I-STO, Stop
-BAS-I-FROLINMOD, from line 991 in module TEST 6
Ready

PRINT MESSAGE$;" FROM FILE";F.NAME$
?End of file on device FROM FILETEST6.BAS
Ready
PRINT F.NAME$;CR; FOR I = 1 TO 5
TEST6.BAS
Ready
```

Figure 5-5 BASIC Program Development Features cont'd

The LOAD Command

A major goal of VAX-11 BASIC is to support a program development *environment*. The LOAD command allows a user to stay in BASIC, even when a program under development involves several separately compiled BASIC subroutines. When a RUN command is issued, any BASIC modules moved into memory by the previous LOAD command are automatically bound together with the module under development and the resulting in-memory image begins execution, i.e., the user is not required to leave BASIC, invoke the LINKER, and use the DIGITAL Command Language (DCL) $RUN command. This speeds program development considerably.

Once an application has been checked out, a final call on the LINKER can be used to create a shareable, native mode, executable image for production use.

Error Handling

The VAX-11 BASIC system allows user-directed error and event handling. Occurence of an error can activate one or more routines which handle the error (or event), and then return control to the point where the error occurred (RESUME), or to the calling program (ON ERROR GOBACK), or to the BASIC system itself for standard cleanup and return of control at the BASIC command level.

In determining the cause of an error, the BASIC program can use the value of: ERR—the error message number assigned by BASIC, ERL—the line number where the error occurred, ERN$—the name of the BASIC module where the error occurred, and ERT$(ERR)—the error message text which the BASIC system would print if the error were not trapped by the program.

Migration to VAX/VMS

Included with the VAX-11 BASIC system is a translator utility which helps to convert BASIC-PLUS programs to VAX-11 BASIC. Generally, OPEN statements and SYS calls need to be modified. However, additional systems-dependent statements may need to be changed as well. For more information, see The BASIC Transportability Manual.

The following are examples of typical changes:

- The MODE expression on an OPEN statement is changed to the corresponding set of keywords, e.g.,
  ```
  OPEN F$ AS FILE #1 MODE2%
  ```
  becomes
  ```
  OPEN F$ AS FILE #1, ACCESS APPEND
  ```
- MAP and DIM statements are moved to occur before OPEN statements
- RSTS/E SYS-CALLS are examined and removed if not supported by the VAX/VMS operating system

Files may be copied over on tape or by using DECnet communications software, and the programs are RUN under VAX-11 BASIC. In the event errors are detected by BASIC, the online HELP facility is used to determine any additional changes needed for correct compilation. A detailed list of differences between VAX-11 BASIC and the PDP-11 BASIC-PLUS-2 (and BASIC-PLUS/EXTEND) can be found in the Users Guides for those products.

Certain features were carried forward from PDP-11 BASIC-PLUS and PDP-11 BASIC-PLUS-2 to VAX-11 BASIC in order to make the move to VAX easier. These include:

- BASIC-PLUS to VAX-11 BASIC Translator utility
- Program RESEQUENCE utility from BASIC-PLUS-2 V1.6
- FIELD statement
- CVT, SWAP, and MAGTAPE functions
- Foreign buffer support

- String arithmetic
- Numerous non-privileged RSTS/E SYS calls
- Virtual arrays

Additional Functions
Additional functions of the VAX-11 BASIC language include the following:

- Powerful string manipulation functions for creating, converting, searching, editing, and extracting character values
- Variable names up to 30 characters long
- Maxiumum length of a single string is 65,535 characters
- Multiple statements on a line
- Multiline IF...THEN...ELSE statements
- Optional use of line continuator "&" and statement separator "\", e.g.,

```
100   PRINT          vs.     100     PRINT          &
      PRINT                  \ PRINT &
      PRINT                  \ PRINT

vs.

100     PRINT       &
      \ PRINT       &
      \ PRINT
```

- DIGITAL Command Language (DCL) pass-through in the BASIC command mode by simply prefixing the DCL command line with a dollar-sign, e.g.,

```
Ready
$DIR     *.BAS, *.OBJ
```

- Provision for up to ten individual BASIC object library files for automatic use at runtime when developing an application using separately-compiled BASIC subroutines

VAX-11 COBOL
VAX-11 COBOL is a high-performance implementation of COBOL. It is based on American National Standard Programming Language COBOL, X3.23-1974, the industry-wide accepted standard for COBOL. Most features planned for the next COBOL standard, based on the specifications in the Draft Proposed Revised X3.23 American National Standard Programming Language COBOL, are also included.

VAX-11 COBOL also supports an embedded Data Manipulation Language (DML) interface to VAX-11 DBMS, Digital's CODASYL- compliant Data Base Management System. Also, it allows access to common record definitions stored in the VAX-11 Common Data Dictionary. VAX-11 COBOL's support of features in the next ANSI COBOL standard, of the VAX Information Architecture, and of other Digital-defined extensions to COBOL makes possible a wider range of COBOL applications on the VAX.

VAX-11 COBOL is properly defined as an implementation of ANSI COBOL with full support of the following:

- full Level 2 Nucleus Module
- full Level 2 Table Hanlding Module.
- full Level 2 Sequential I/O Module
- full Level 2 Relative I/O Module
- full Level 2 Indexed I/O Module
- full Level 1 Report Writer Module
- full Level 2 Segmentation Module
- full Level 2 SORT/MERGE Module
- full Level 2 Library Module
- full Level 2 Interprogram Communication Module

Most code in the object module produced by the VAX-11 COBOL compiler is implemented with in-line VAX-11 instructions. The object code takes advantage of such native mode features as:

- many of the VAX-11 string manipulation instructions
- the packed decimal instructions
- direct calls to VAX/VMS
- direct calls to VAX-11 RMS
- direct calls to VAX-11 DBMS
- direct calls to VAX-11 SORT
- direct calls to the Common Run-Time Library
- transparent access to DECnet

The VAX-11 Symbolic Debugger many be used for program development with VAX-11 COBOL. Features supported include the source program display facility in which the COBOL source code is displayed as the debugger traces the program. This reduces the need for sources listings during program development. Other features include complete support of COBOL qualified names, breakpoints, and the examination and setting of program variables.

Object modules produced by the compiler can be linked with native mode object modules produced by other VAX language processors including BASIC, FORTRAN, PL/I, and MACRO.

Structured Programming

VAX-11 COBOL adds some of the features of traditional structured programming languages (such as ALGOL and PL/I) to the VAX-11 COBOL compiler. This facility makes programs easier to develop, understand, and maintain, thereby reducing program development and maintenance costs. The structured programming facilities supported by VAX-11 COBOL include the EVALUATE statement, scope-delimited statements, and the in-line PERFORM statement.

The EVALUATE statement in a CASE-like statement found in modern programming languages and allows the selection of statements to be exeucted, dependent on the state of program variables. Scope-delimited statements simplify COBOL coding that previously required additional GO TO statements and procedure names. The in-line PERFORM statement reduces program complexity by putting logic of the PERFORM in-line.

The following program example from a transaction processing application illustrates the usage of the structured programming facilities in VAX-11 COBOL.

```
INITIALIZE STATE.
 .
 .
 .
PERFORM VARYING I FROM 1 BY 1 UNIT I > 12
        MOVE 0 TO MONTYLY-RETRIEVE-TRANSACTIONS(I)
        MOVE 0 TO MONTHLY-UPDATE-TRANSACTIONS(I)
        END-PERFORM
TRANSACTION-LOOP.
 .
 .
 .
MOVE MONTH-INDEX TO I.
EVALUATE TRANSACTION-TYPE
WHEN "RETRIEVE"
WHEN "retrieve"
        READ TRANS-FILE AT END
                MOVE "EOF" TO TRANS-EOF-SWITCH
                END-READ
        IF TRANS-EOF-SWITCH NOT = "EOF"
        THEN
                ADD 1 TO MONTHLY-RETRIEVE-TRANSACTIONS(I)
                END-IF
        WHEN "UPDATE"
        WHEN "update"
                REWRITE TRANS-REC
                ADD 1 TO MONTLY-UPDATE-TRANSACTION(I)
        WHEN OTHER
                DISPLAY TRANSACTION-TYPE "is an invalid transac-
                tion"
                ADD 1 TO TRANS-ERROR-CNT
                END-EVALUATE.
        GO TO TRANSACTION-LOOP.
.
.
.
```

The example illustrates the usage of the in-line PERFORM statement whose scope is delimited by END-PERFORM. The in-line PERFORM loop initializes monthly transaction counts in preparation for the subsequent transaction analysis. The EVALUATE statement performs the transaction analysis and illustrates the typical usage of this statement: a set of actions to be executed, dependent on the state of a program variable (e.g., TRANSACTION-TYPE). For the cases not specifically mentioned, the (catch-all) WHEN OTHERS imperative statement sequence is executed which, in this example, does exception reporting and a count of the transaction errors. The scope-delimiters are END-PERFORM, END-READ, END-IF, and END-EVALUATE. These help to organize the program and to make the program more understandable and maintainable.

Data Types

VAX-11 COBOL supports the data types specified in the ANSI COBOL Standard. VAX-11 COBOL also supports, as extensions, the packed decimal (COMP-3), floating point (COMP-1), double floating (COMP-2), and address (POINTER) data types.

The following is a summary of the data types supported by VAX-11 COBOL:

- Numeric DISPLAY Date
  - Trailing overpunch sign
  - Leading overpunch sign
  - Trailing separate sign
  - Leading separate sign
  - Unsigned
  - Numeric-edited
- Numeric COMPUTATIONAL Data
  - Word fixed binary
  - Longword fixed binary
  - Quadword fixed binary
- Packed-Decimal Data (COMPUTATIONAL-3)
  - Unsigned packed decimal
  - Signed packed decimal
- Floating Point Data
  - F_floating (COMPUTATIONAL-1)
  - D_floating (COMPUTATIONAL-2)
- Alphanumeric DISPLAY Data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric-edited
- Address Data
  - Pointer

Contained Programs and CALL Facilities

VAX-11 COBOL supports both the contained programs and CALL statement facilities. Contained programs allows the nesting of one or more contained subprograms in a containing program within a source module. A containing progam may call any of its directly contained subprograms. Moreover, resources such as variables, files, alphabets, symbolic characters, and use procedures defined in a containing program may be referenced in the contained subprogram, provided such resources are defined in the containing program with the GLOBAL attribute. Thus, the contained programs facility allows the sharing of programs and data within the same source module.

The CALL statement enables a COBOL programmer to execute routines that are external to, or contained in, the source module in which the CALL statement appears. The VAX-11 COBOL compiler produces an object module from a source module. The object module file can be linked with other VAX object modules to produce an executable image. Thus, COBOL programs can call external routines written in other VAX-11 languages including BASIC, FORTRAN, PL/I, and MACRO.

The CALL statement has been extended by allowing arguments to be passed BY REFERENCE (the default in COBOL), BY CONTENT, BY DESCRIPTIOR, and BY VALUE. The BY REFERENCE and BY CONTENT argument-passing mechanisms are defined by the next ANSI COBOL standard. The BY DESCRIPTOR and BY VALUE argument-passing mechanism are Digital extensions to COBOL and are useful in calling VAX/VMS system service routines and common run-time library procedures. These argument-passing mechanisms conform to the VAX calling standard. Also, a COBOL program can receive a returned status value from the routine it calls via the GIVING clause associated with the extended CALL statement.

Other extensions to VAX-11 COBOL that are useful in accessing the VAX/VMS environment from COBOL are the external constants (VALUE IS EXTERNAL), address data, and the SUCCESS/FAILURE class conditions.

The external constants facility gives the COBOL programmer access to values that are known at link-time only. The address data extensions to VAX-11 COBOL include:

- USAGE IS POINTER clause
- VALUE IS REFERENCE clause
- SET TO REFERENCE statement

USUAGE IS POINTER specifies that the associated variable is to contain an address value; the VALUE IS REFERENCE clause allows compile-time initialization of a pointer variable to the address of COBOL data. The SET TO REFERENCE statement allows run-time initialization of a pointer variable to the address of COBOL data. The SUCCESS/FAILURE class condition allow a COBOL program to test the low-order bit of a returned status variable from a system service routine call.

COBOL Data Manipulation Language (DML)

VAX-11 COBOL supports the COBOL Data Manipulation Language interface to VAX-11 DMBS, Digital's CODASYL-compliant Data Base Management System. Digital refers to the DML interface as an "embedded DML" because no preprocessor techniques are used by the compiler in the translation of the DML statements. Instead, the VAX-11 COBOL compiler translates directly the DML statements to calls on the Data Base Control System (DBCS) component of VAX-11 DBMS.

This DML facility is an intergal part of the VAX Informaton Architecture and consists of the following:

- the DB statement in the Sub-Schema Section
- the USE FOR DB-EXCEPTION declarative
- the database special registers
- the data manipulation verbs

The DB statement specifies the subschema and schema that a DML program uses. The subschema and schema define the sets, record types, and realms that the DML program manipulates. The USE FOR DB-EXCEPTION declaratives specify error handling procedures for database exception conditions that may arise during DML program execution. The database special register DB-CONDITION identifies specific database exception conditions. The data manipulation verbs enable a DML program to navigate through a database, to test the state of a database, and to create, update, and delete records in a database. Some of the DML verbs supported are:

- READY - Begin database transaction
- FIND - Find record in database
- GET - Get current record in database
- STORE - Store record in database
- MODIFY - Update record in database
- ERASE - Erase record(s) in database
- COMMIT - Terminate database transaction; change database
- ROLLBACK - Terminate database transaction; no change to database

The following program example from a database transaction processing application illustrates the use of the DML facilities in VAX-11 COBOL.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.     DMLRETRIEVE.
DATA DIVISION.
SUB-SCHEMA SECTION.
DB TRANSUBSCHEMA WITHIN TRANSCHEMA.
LINKAGE SECTION.
01 RET-KEY                      PIC X(7).
01 RT-INFO                      PIC X(73(.
PROCEDURE DIVISION USING RET-KEY, RET-INFO GIVING DB-
CONDITION.
DECLARATIVES.
```

```
RETRIEVAL-HANDLER-SECT SECTION.
        USE FOR DB-EXCEPTION.
RETRIEVAL-HANDLER.
        ROLLBACK.
        EXIT PROGRAM.
END DECLARATIVES.
RETRIEVAL-SECTION SECTION.
RETRIEVE-REC.
        READY CONCURRENT RETRIEVAL.
        MOVE RET-KEY TO TRANSKEY.
        FIND FIRST TRANSREC USING TRANSKEY.
        GET TRANSREC.
        MOVE TRANSINFO TO RET-INFO.
        ROLLBACK.
        EXIT PROGRAM.
```

This program is a COBOL subprogram designed to find and return information from the TRANSCHEMA database to the caller of the subprogram. The DB statement shows that the TRANSUBSCHEMA subschema is to be used for the TRANSCHEMA schema (database). The program is given a lookup key, RET-KEY, as input to locate the record in the database with the FIND statement. The GET statement retrieves the record into memory and returns the associated information (via RET-INFO) to the caller. The USE FOR DB-EXCEPTION procedure handles any database exception conditions that may arise during the execution of the READY, FIND, or GET statements. If this execution procedure is invoked due to such an error condition, the specific database exception condition, specified in the special register DB-CONDITION, is returned to the caller (via the GIVING option) of the subprogram. The ROLLBACK statement terminates the database transaction and leaves the database unchanged.

Files and Records
VAX-11 COBOL Sequential I/O, Relative I/O, and Indexed I/O modules meet the full ANSI Level 2 standard. The Language's Level 2 Indexed I/O module statements enable VAX-11 COBOL programs to use the VAX-11 RMS multikey indexed record management services to process files. These files can be accessed sequentially, randomly, or dynamically usng one or more indexed keys to select records. VAX-11 COBOL has full variable-length record capability for all three I/O modules.

VAX-11 COBOL supports the EXTERNAL files capability for all three I/O modules. This facility allows a program to open a file in one separately compiled program and perform record operations in another separately compiled program.

VAX-11 COBOL has extended COBOL by supporting the special registers RMS-STS, RMS-STV, and RMS-FILENAME. These special registers give the COBOL program access to the VAX-11 RMS status return values and file specification for each I/O operation execution. These special registers are defined in addition to the file status values specified in the ANSI COBOL standard.

An additional extension to VAX-11 COBOL is the file sharing and record locking facilities. These facilities are defined for interactive applications where multiple, concurrent access to a file is required. The facilities include exclusive, concurrent read-only, and concurrent read/write access to a file. Automatic and manual record locking capabilities are supported to protect multiple accessors to the same record in a file.

Report Writer Facility

VAX-11 COBOL supports the full Report Writer Module. The report writer is a facility that places its emphasis on the organization, format, and contents of an output report. Although a report can be produced with the standard COBOL I/O verbs, the Report Writer facility is a much more concise facility for report structuring and generation. Much of the Procedure Division coding required to produce reports in the traditional manner is done automatically by the VAX-11 COBOL Report Writer Control System. Based on the report group description entries in the COBOL program, the report writer control system automatically:

- Moves data
- Constructs print lines
- Counts lines on a page
- Numbers pages
- Produces heading and footing lines
- Recognizes the end of logical data subdivisions
- Updates sum counters

Hence, the VAX-11 Report Writer improves programmer productivity and produces programs that are more cost-effective to maintain.

SORT/MERGE Facility

The VAX-11 COBOL SORT/MERGE module meets the full ANSI standard and permits performing sort and merge operations at the CO-

BOL source language level without requiring the programmer to understand the VAX-11 SORT interface. The COBOL SORT/MERGE capability includes sorting and/or merging one or more files in the same source module, specifying one or more sort/merge keys (in ascending or descending order) for each file, and the option to use either standard or user-specified input/output procedures. The VAX-11 COBOL SORT/MERGE facility supports the sorting/merging of veriable length records and input/output files of differing file organizations.

Source Library Facility
VAX-11 COBOL supports the full ANSI COBOL Library facility. All frequently used data descriptions and program text sections can be stored in library files available to all programs. These files can then be copied into source programs performing textual substitution in the process.

VAX-11 COBOL has extended the COPY statement by supporting the COPY FROM DICTIONARY statement. This facility allows common record defintions to be copied from the VAX-11 Common Data Dictionary. This facility is an integral part of the VAX-11 Information Architecture. Record definitions may be inserted into the dictionary by VAX-11 DATATRIEVE or by the Common Data Definition Language utility.

Debugging COBOL Programs
The VAX-11 compiler produces source language listings with embedded diagnostics indicating line and position of error. Fully descriptive diagnostic messages are listed at the point of error. Many error conditions are checked at compile time, varying from simple informational indications to severe error detections. At the user's option, the compiler can also produce a machine language listings, a map of file names, data names, procedure names, external program names, subschema information, and a cross reference listing. The maps and cross reference listing may be shown in alphabetic order or in order of declaration. The cross reference line numbers on which data-names/procedure-names are defined are indicated and destructive references to date are distinguished from read-only references.

When a fatal error occurs at run time, an error message identifying the cause of the error is displayed to the user. Additionally, the system traceback facility prints the sequence of routine invocations active at the time of the fatal error. For each routine invocation, traceback displays the module name, routine name, and source line number in which either an invocation to another user routine occurs or the fatal error itself occurs.

The VAX-11 Symbolic Debugger may be used for program development with VAX-11 COBOL. Features supported include the source program display facility. By using the facility, the COBOL source code may be displayed at breakpoints and tracepoints. This reduces the need for source listings during program development. Other significant features include full support of COBOL qualified names, breakpoints, examination and setting of program variables.

VAX-11 COBOL also supports the ANSI conditional compilation facility: debug lines. This facility allows "D-lines" to be included conditionally in the compilation, depending on the presence of the WITH DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph. The feature, however, requires editing and recompilaton of the source program. To overcome this limitation, VAX-11 COBOL has extended the conditional compilation facility by providing a compile-time qualifier, /CONDITIONALS, to indicate the inclusion or omission of debug lines in the compilation.

VAX-11 COBOL-74 Translator Utility

The VAX-11 COBOL-74 Translator Utility is helpful to those users migrating from PDP-11 COBOL and VAX-11 COBOL-74 to the VAX-11 COBOL compiler. This utility produces a translated source program and a listing with flags indicating those language elements that could not be mechanically translated and therefore require further investigation by the programmer.

Some of the differences between VAX-11 COBOL and PDP-11 COBOL or VAX-11 COBOL-74 that require such a translator are:

- different allocation and alignment techniques
- different methods of specifying file optimization attributes
- different methods of handling variable length records

Fortunately, most differences are transparent to the programmer, and moving programs form PDP-11 COBOL or VAX-11 requires little (is some cases, no) programmer work.

Source Program Formats

The VAX-11 COBOL compiler accepts source programs that are coded using either the ANSI standard format or a shorter, easy-to-enter Digital terminal format. Terminal format is designed for use with the Digital interactive fields and allows the user to enter horizontal tab characters and short text lines.

The REFORMAT utility reads COBOL source programs that are coded using Digital terminal format and converts the source statements to the ANSI standard format accepted by other COBOL compilers throughout the industry. It also has the inverse option to accept programs written in ANSI standard format and to convert the source statements to Digital terminal format. This offers the advantages of saving disk space and compile-time processingwhen programs are initially migrating from a non-Digital COBOL system to VAX-11 COBOL.

Additional Features

Some additional features of the VAX-11 COBOL compiler are:

- Subscripts can be arithmetic expressions
- Subscripting and indexing are interchangeable.
- The CONTINUE statement is included. It transfers control to the next executable statement and can replace conditional or imperative statements.
- The AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs are included.
- The INITIAL and COMMON clauses on the Program-Id are included:
- User-defined alphabets are included.
- Alter statement is included.
- CALL data-name is included. Both on OVERFLOW and EXCEPTION are supported.
- CANCEL statement is fully implemented.
- INITIALIZE statement is fully implemented.
- Complete string handling facility of COBOL are supported including the INSPECT, STRING, and UNSTRING statements. The reference-modification (substringing) feature is fully supported.
- SET statement supporting mnemonic-names, condition-names, and the Digital-defined extension of SET TO SUCCESS/FAILURE is included.
- Independent segments (segments 50 and above) of the Segmentation module are included.
- WRITE advancing mnemonic-name and associated Special-Names C01 is included.
- Use of source file libraries by the COPY statement is supported.
- The Digital extension of non-numeric literals as arguments in the BY REFERENCE, BY CONTENT, and BY DESCRIPTIOR argument-passing mechanisms is included.
- Single-quote-limited non-numeric literals, a Digital extension, are supported in addition to the standard double-quote-delimited non-numeric literals.
- De-edited MOVE operations are supported.

- OPEN EXTEND on relative and indexed files is included.
- ALPHABETIC-UPPER and ALPHABETIC-LOWER class conditions are implemented.
- The ALLOWING extension on the READ, START, REWRITE, and WRITE statements for manual locking of records in the interactive file sharing environment is included.
- The READ REGARDLESS extension that allows the reading of records in a file sharing environment, independent of record locks held on the record is supported.
- The UNLOC statement, a Digital extension, for explicit unlocking of records in the file sharing environment is implemented.
- ACCEPT AT END, a Digital extension, is included.
- Thirty-one character user-names are supported.

VAX-11 FORTRAN

VAX-11 FORTRAN is an optional native-mode language processing system for VAX/VMS. The language specifications are based on American National Standard FORTRAN X3.9-1978 (commonly called FORTRAN-77). The VAX-11 FORTRAN compiler supports this standard at the *full-language* level. Also, it provides optional, switch-selectable support for many industry-standard FORTRAN features based on FORTRAN-66, the previous ANSI standard. The qualifier /NOF77 will invoke such FORTRAN-66 features.

The VAX-11 FORTRAN compiler performs the following functions:

- Produces highly optimized VAX native object code
- Makes use of the VAX floating point and character string instructions
- Produces shareable code

File Manipulation

OPEN and CLOSE statements extend the file management characteristics of the FORTRAN language. An open statement can contain specifications for file attributes that direct file creation or subsequent processing. Attributes include: file organization (sequential, relative, indexed); access method (sequential, direct, keyed); protection (read-only, read/write); record type (formatted, unformatted); record size; and file allocation or extension. The program can also specify whether the file can be shared, and whether the file is to be saved or deleted when closed. An ERR keyword can modify the OPEN statement and specify the statement to which control is transferred if an error is detected during OPEN.

Of particular interest is the VAX-11 FORTRAN support for the Indexed Sequential Access Method (ISAM), a powerful keyed input/output file access capability. The VAX-11 FORTRAN language is able to create, read, and write indexed (and relative) files. In addition, it is able to reference a relative or indexed file already created by another language (for instance, the COBOL language), provided the file and data formats and the key information are compatible. Some specifics of FORTRAN ISAM are covered below, while more details on the various file structures and access methods are included in Chapter 12 *I/O Services*.

Simplified I/O Formats
List-directed and NAMELIST-directed input and output statements provide a method for obtaining simple sequential formatted input or output without the need for FORMAT statements. Using list-directed input, values are read, converted to internal format, and assigned to the elements of the I/O list. On output, values in the I/O list are converted to characters and written in a fixed format according to the data type of the value.

The NAMELIST statement and the associated forms of input/output statements provide a simplified means of transmitting lists of data to and from files. The list of items that can be transferred is specified in a NAMELIST statement. The associated I/O statement refers to the list of items to be transferred by including the name of the NAMELIST as a control parameter. NAMELIST I/O statements do not contain explicit I/O lists; therefore, it is possible to reference a single name in a simple I/O statement and get an effect similar to a statement with a long list and a reference to a complicated format statement.

Character Data Type
A program can create fixed-length CHARACTER variables and arrays to store ASCII character strings. The VAX-11 FORTRAN language provides a concatenation operator, substring notation, CHARACTER relational expressions, and CHARACTER-valued functions. CHARACTER constants, consisting of a string of printable ASCII characters enclosed in string quotes, can be assigned symbolic names using the PARAMETER statement. Operations employing CHAR strings are more efficient and easier to use than their analogs using arithmetic data types. The VAX/VMS operating system provides a set of character manipulation procedures that are FORTRAN-callable (e.g., LIB$LOCC, locate a character in a string).

Source Program Libraries
The INCLUDE statement provides a mechanism for writing modular, reliable, and maintainable programs by eliminating duplication of source code. A section of program text that is used by several program units, such as a COMMON block specification, can be created and maintained as a separate source file. All program units that reference the COMMON block then merely INCLUDE this common file. Any changes to the COMMON block will be reflected automatically in all program units after compilation. INCLUDE also allows the user to include modules from the text libraries. VAX-11 FORTRAN provides a text library that contains FORTRAN source for many VAX/VMS symbols.

Calling External Functions and Procedures
FORTRAN programs can call MACRO assembly language subroutines and the system services using the VAX-11 procedure calling standard. Special operators exist for passing argument values directly, by reference, or by descriptor. A special operator also exists for obtaining the location of argument values used by the system services procedures.

Shared Programs
The FORTRAN language can be used to create shareable images which can be used by any program written in a native programming language.

Diagnostic Messages
Diagnostic messages are generated when an error or potential error is detected. Errors detected during compilation are reported by the compiler, and include source program errors, such as misspelled variable names, missing punctuation marks, etc.

Source program diagnostic messages are classified according to severity: F (Fatal), E (Error), or W (Warning). F-class messages indicate errors that must be corrected before compilation can be completed. Object code is not produced. E-class messages indicate that an error was detected that is likely to produce incorrect results; however, an object file is generated. W-class messages are produced when the compiler detects acceptable but non-standard syntax; or when it corrects a syntactically incorrect statement. The message indicates the existence of possible trouble in executing the program.

The VAX-11 FORTRAN compiler optionally produces diagnostic messages for VAX FORTRAN extentions to ANSI FORTRAN-77. This flagger can check both syntax and/or source form extentions.

Debugging FORTRAN Programs

The VAX-11 FORTRAN language provides facilities to aid the debugging of programs written in native mode. It accomplishes this via a program known as the interactive symbolic debugger. The debugger can be linked with a native program image to control image execution during development. It can be used interactively or can be controlled from a command procedure file. The debugging language is similar to the VAX/VMS command language. Expressions and data references are similar to those of the source language used to create the image being debugged. Debugging statements can be conditionally compiled.

Debugging commands include the ability to start and interrupt program execution, to step through instruction sequences, to display source statements, to call routines, to set break or trace points, to set default modes, to define symbols, and to deposit, examine, or evaluate virtual memory locations.

Compiler Operations and Optimizations

The VAX-11 FORTRAN compiler accepts sources written in the FORTRAN language and produces an object file which must be linked prior to execution. The compiler generates VAX-11 native machine language code. It will also generate an optional cross-reference listing.

During compilation, the compiler performs many code optimizations. The optimizations are designed to produce an object program that executes in less time than an equivalent non-optimized program. Also, the optimizations are designed to reduce the size of the object program.

The VAX-11 FORTRAN compiler performs the following optimizations:

- Constant folding—constant expressions are evaluated at compile-time.
- Compile-time constant conversion.
- Compile-time evaluation of constant subscript expressions in array calculations.
- Constant pooling—only a single copy of a constant is allocated storage in the compiled program. Constants that can be used as immediate mode operands are not allocated storage. For example, logical, integer, and small floating point constants are generated as immediate mode or short literal operands wherever possible.
- Inline expansion of statemment functions.
- Argument list merging—if two function or subroutine references have the same arguments, a single copy of the argument list is generated.

- Branch instruction optimizations for arithmetic or logical IF statements.
- Elimination of unreachable code—an optional warning message is issued to mark unreachable statements in the source program listing.
- Recognition and replacement of common subexpressions.
- Removal of invariant computations from DO loops.
- Local register assignment—frequently referenced variables are retained (if possible) in registers to reduce the number of load and store instructions.
- Assignment of frequently used variables and expressions to registers across DO loops.
- Reordering expression evaluation to minimize the number of temporary registers required.
- Delaying negation/not to eliminate unary complement operations.
- Flow-Boolean optimizations.
- Jump/Branch instruction resolution—the Branch instruction is used wherever possible to eliminate unnecessary Jump instructions. This reduces code size.
- Peephole optimizations—the code is examined on an operation-by-operation basis to replace sequences of operations with shorter and faster equivalent operations.

When debugging FORTRAN programs, the programmer can disable optimizations that would remove unreferenced statement labels, FORMAT statement labels, and immediately referenced labels. This ensures that all statement labels are available to the debugger.

VAX-11 FORTRAN LANGUAGE ELEMENTS

A FORTRAN program consists of FORTRAN statements and optional comments. In the first category are assignment, control, I/O, format, and specification statements.

Following are three tables: Table 5-1 is a brief summary of FORTRAN-77; Table 5-2 is a summary of VAX-11 FORTRAN extensions to the ANSI standard. And Table 5-3 is a summary of traditional FORTRAN IV (industry-compatible) features supported by VAX-11 FORTRAN.

Table 5-2 FORTRAN-77 Language Summary

ASSIGNMENT STATEMENTS
variable = expression
ASSIGN label TO variable

Control Statements

GO TO
DO
CONTINUE
CALL
RETURN
PAUSE
STOP
ARITHMETIC IF, LOGICAL IF

IF-THEN-ELSE	Allows conditional expression evaluation. VAX-11 FORTRAN provides the block IF statements:
	IF (logical expression) THEN
	ELSE IF (logical expression) THEN
	ELSE
	ENDIF
	These are structured programming control statements which provide a readable and reliable means of writing conditional statements.

END

INPUT/OUTPUT STATEMENTS
OPEN
CLOSE
INQUIRE
READ
WRITE
LIST DIRECTED INPUT/OUTPUT
REWIND
BACKSPACE

FORMAT STATEMENTS
FORMAT

ADDITIONAL DATA TYPES
The CHARACTER data type can be used to declare and manipulate fixed-length CHARACTER variables and arrays. CHARACTER

expressions can contain concatenation operators (//), substring references, and references to CHARACTER variables, array elements, and functions. A CHARACTER assignment statement can be used to assign a character value to a character variable or substring. Built-in functions are provided for locating a substring within a character expression, computing the length of a character dummy argument, and for conversions between character values and integer-valued ASCII character codes.

SPECIFICATION STATEMENTS

IMPLICIT	
IMPLICIT NONE	Overides all default implicit types.
type var1,var2,...,varn	Type is one of: LOGICAL, INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, BYTE
DIMENSION	
COMMON	
EQUIVALENCE	
EXTERNAL	
INTRINSIC	
PARAMETER	
DATA	
PROGRAM	
SAVE	

USER-WRITTEN SUBPROGRAMS

name (var1, var2,...)	
—expression	
FUNCTION	
SUBROUTINE	
BLOCK DATA	
ENTRY statement	Multiple entry points in a single program unit

Table 5-3 VAX-11 FORTRAN Extensions

Thirty-one-character symbolic names	
CALL extensions	Permit interfacing to VAX/VMS system service procedures using the VAX-11 calling standards.
Hexadecimal and octal constants and field descriptors	
Bit Manipulation	Intrinsic functions used to set, clear, test, extract, or move bits.
DO WHILE END DO	Structured looping control constructs.
Additional data types and type declaration statements	BYTE, LOGICAL*1, LOGICAL*2, **LOGICAL**, LOGICAL*4, INTEGER*2, **INTEGER**, INTEGER*4, **REAL**, REAL*4, **DOUBLE PRECISION**, REAL*8, **COMPLEX**, COMPLEX*8, DOUBLE COMPLEX, COMPLEX*16, **CHARACTER*n** NOTE Names on the same line above are synonyms. Those in boldface are the ANSI standard ones.
Indexed File I/O	
Keyed READ	Key types: INTEGER*2, INTEGER*4, CHARACTER with generic, and approximate key match
Indexed file WRITE REWRITE statement DELETE statement UNLOCK statement	

Table 5-3 VAX-11 FORTRAN Extensions cont'd

Single-record locking in
shared file environments
for relative and indexed
organization files

Data initialization in type-declaration statements

Array Subscripts using general expressions
of any numeric data type

End-of-Line comments

Conditional Compilation of debugging statements

Default FORMAT width

Logical Operations on integers

INCLUDE statement

CALL extensions

INTEGER Data Type Defaults

Table 5-4 Traditional FORTRAN IV (Industry-Compatible) Features

FORTRAN IV Compatible Direct Access I/O:
(Where u = logical unit #, and r = record #)

DEFINE FILE
READ (u'r)
WRITE (u'r)
FIND (u'r)

ENCODE Statement
DECODE Statement
Hollerith processing of character data
Character literals
(Optional) One-trip DO loops instead of FORTRAN-77
zero-trip DO loops

Device-oriented I/O Statements:

TYPE
ACCEPT
PRINT

VAX-11 PASCAL

VAX-11 PASCAL, a re-entrant native mode compiler, is an extended implementation of the PASCAL language as defined by Jensen and Wirth in *PASCAL User Manual and Report* (1974).

PASCAL language has become an increasingly popular general purpose language. It implements a well-chosen, compact set of general purpose language features. In addition, portability is easily achievable in PASCAL programs.

Block structuring and flexible data types make the PASCAL language a good language for commercial users. It is also suitable for systems programming and research applications.

The VAX-11 PASCAL language takes advantage of the VAX hardware floating point, character instruction sets, and virtual memory capabilities of the VAX/VMS operating system. Features common to other languages of the VAX/VMS operating system are available through the VAX-11 PASCAL language, including:

- VAX-11 Symbolic Debugger support
- Separate compilation of modules
- Standard call interface to routines written in other languages
- Access to VAX/VMS system services

At compile time, options available to the process include:

- Runtime checks for illegal assignment to set and subrange variables, and illegal array subscripts
- Cross-reference listing of identifiers
- Source program listing
- Machine code listing

Standard PASCAL provides a modular, systematic approach to computerized problem solving. Major features of the language are:

- INTEGER, REAL, CHAR, BOOLEAN, user-defined, and subrange scalar data types
- ARRAY, RECORD, SET, and FILE structured data types
- Constant identifier definition
- FOR, REPEAT, and WHILE loop control statements
- CASE and IF-THEN-ELSE conditional statements
- BEGIN...END compound statement
- GOTO statement
- GET, PUT, READ, WRITE, READLN, and WRITELN I/O procedures
- Standard functions and procedures

In addition, the VAX-11 PASCAL language incorporates the following extensions to standard PASCAL, some of which are common in PASCAL implementations:

1. Lexical
   - Upper- and lower-case letters treated identically except in character and string constants
   - New reserved words: MODULE, OTHERWISE, SEQUENTIAL, VALUE, %DESCR, %IMMED, %INCLUDE, and %STDESCR
   - The exponentiation operator, **
   - Hexadecimal and octal constants
   - DOUBLE constants
   - $ and (underscore) characters in identifiers

2. Predefined data types
   - DOUBLE
   - SINGLE

3. Predefined procedures
   - CLOSE (f)
   - FIND (f,n)
   - OPEN (f,...)
   - DATE (a)
   - HALT
   - LINELIMIT (f,n)
   - TIME (a)

4. Predefined functions
   - LOWER (a,n)
   - SNGL (d)
   - UPPER (a,n)
   - EXPO (r)
   - CARD (s)
   - CLOCK
   - UNDEFINED (r)

5. Extensions to procedures READ and WRITE
   - READ (or READLN) of user-defined scalar type
   - READ (or READLN) of string
   - WRITE (or WRITELN) of user-defined scalar type
   - WRITE (or WRITELN) of any data using hexadecimal or octal format

6. %INCLUDE directive

7. VALUE initialization

8. OTHERWISE clause in CASE statement

9. External procedure and function declarations

10. Dynamic array parameters

11. Extended parameter specifications
    - %DESCR
    - %IMMED
    - %IMMED PROCEDURE and %IMMED FUNCTION
    - %STDESCR

12. Separate compilation of procedures and functions. (A separate compilation unit is termed a MODULE and several routines may be part of a MODULE. Each MODULE is eventually embedded in a host or main program.)

The OPEN, CLOSE and FIND procedures extend the I/O capabilities of the PASCAL language. The OPEN procedure can contain file attributes that define the creation or subsequent processing of the file. A FIND procedure is another extension to the language for direct access to sequential files of fixed length records. The standard I/O procedures of GET, PUT, READ, WRITE, READLN and WRITELN are also available in the VAX-11 PASCAL language.

The extended parameter specifications %DESCR, %IMMED, and % STDESCR are added to the PASCAL language to denote the method of argument passing when calling a system service, procedure, or function not written in the PASCAL language (for example, in the VAX-11 FORTRAN or MACRO languages.)

VAX-11 PL/I

The VAX-11 PL/I compiler supports the PL/I language defined in the American National Standard (ANSI) General Purpose Subset. This subset, defined by ANSI standard X3.74, is a proper subset of the full ANSI PL/I (ANSI X3.53-1976). The PL/I language is a versatile language that is easily adapted to commercial, scientific, and systems programming applications.

The General Purpose Subset includes the most widely used features of the full PL/I language. It excludes features that were more error-prone, difficult to understand or use, and that tended to be implementation-dependent.

VAX extensions to the Subset provide additional language features that allow PL/I programmers to take advantage of the facilities of the VAX/VMS operating system and its components.

Extensions provided in the VAX-11 PL/I language include selected features of the full PL/I language that were excluded from the subset because of their implementation cost on computers with restricted memory and/or address space.

VAX-11 PL/I programmers can thus choose to restrict their programs to the General Purpose Subset, ensuring compatibility with other implementations of the subset. Or they can take advantage of the full PL/I features and VAX extensions in programming applications.

Applications

Data processing applications can take advantage of the extensive character-handling functions and data structuring capabilities of the PL/I language. By declaring variables within a structure, the program can easily refer to entire records or to fields within records by referencing the name of the structure or the name of a variable within it.

In addition, the VAX-11 PL/I language provides extended access to the features of VAX-11 Record Management Services (RMS). By specification of ENVIRONMENT options or special options supplied for input/output statements, PL/I programs can dynamically specify RMS optimization parameters and values, spool a file to a printer or batch job queue, and set or change the protection on a file.

The VAX-11 PL/I language supports all RMS (Record Management Services) file organizations, including sequential, relative, and indexed sequential. It also permits block input/output operations. Using PL/I statements, a program can read, write, delete, and update records. Using built-in file handling functions provided by the VAX-11 PL/I language, a program can call RMS file handling services to forward space or backward space a file or volume, to increase the allocation of a disk file, or to obtain information about the properties of a file.

Scientific applications can use the PL/I array-handling capabilities to define arrays of up to eight dimensions. Common arithmetic and trignometric functions are defined within the language. The VAX-11 PL/I language supports all of the VAX hardware's floating-point data types.

System programming applications can use PL/I language features to allocate storage dynamically, process linked lists and queues, and perform a wide range of bit-string functions and operations.

In addition, VAX extensions to the language provide a simple means to refer to VAX/VMS system global symbols and data structures. VAX-11 PL/I programs can take advantage of the VAX linker's allocation of storage by defining variables as read-only or as global symbols.

Full access to all of the VAX/VMS operating system's services and procedures is possible through VAX-11 PL/I extensions to support the VAX-11 Procedure Calling Standard. Procedures written in the PL/I language can call and be called by procedures written in any other native mode language.

Error and Condition Handling

VAX-11 PL/I generates traceback records in the object module of a PL/I procedure, so that when an error occurs at runtime, the VAX condition handling facility can report on the error and provide a module traceback.

Within the PL/I language, extensive condition handling capabilities are available via the ON statement, which allows a program to define the action to take in the event of hardware arithmetic exceptions and errors that occur during file processing.

VAX extensions to the ON statement permit the specification of condition handlers for any specific hardware or software condition that can occur.

Debugging Facilities

The PL/I compiler generates useful diagnostics that signal syntactical errors and language violations. Most compiler messages are two or three lines long and provide information on how to correct the indicated error.

The VAX DEBUG utility supports symbolic debugging of PL/I programs. Programmers can set breakpoints in PL/I programs, examine and change variables, and monitor the calls and function references that occur.

Libraries

The VAX-11 PL/I language is fully compatible with the VAX Run Time Procedure Library and provides additional runtime procedures for language support.

Source file library support is provided by the %INCLUDE statement, which allows a program to specify at compile time an external file from which source statements are to be read. Included files can also be collected in VAX/VMS text file libraries. The VAX-11 PL/I compiler searches specified libraries for the names of the included modules.

Performance

The VAX-11 PL/I compiler is a shareable, native VAX/VMS image that can be run on any supported VAX/VMS configuration. It produces optimized, shareable, VAX/VMS object code that is runtime compatible with other native VAX/VMS language products.

The degree of optimization performed by the compiler can be controlled by the user at compile time, by qualifiers on the PL/I command.

VAX-11 C

VAX-11 C fully supports all of the language features of C, as described in "The C Programming Language", by Kernighan and Ritchie * . The program flow control constructs for logical and efficient program structuring, and the rich assortment of operators that enable an elegant conciseness of expression, are there in VAX-11 C. So, too, are the common runtime routines - only those UNIX-specific routines that cannot be reasonably emulated under VAX/VMS are omitted. VAX-11 C even includes language extentions developed since the Kernighan and Ritchie book was published, including the structure assignment feature.

But VAX-11 C is more than just a faithful implemention of the C programming lanuage. It is a very powerful implementation with an

* "The C Programming Language", B. Kernighan and D. Ritchie, Prentice-Hall, 1978.

impressive collection of features, and, as important, VAX-11 C is an integrated VAX/VMS layered language product; which means that programmers have available to them all of the services and program development aids that the VAX/VMS system provides.

The Language
VAX-11 C is a versatile programming language that combines many of the features of a high-level language with the generality of MACRO.

Program control flow — C uses simple, appropriate English for performing conditional loops (WHILE, FOR, DO), simple decisions (IF — ELSE), and multicase decisions (SWITCH); and for escaping loops or multi-case decisions (BREAK, GOTO label:). These facilities not only aid in creating well-structured programs, but, combined with C's clear statement and expression delimiters, they can provide easy to understand, thus maintain, source code.

Operators — C provides an unusually large array of operators that allow programming with clarity and economy of expression. (Refer to Table 5-4).

Data types — Because C was designed to be a powerful, lean generalist among languages, it uses only the fundamental datatypes commonly represented by computers directly: integers of various, fixed sizes, and single and double-precision floating point. VAX-11 C also provides for user-defined, or enumerated, scalars (ENUM values). ENUM data-types are defined by writing the type name followed by an ordered list of indentifiers that are the constants of that type.

Run-time support — In order to retain its flexibility of application, the C language does not directly support many functions usually attributed to high-level languages; for example, I/O or common math routines. But most implemenations of C include a common set of run-time support routines for accomplishing those tasks. VAX-11 C includes all of the non-UNIX-specific run-time support offered in the Bell Laboratories version (even many of the UNIX-specific routines have been emulated) **and** all of the additional support included in the VAX-11 Common Run-Time Library.

Unique to VAX-11 C — New keywords for sharing data among program modules are offered by VAX-11 C to augment the capability of the standard keywork for passing arguments, EXTERN. The new keywords—GLOBALDEF, GLOBALREF, and GLOBALVALUE, which allow VAX-11 C programs to define and reference global symbols offer an alternative method for dealing with external variables and values. They provide, in addition to enhanced data-sharing among C program modules, a convenient and efficient way for a C function to communicate with MACRO programs, with VAX/VMS system services and data structures, and with other high-level languages that support global symbol definition, such as VAX-11 PL/I.

The Compiler
VAX-11 C has an extremely powerful compiler that generates shareable, position-independent, native VAX object code directly from C source programs (i.e., no separate assembly step) at an average rate in excess of 3000 source statements per minute. In the process, the compiler can perform global and local optimization by, for example, doing global flow analysis, assigning automatic variables to register temporaries, and removing invariant computations from loops, to mention a few. The compiler also does peephole optimizations on the generated machine code. The result: VAX-11 C produces faster and smaller programs.

The VAX-11 C compiler will produce an annotated listing with statement numbers and, optionally, an inline listing of generated machine code, expanded macros, storage allocation map, cross-reference listing of variable usage, and compilation statistics.

The VAX/VMS Programming Environment
What most distinguishes VAX-11 C from other implementations of the language is that it is an integrated constituent of a total VAX/VMS environment. This means VAX-11 C provides C programmers with an easy interface and an exceptional array of services and tools that can maximize their productivity and the efficacy of the programs they produce.

VAX-11 RMS — In addition to performing stream file access conventional among most C implementations, and because it is a VAX/VMS layered language product, VAX-11 C allows all of the features of the VAX-11 Record Management Services (RMS) to be exploited. RMS supports sequential, relative, and indexed file organizations, thus expanding the potential applications for C programs.

VAX/VMS System Services — The VAX-11 C programmer can utilize all of the VAX/VMS System Services, including, for example, the ability to define logical names. By referencing files or devices by logical names, which in turn are defined by the user prior to execution, VAX-11 C programs can be device or file independent; a useful quality for many applications.

The common language environment — All DIGITAL VAX-11 language products, VAX-11 C among them, follow the VAX calling standard. This permits C programmers to call, as subroutines, object modules created using other languages — say VAX-11 FORTRAN or

VAX-11 PL/I — so that particular tasks may be coded in the most suitable language, or proven routines already in use can be applied by the programmer without having to "re-invent the wheel." Of course the inverse is true as well: Programs written in other VAX-11 languages can call routines originally developed in VAX-11 C.

VAX-11 Symbolic Debugger — With the VAX-11 Symbolic Debugger, the VAX-11 C programmer can set breakpoints, and examine and modify the contents of user variables dynamically while the C program is executing. Additionally, if a C program is not performing as expected, program execution can be interrupted, the debugger called, and execution continued.

Compatibility Across Implementations
There are no national or international standards for the C language; however, "The C Programming Language" is generally regarded as the definitive document, along with technical notices subsequently published by the American Telephone and Telegraph Company. But because C is a relatively simple language, even without formal standardization, most programs written in VAX-11 C can be re-compiled successfully using another implementation of the language, or vice versa, usually with few if any modifications.

Certain incompatibilities among implementations do exist, however, especially in machine-specific library routines. To aid creating portable programs, VAX-11 C provides predefined constants ("vms", "vax", and "vax11c") Which can be used, for example, in program control lines to decide whether to compile source code that may not be portable. The VAX-11 C compiler command, CC, also has an optional feature that detects several nonportable program constructions and issues warning messages.

UNIX/VAX, VAX/VMS coexistance — The C programming language was originally developed at Bell Laboratories for creating the UNIX operating system, and it has become the language of choice for may applications developed on that system. As an aid to migrating programs from UNIX systems to VAX/VMS, the VAX-11 C run-time library includes many of the UNIX-specific UNIX/C routines, emulated to run under VMS. Also, VAX-11 C allows UNIX-style stream I/O access to VAX-11 record formats.

Table 5-5 Summary of C Operators

Operator	Example	Result
− [unary]	−a	negative of a
* [unary]	*a	reference to object at address a
& [unary]	&a	address of a
~	~a	one's complement of a
++ [prefix]	++a	a after increment
++ [postfix]	a++	a before increment
−− [prefix]	−−a	a after decrement
−− [postfix]	a−−	a before decrement
sizeof	sizeof(t1)	size in bytes of type t1
	sizeof e	size in bytes of expression e
(type-name)	(t1)e	expression e, converted to type t1
+	a + b	a plus b
− [binary]	a − b	a minus b
* [binary]	a * b	a times b
/	a / b	a divided by b
%	a % b	remainder of a/b
>>	a >> b	a, right-shifted b bits
<<	a << b	a, left-shifted b bits
<	a < b	1 if a < b; 0 otherwise
>	a > b	1 if a > b; 0 otherwise
<=	a <= b	1 if a <= b; 0 otherwise
>=	a >= b	1 if a >= b; 0 otherwise
==	a == b	1 if a equal to b; 0 otherwise
!=	a != b	1 if a not equal to b; 0 otherwise
& [binary]	a & b	bitwise AND of a and b
¦	a ¦ b	bitwise OR of a and b
↑	a ↑ b	bitwise XOR (exclusive OR) of a and b
&&	a && b	logical AND of a and b (yields 0 or 1)
¦¦	a ¦¦ b	logical OR of a and b (yields 0 or 1)
!	!a	logical NOT of a (yields 0 or 1)
?:	a ? e1 : e2	expression e1 if a is nonzero, expression e2 if a is zero

Table 5-5 Summary of C Operators cont'd

Operator	Example	Result
=	a = b	b (assigned to a)
+=	a += b	a plus b (assigned to a)
−=	a −= b	a minus b (assigned to a)
*=	a *= b	a times b (assigned to a)
/=	a /= b	a divided by b (assigned to a)
%=	a %= b	remainder of a/b (assigned to a)
>>=	a >>= b	a, right-shifted b bits (assigned to a)
<<=	a <<= b	a, left-shifted b bits (assigned to a)
&=	a &= b	a AND b (assigned to a)
¦ =	a ¦ = b	a OR b (assigned to a)
↑=	a ↑= b	a XOR b (assigned to a)

VAX-11 BLISS-32

VAX-11 BLISS-32 is a high-level systems implementation language. The BLISS-32 language supports development of modular software according to structured programming concepts by providing an advanced set of language features. It provides access to most of the hardware features of the VAX systems to facilitate programming of time-critical and hardware dependent applications. The BLISS-32 language is specifically designed for the development of operating systemms, compilers, runtime system components, database file systems, communications software, and utilities for use on a VAX system.

The BLISS-32 compiler runs in native mode under the VAX/VMS operating system. It translates BLISS-32 source programs into relocatable object modules that can be linked for execution. BLISS-32 compiled code is optimmized for execution efficiency.

The following features of BLISS-32 are machine independent. Collectively, this set of features is known as "Common BLISS" and can be used to develop transportable programs that will run on VAX, DECsystem-10, DECSYSTEM-20, and PDP-11 systems.

- Modules are compiled separately for modularity and convenient development. Object modules are relocatable and can be linked with other object modules produced by the VAX-11 MACRO assembler or other native mode languages
- The BLISS-32 language provides expressions for describing the actions to be performed and declarations for allocating, describing, and initializing data, and for defining macros and literals, etc.

- The BLISS-32 language is "type-free": all data is manipulated as longwords or 32 bits. Interpretation of data is provided by language operators
- The operators provide a set of operations for integer arithmetic, for comparison, maximization, and minimization of signed integer, unsigned integer, and address values, and for Boolean operations
- Field references allow values to be retrieved from or assigned to any contiguous field from 1 to 32 bits located anywhere in the VAX virtual address space
- Character sequence functions provide for efficient runtime manipulation of character data. Operations include moving, concatenating, comparing and translating character sequences, as well as searching for particular characters or substrings of characters
- IF, CASE, SELECT, and SELECTONE allow the choice of an expression or group of expressions to be executed based on programmed tests
- DO, WHILE, and UNTIL allow general loops that cycle as long as a programmed test is satisfied. The test can be made at either the beginning or the end of the loop
- INCR and DECR allow counted loops that execute a computed number of times under control of a loop variable
- LEAVE allows early termination of the processing of a named block and continuation after the named block. LEAVE may be considered a restricted form of forward-only GOTO, as there is no general GOTO in the BLISS-32 language
- OWN and GLOBAL declarations provide static storage allocation; GLOBAL names are made available to the linker to resolve EXTERNAL data declarations in other modules
- LOCAL, STACKLOCAL, and REGISTER declarations allow dynamic stack-like allocation using either the execution stack or the general registers
- STRUCTURE declarations allow the programmed definition of arbitrary data structures in terms of an accessing algorithm for locating elements of a structure. Predefined declarations for VECTOR, BLOCK, BITVECTOR, and BLOCKVECTOR provide commonly needed structures
- ROUTINE declarations provide subroutines or functions in the BLISS-32 language. Routines are recursive and reentrant, and can be linked in resident libraries for use by multiple processes
- REQUIRE declarations allow source files to be automatically included in the module being compiled

- LIBRARY declarations allow special compiler-produced binary declaration files to be included in the module being compiled. The effect is substantially the same as REQUIRE, but is more efficient because a restricted set of declarations are precompiled into internal form
- MACRO and KEYWORDMACRO declarations allow compile-time definition of both positional and keyword-oriented macros. Macro definition and replacement are in terms of source lexical units called lexemes (atoms, tokens) rather than character text. Macro calls and declarations may be nested and recursive
- %IF, %THEN, %ELSE, and &FI allow conditional compilation of BLISS source depeneding on programmed compile-time tests. These can be used to control expansion of macros
- Lexical functions allow a variety of compile time operations such as concatenation of strings, construction of names, testing properties of macro parameters, testing compiler qualifiers, generating compiler diagnostic messages, and controlling macro expansion

The following features of the BLISS-32 language are specialized for use on VAX systems. They provide precise means to tailor BLISS-32 programs to the special capabilities of VAX systems and the VAX/VMS operating system.

- LINKAGE declarations allow definition of specialized calling sequences for time critical or unusual applications. Options allow for use of CALLS/CALLG/RET or JSB/BSB/RSB type call and return instructions, for passing parameters in VAX general registers or in parameter blocks, for controlling the preservation and use of registers by a routine, and for the sharing of registers across a set of routines as highspeed, common storage. Built-in linkage declarations for the BLISS and FORTRAN languages fully support the VAX calling sequence conventions
- PSECT declarations allow use of link-time program sections for efficient use of the virtual address space. By default, generated code sections are position independent
- BUILTIN declarations allow use of the VAX machine-specific functions for access to VAX features not otherwise provided by the BLISS-32 language. Machine specific functions generally correspond to VAX instructions such as ADAWI, BISPSW, CRC, HALT, INDEX, MTPR, PROBER, REMQUE, etc. Over 50 such functions are provided.(The complete list is shown in Table 5-4)
- ENABLE declarations, together with SIGNAL, SIGNAL-STOP, and SETUNWIND functions, allow use of the VAX/VMS condition handling and error message reporting mechanisms

Table 5-6 VAX Machine Specific Functions

	PROCESSOR REGISTER OPERATIONS
MTPR	Move to a Processor Register
MFPR	Move from a Processor Register
	PARAMETER VALIDATION OPERATIONS
PROBER	Probe Read accessibility
PROBEW	Probe Write accessibility
	PROGRAM STATUS OPERATIONS
MOVPSL	Move from PSL
BISPSW	Bit set PSW
BICPSW	Bit clear PSW
	QUEUE OPERATIONS
INSQUE	Insert entry in Queue
REMQUE	Remove entry from Queue
	BIT OPERATIONS
TESTBITSS	Test for Bit Set, then Set bit
TESTBITSC	Test for Bit Set, then Clear bit
TESTBITCS	Test for Bit Clear, then Set bit
TESTBITCC	Test for Bit Clear, then Clear bit
	BIT OPERATIONS
TESTBITSSI	Test for Bit Set, then Set bit Interlocked
TESTBITCCI	Test for Bit Clear, then Clear bit Interlocked
FFS	Find First Set bit
FFC	Find First Clear bit

Table 5-6 VAX Machine Specific Functions cont'd

EXTENDED ARITHMETIC OPERATIONS

ASHQ	Arithmetic Shift Quad
EDIV	Extended Divide
EMUL	Extended Multiply
INDEX	Index (Subscript) Calculation
CRC	Cyclic Redundancy Calculation

FLOATING POINT CONVERSION OPERATIONS

CVTLF	Convert Long to Floating
CVTLD	Convert Long to Double
CVTFL	Convert Floating to Long
CVTDL	Convert Double to Long
CVTFD	Convert Floating to Double
CVTDF	Convert Double to Floating
CVTRDL	Convert Rounded Double to Long
CVTRFL	Convert Rounded Floating to Long
CMPF	Compare Floating
CMPD	Compare Double

STRING OPERATIONS

MOVTUC	Move Translated Until Character
SCANC	Scan Characters
SPANC	Span Characters

Table 5-6 VAX Machine Specific Functions cont'd

	DECIMAL STRING OPERATIONS
MOVP	Move Packed
CMPP	Compare Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing Numeric
CVTTP	Convert Trailing Numeric to Packed
CVTPS	Convert Packed to Leading Separate Numeric
CVTSP	Convert Leading Separate Numeric to Packed
EDITPC	Edit Packed to Character
	MISCELLANEOUS OPERATIONS
HALT	Halt Processor
ROT	Rotate
ADAWI	Add Aligned Word Interlocked
BPT	Breakpoint
CHMx	Change Mode
CALLG	Call with General Argument List
NOP	No Operating

The VAX-11 BLISS-32 Compiler

The VAX-11 BLISS-32 compiler performs a number of optimizations. These include common subexpression elimination, removal of loop invariants, constant folding, block register allocation, peephole replacement, test instruction elimination, jump vs. branch instruction resolution, branch chaining, and cross-jumping.

The VAX-11 BLISS-32 compiler optionally produces source text and generated code in a format closely resembling a VAX-11 assembly listing. Other options allow the programmer to control the degree of optimization, suppress production of object code, determine types and formats of output listings, generate traceback information, and specify the types of information to be listed at the terminal.

Library and Require Files

The BLISS-32 language provides two methods for including commonly used text into BLISS programs at compile time. These involve use of either Library files or Require files:

- Library Files—These are special files created by the compiler in a previous library compilation and are invoked by the LIBRARY declaration in the BLISS source program
- Require Files—These are source (text) files which are invoked via the REQUIRE declaration in the BLISS source program

Since Library files are "pre-compiled," lexical processing and declaration parsing and checking need not be repeated each time these files are included in a compilation; their use results in a considerable reduction in total compilation time.

The contents of Require files must be fully processed each time the file is used in a compilation. Hence, using Require files will, in general, be less efficient than using Library files. However, since these files operate under a less stringent set of syntactical rules, their use may be warranted in situations where a higher level of flexibility is desired.

Macros

The VAX-11 BLISS-32 language provides an extensive macro-building facility, allowing frequently used groups of declarations or expressions to be expressed in an abbreviated way. Macros are defined via MACRO declarations and are accessed by simple call statements. They are fully expanded at compile time. The BLISS-32 language allows parameters to be specified in the macro definition, thus allowing each block of text to be specialized by the actual parameters passed to it. Macros may be positional or keyword, and may be simple, iterative, or conditional.

Debugging

The VAX-11 BLISS-32 compiler produces a list of error messages showing the source program line on which the error occurred followed by a description of the error. If the error is recoverable, then the compiler will generate a "warning" diagnostic and continue with the compilation process. If the error is serious enough to invalidate the compiler's internal representation of the module, then an "error" diagnostic is generated, and processing ceases following the syntax checking—no object module is produced.

If an error occurs at execution time, the process image can access the VAX DEBUG program. This program may be accessed when the object module is linked with the DEBUG option. The DEBUG program allows the programmer to examine and deposit values in storage, set breakpoints, call routines, trace through a program as it executes, and perform other operations useful in checking out a program. It understands BLISS syntax and permits the use of the user's symbolic names. (See the section on the VAX DEBUG for a further description of VAX debugging facilities.)

Transportability Features

The VAX-11 BLISS-32 language is designed to facilitate transportability, that is, the writing of programs that can be executed on architecturally different machines with little or no modification. The VAX-11 BLISS-16 language, which is discussed later in this chapter, is a high-level implementation language for the development of systems software for use on PDP-11 systems. For DECsystem-10 and DECSYSTEM-20 users, there is the BLISS-36 language. Several language features enhance transportability:

- The high-level language constructs may be transferred from one machine to another with little or no alteration
- Machine-specific functions can be separated from the common, mainline code via modularization, macros, and Library and Require files
- Parameterization allows machine-specific characteristics to be passed to BLISS data structures

The BLISS-32 language's transportability makes it an ideal language for system programming applications—and a desirable alternative to assembly language coding in those applications in which extreme machine dependence is not involved. The following program shows how the VAX-11 BLISS-32 language can call VAX/VMS system services and the VAX Common Run Time Procedure Library to print the current time on SYS$OUTPUT.

SAMPLE PROGRAM

```
;           MODULE showtime( IDENT='1-1' %TITLE 'SHOW TIME', MAIN=timeout)=
;           BEGIN
;
;           LIBRARY 'SYS$LIBRARY:STARLET';          ! Defines System Services, etc.
;
;           MACRO
;       M       desc[] = %CHARCOUNT(%REMAINING),    ! A VAX-11 Style String descriptor
;                         UPLIT BYTE( %REMAINING ) %;
;
;           OWN
;               timebuf:    VECTOR[2],                              ! 64 bit system time
;               msgbuf:     VECTOR[80,BYTE],                        ! Output msg. buffer
;               msgdesc:    BLOCK[8,BYTE]                           ! String descriptor
;                             PRESET( [dsc$w_length]= 80,           !   for output buffer
;                                     [dsc$a_pointer] = msgbuf );
;
;           BIND
;               fmtdesc= UPLIT(DESC('At the tone, the time is ', %CHAR(7), '!15%T' ));
;
;           EXTERNAL ROUTINE
;               lib$put_output :    ADDRESSING_MODE(GENERAL);       ! From VMS RTL
;
;           ROUTINE timeout=
;               BEGIN
;               LOCAL
;                   RSLT:   WORD;                   ! Resultant string length
;
;               $GETTIM( TIMADR=timebuf );          ! Get time as 64 bit integer
;
;       P       $FAOL( CTRSTR=fmtdesc,              ! Format control-string address
;       P               OUTLEN=rslt,                ! Resultant length (only a word!)
;       P               OUTBUF=msgdesc,             ! Output buffer descriptor address
;                       PRMLST=%REF(timebuf));      ! Address of pointer to time block
;
;               MSGDESC[dsc$w_length] = .rslt;      ! modify output descriptor
;
;               lib$put_output( msgdesc )           ! print the formatted time
;
;               END;

                                                                                .TITLE  SHOWTIME SHOW TIME
                                                                                .IDENT  \1-1\

                                                                                .PSECT  $PLIT$,NOWRT,NOEXE,2

68  74  20  2C  65  6E  6F  74  20  65  68  74  20  74  41  00000 P.AAB:  .ASCII  \At the tone, the time is \
                    20  73  69  20  65  6D  69  74  20  65  0000F
                                                        07  00019         .ASCII  <7>
                                        54  25  35  31  21  0001A         .ASCII  \!15%T\
                                                            0001F         .BLKB   1
```

```
                                                  0000001F  00020 P.AAA:  .LONG   31                                           ;
                                                  00000000' 00024         .ADDRESS P.AAB                                       ;

                                                                          .PSECT  $OWN$,NOEXE,2

                                                            00000 TIMEBUF:.BLKB   8
                                                            00008 MSGBUF: .BLKB   80
                                                      0050  00058 MSGDESC:.WORD   80                                           ;
                                                        00# 0005A         .BYTE   0[2]                                         ;
                                                  00000000' 0005C         .ADDRESS MSGBUF                                      ;

                                                                  FMTDESC=            P.AAA
                                                                          .EXTRN  LIB$PUT_OUTPUT, SYS$GETTIM
                                                                          .EXTRN  SYS$FAOL

                                                                          .PSECT  $CODE$,NOWRT,2

                                                       0004 00000 TIMEOUT:.WORD   Save R2                                      ; 0023
                                      52      0000'  CF  9E 00002         MOVAB   MSGDESC, R2                                  ;
                                      5E             08  C2 00007         SUBL2   #8, SP                                       ;
                                                A8   A2  9F 0000A         PUSHAB  TIMEBUF                                      ; 0028
                            00000000G 9F             01  FB 0000D         CALLS   #1, @#SYS$GETTIM                             ;
                                      6F        A8   A2  9E 00014         MOVAB   TIMEBUF, (SP)                                ; 0033
                                               4004  8F  BB 00018         PUSHR   #^M<R2,SP>                                   ;
                                                0C   AE  9F 0001C         PUSHAB  RSLT                                         ;
                                              0000'  CF  9F 0001F         PUSHAB  FMTDESC                                      ;
                            00000000G 9F             04  FB 00023         CALLS   #4, @#SYS$FAOL                               ;
                                      62        04   AE  B0 0002A         MOVW    RSLT, MSGDESC                                ; 0035
                                                     52  DD 0002E         PUSHL   R2                                           ; 0037
                            00000000G 00             01  FB 00030         CALLS   #1, LIB$PUT_OUTPUT                           ;
                                                         04 00037         RET                                                  ; 0023

; Routine Size:  56 bytes,    Routine Base:  $CODE$ + 0000


;         0040  END ELUDOM
;                                      PSECT SUMMARY
;
;      Name                         Bytes                         Attributes
;
;  $OWN$                                96    WRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $PLIT$                               40  NOWRT,  RD ,NOEXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)
;  $CODE$                               56  NOWRT,  RD ,  EXE,NOSHR,  LCL,  REL,  CON,NOPIC,ALIGN(2)


;
;                                             -------- Symbols --------     Blocks
;      File                                   Total    Loaded  Percent       Read
;
;  DBA0:[SYSLIB]STARLET.L32;9                  2783         4        0        103




;                                       COMMAND QUALIFIERS

;      BLISS /LIS/NOOB SHOWTIME

; Size:         56 code + 136 data bytes
; Run Time:     00:01.9
; Elapsed Time: 00:03.7
; Memory Used:  117 pages
; Compilation Complete
```

```
MODULE showtime( IDENT='1-1' %TITLE 'SHOW TIME', MAIN=timeout)=
BEGIN

LIBRARY 'SYS$LIBRARY:STARLET';          ! Defines System Services, etc.

MACRO
    desc[] = %CHARCOUNT(%REMAINING),    ! A VAX-11 Style String descriptor
            UPLIT BYTE( %REMAINING ) %;

OWN
    timebuf:    VECTOR[2],                              ! 64 bit system time
    msgbuf:     VECTOR[80,BYTE],                        ! Output msg. buffer
    msgdesc:    BLOCK[8,BYTE]                           ! String descriptor
                  PRESET( [dsc$w_length]= 80,           !   for output buffer
                          [dsc$a_pointer] = msgbuf );

BIND
    fmtdesc= UPLIT(DESC('At the tone, the time is ', %CHAR(7), '!15%T' ));

EXTERNAL ROUTINE
    lib$put_output :    ADDRESSING_MODE(GENERAL);       ! From VMS RTL

ROUTINE timeout=
    BEGIN
    LOCAL
        RSLT:   WORD;                       ! Resultant string length

    $GETTIM( TIMADR=timebuf );              ! Get time as 64 bit integer

    $FAOL( CTRSTR=fmtdesc,                  ! Format control-string address
           OUTLEN=rslt,                     ! Resultant length (only a word!)
           OUTBUF=msgdesc,                  ! Output buffer descriptor address
           PRMLST=%REF(timebuf));           ! Address of pointer to time block

    MSGDESC[dsc$w_length] = .rslt;          ! modify output descriptor

    lib$put_output( msgdesc )               ! print the formatted time

    END;
END ELUDOM
```

VAX-11 BLISS-16

The VAX-11 BLISS-16 language is a high-level systems implementation language designed specifically for the development of systems software for use on a PDP-11 system. An advanced set of language features supports development of modular software according to structured programming concepts. Access to many of the hardware features of PDP-11 systems is provided in order to facilitate programming of time-critical and hardware dependent applications.

Although the VAX-11 BLISS-16 language runs on a VAX system, the target system for the developed programs is the PDP-11 system. The BLISS-16 cross-compiler runs in native mode under the VAX/VMS operating system and translates BLISS-16 source programs into relocatable PDP-11 object modules that have been optimized for time and space efficiency.

The following features of the BLISS-16 language are machine independent. Collectively, this set of features is known as "Common BLISS" and can be used to develop transportable programs that will run on VAX, DECsystem-10, DECSYSTEM-20, and PDP-11 systems.

- Modules are compiled separately for modularity and convenient development. Object modules are relocatable and can be linked with other BLISS-16 object modules or object modules produced by the compiler or other PDP-11 language processors
- The BLISS-16 language provides expressions for describing the actions to be performed and declarations for allocating, describing, and initializing data, and for defining macros and literals, etc.
- The BLISS-16 language is "type-free": all data is manipulated as words of 16 bits. Interpretation of data is provided by language operators
- The operators provide a set of operations for integer arithmetic, for comparison, maximization, and minimization of signed integer, unsigned integer, and address values, and for Boolean operations
- Field references allow values to be retrieved from or assigned to any contiguous field from 1 to 16 bits within a 16 bit-word
- Character sequence functions provide for efficient runtime manipulation of character data. Operations include moving, concatenating, comparing and translating character sequences, as well as searching for particular characters or substrings of characters
- IF, CASE, SELECT, and SELECTONE allow the choice of an expression or group of expressions to be executed based on programmed tests
- DO, WHILE, and UNTIL allow general loops that cycle as long as a programmed test is satisfied. The test can be made at either the beginning or the end of the loop
- INCR and DECR allow counted loops that execute a computed number of times under control of a loop variable
- LEAVE allows early termination of the processing of a named block and continuation after the named block. LEAVE may be considered a restricted form of forward-only GOTO, as there is no general GOTO in the BLISS-16 language
- OWN and GLOBAL declarations provide static storage allocation; GLOBAL names are made available to the linker to resolve EXTERNAL data declarations in other modules
- LOCAL, STACKLOCAL, and REGISTER declarations allow dynamic stack-like allocation using either the execution stack or the general registers
- STRUCTURE declarations allow the programmed definition of arbitrary data structures in terms of an accessing algorithm for locating elements of a structure. Predefined declarations for VECTOR, BLOCK, BITVECTOR, and BLOCKVECTOR provide commonly needed structures

- ROUTINE declarations provide subroutines or functions in the BLISS-16 language. Routines are recursive and reentrant, and can be linked in resident libraries for use by multiple processes
- REQUIRE declarations allow source files to be automatically included in the module being compiled
- LIBRARY declarations allow special compiler-produced binary declaration files to be included in the module being compiled. The effect is substantially the same as REQUIRE, but is more efficient because a restricted set of declarations are precompiled into internal form
- MACRO and KEYWORDMACRO declarations allow compile-time definition of both positional and keyword-oriented macros. Macro definition and replacement are in terms of source lexical units called lexemes (atoms, tokens) rather than character text. Macro calls and declarations may be nested and recursive
- %IF, %THEN, %ELSE, and &FI allow conditional compilation of BLISS source depeneding on programmed compile-time tests. These can be used to control expansion of macros
- Lexical functions allow a variety of compile time operations such as concatenation of strings, construction of names, testing properties of macro parameters, testing compiler qualifiers, generating compiler diagnostic messages, and controlling macro expansion

The following features of the BLISS-16 language are specialized for use on PDP-11 systems.

- ENVIRONMENT specifies the hardware instructions available on the target PDP-11 (EIS or non-EIS) and controls the optional generation of position independent code
- BLISS-16 generated object code can be mapped to run under I/D space
- BLISS-16 generated object code is compatible with a wide range of DIGITAL supported operating system environments
- PSECT declarations allow use of link-time program sections for efficient use of the address space
- BUILTIN declarations allow use of PDP-11 machine specific functions for access to PDP-11 features not otherwise provided by the BLISS language. Machine specific functions generally correspond to PDP-11 instructions such as: HALT, NOP, RESET, WAIT, BPT, SWAB, MFPS, MTPS, MFPD, and MTPI
- ENABLE declarations, together with SIGNAL, SIGNAL-STOP, and SETUNWIND functions, allow condition handling— the response to an unusual event signaled during the execution of a program

The VAX-11 BLISS-16 Compiler

The VAX-11 BLISS-16 compiler performs global and local optimizations to produce efficient and compact generated code. Each routine definition is treated as a complete unit in compiling the code for that routine.

The VAX-11 BLISS-16 optimizations employed are: common subexpression elimination, removing loop invariants, constant folding, block register allocation, peephole replacement, test instruction elimination, jumps. branch instruction resolution, branch chaining, cross-jumping, constant propagation, and redundant store elimination.

The BLISS-16 compiler optionally produces a side-by-side listing file that shows the source text compiled and the generated code in a format that closely resembles a PDP-11 MACRO assembly listing. Multiple listing options allow selective inclusion or exclusion of source and generated code, source names and source line numbers as commentary to the assembly listing (where feasible), macro expansion and tracing information, and identification of names acquired from library files. A listing file that can be assembled by the MACRO-11 assembler can also be requested.

VAX-11 CORAL 66

The VAX-11 CORAL 66 compiler compiles in compatibility mode and generates native mode object code under the VAX/VMS operating system. The CORAL language, derived from the JOVIAL and ALGOL-60 languages in 1966, is the standard language prescribed by the British government for military realtime applications and systems implementation. A government agency controls the CORAL language standard, which was first widely used in military projects beginning in 1970. Her Majesty's Stationery Office publishes the "Official Definition of CORAL 66."

The CORAL language replaces assembly level programming in a number of commercial, process control, research, and military applications. It is particularly adapted to long-life products requiring flexibility and ease of maintenance.

The VAX-11 CORAL 66 language is a block-structured language. A block is a piece of a program that can be entered only at the beginning. Though internal structures cannot be "seen" from the outside, statements inside a block can "see" out. Sorting is possible, so that programs may be written in which information is accessible for only the time it is required, and no longer. In this way, unwanted interactions among program parts are avoided, and out-of-date information is very quickly forgotten.

To satisfy realtime needs, the CORAL 66 language allows different modules of the same suite of programs to be executed at apparently the same time. A CORAL compiler assumes that any subroutine global to the whole program is likely to be active at the same time as any other, so the compiler assures that such subroutines do not share any local storage. Interactions, however, can be explicitly arranged by the programmer. A program consists of communicators and separately compiled segments. Each segment has the form of an ALGOL 60 block, within which blocks and procedures may be nested to arbitrary depth. In the absence of communicators, block structure would prevent different segments from using common data, labels, command qualifiers, or procedures. The purpose of a communicator is to specify and name those objects which are to be commonly accessible to all segments. The presence of communicators imposes a modular and disciplined approach to programming larger systems where a team of programmers is employed.

In addition to the functionalities prescribed in the Official Definition, the VAX-11 CORAL 66 compiler provides the following features:

- BYTE, LONG (32-bit integer) and DOUBLE (64-bit floating point) numeric types
- Generation of re-entrant code at the procedure level
- Command-qualifier-selectable option to optimize generated code
- Conditional compilation of defined parts of source code
- English text error messages at compile and (optionally) runtime
- Command-qualifier-selectable option to control listing output
- INCLUDE keyword to incorporate CORAL 66 source code from user-defined files
- Command-qualifier-selectable option to read card format

The VAX-11 CORAL 66 language is essentially a high-level, block-structured language possessing certain facilities associated with low-level languages, and is designed for use on small or medium-size dedicated computers. One of the main intentions is that programs written in the CORAL language should be fast to execute, taking up limited quantities of storage, while being easy to write.

The realtime applications of the language are implicit rather than explicit, permitting the utilization of any hardware or special features. Procedures, optionally with parameters, permit communication with and reaction to external events. This is aided further by a direct code facility which enables machine code to be included in the source program for extra efficiency in any critical tasks.

VAX-11 DSM

VAX-11 DSM (DIGITAL Standard MUMPS) is a multiuser data management system and a language processing system. The DSM language is a high-level, interpretive language well-suited for the processing of variable-length string data. It conforms to the American National Standard MUMPS specification X11.1-1977. In addition, it provides a number of extensions.

Interpretive processing of the language means that each line of a DSM routine is executed as it is entered. Routines written in the DSM language do not have to be compiled or linked, making it easier to write, debug, edit and run a routine in one interactive session.

As DSM program lines are entered, the DSM interpreter examines and analyzes each DSM statement and executes the specified operation. It performs error checking during routine execution and reports all errors at the terminal. This reduces problem-solving time, the computer time required to check the routine, and most importantly, the time required to obtain a final running application.

The DSM language has many capabilities, but its basic orientation is procedural. The language is directed primarily toward the processing of variable-length string data, making interactive database systems easier to implement and maintain.

Data Management

The DSM language allows the user to reference data symbolically through variables. A variable can contain either a numeric value or an alphanumeric string.

The VAX-11 DSM system utilizes two types of variables: local and global variables. Local variables are defined solely for the current user (or process). Local variables are not intended for permanent storage, but only for temporary use during the life of the process.

Global variables, or simply globals, are stored on disk. Globals are referenced symbolically using names similar to those of local variables, the only difference being the circumflex (↑) preceding the first character in the variable name. Subscripted global variables form a system of arrays stored on disk, the data of which forms a common database that can be made available to one or more users in the system.

Global arrays are sparse arrays, that is, the system dynamically adds nodes to the array as a user stores data in them, and deletes nodes as a user deletes them. Thus, users never have to preallocate space for globals through dimensioning, nor do they have to explicitly recover disk space when they delete data.

All VAX-11 DSM globals are implemented as VAX-11 RMS (Record Management Services) ISAM (Indexed Sequential Access Method) files. This makes DSM global arrays accessible by other VAX/VMS operating system languages and by DECnet communications software. VAX-11 DSM represents each global by one indexed file. The mapping of the logical structure of a global array into the corresponding ISAM file is transparent to the DSM user. Thus, there is no concept of "opening" and "closing" a global.

In general, global arrays are treated syntactically in the DSM language the same way as local arrays: to create a global, the SET command is issued; to access and manipulate its contents, any number of commands and functions in the DSM language set are used; to delete a global node, the KILL command is issued; and to delete the entire global array, its root node is killed.

This arrangement eliminates the need to be concerned with the physical structure of files when designing a database application (as is the case with some database systems). Using globals, you need only be concerned with the logical relationships between elements of a database.

The Precompiler

The VAX-11 DSM system provides a language precompiler to optimize the execution of DSM routines in an application environment. The precompiler is a component of the VAX-11 DSM interpreter that processes all DSM program lines into a more efficient, intermediate format, called precompiled format, in order to expedite subsequent interpretation.

When a user executes a routine, the interpreter executes the precompiled program. Syntax errors are reported at this point.

When a user stores a routine on disk, the system places both the source and precompiled versions in the DSM routine directory. For a given routine version, the precompilation procedure occurs only once. When users execute a routine from the directory, the VAX-11 DSM system automatically loads the precompiled version.

Because the system saves both routine versions, users can always load, edit, and test DSM routines interactively. The precompilation procedure is repeated if a routine is edited or updated.

The VAX-11 DSM precompiler transforms DSM program lines into precompiled format with the following optimizations:

- It strips comments
- It checks syntax

- It sets up an internal table for line labels which optimizes GOTO statements and DO statements that transfer control to other routine lines
- It evaluates constants and transforms numbers into an internal representation (that is, packed decimal or longword)
- It converts arithmetic statements into Reverse Polish Notation
- It restricts the evaluation of a series of postconditionals to the occurrence of the first false condition. To do this, the precompiler generates code that specifies the appropriate offset to a given instruction

Procedure Calls

The VAX-11 DSM system allows users to access services that are not part of the DSM language through a DIGITAL-implemented extension to Standard MUMPS called the $ZCALL function. Through $ZCALL, a user can call VAX/VMS system services, routines in the VAX-11 Common Run Time Library, or routines written in other languages directly from DSM application routines. For example, the DSM language does not include a square root function. Through the procedure calling mechanism, however, a DSM user can access the corresponding Run Time Library function.

I/O Options

The VAX-11 DSM system provides a subset of the Input/Output (I/O) options of the VAX/VMS operating system. Each of these options can be accessed through commands in the DSM language set. DSM users can access any VAX/VMS-supported device available for use.

The VAX-11 DSM system provides an interface to VAX/VMS I/O handlers according to device type. Terminal I/O and interprocess communication through mailboxes is handled by the VAX/VMS Queue I/O service, whereas I/O to all other devices is performed through VAX-11 RMS (Record Management Services). This allows DSM users to access RMS sequential, relative, and indexed files, in addition to global variable files, and perform transparent communication through the DECnet software.

Shared Memory Areas

The VAX-11 DSM system supports a high degree of code and data sharing through the use of VAX/VMS virtual memory sections. Mapping a set precompiled DSM routines in a virtual memory section improves the performance of a DSM application because the system does not have to perform I/O to access DSM routines stored on disk. Instead, it can execute the routines directly from virtual memory.

Virtual memory sections can be either private or shared. If shared, they are called global sections. Global sections can be created dynamically by a process or they can be permanently present in the system. Permanent global sections are generally created from routines to which a number of users require access. When a group of routines or an application is installed in a global section, all users share the same copy of precompiled DSM routines. At runtime, a copy of this set of routines is mapped into the virtual address space of each requesting process.

All users can create private virtual memory sections. However, users must have sufficient VAX/VMS operating system privileges to create and install a global section.

DSM Job Controller

The DSM Job Controller is a separate process that manages interlock requests by multiple DSM user processes. It also allows system-wide control over the running of DSM application, providing functions such as enabling and disabling journaling.

Communication between a VAX-11 DSM process and the DSM Job Controller takes place through mailboxes.

The VAX-11 DSM system lets users either use or bypass the DSM Job Controller at DSM image activation. Work that does not affect a common database—typically program development— can bypass the Job Controller. However, when multiple users are running a DSM application, interlocking requires the use of the DSM Job Controller.

Journaling

Journaling is a means of keeping a record on secondary storage (disk or magnetic tape) of transactions that alter the database (i.e., global variable SETs and KILLs). VAX-11 DSM journaling is handled by a separate process communicating with DSM users through mailboxes.

The VAX-11 DSM system provides a number of journaling options to meet the needs of a system running multiple applications. Depending on the options selected, there can be one or more journal processes. One journal process can be run for each group in the system, for a number of groups in the system, or for the entire system.

Each journal process monitors database transactions through mailboxes, which are VAX/VMS pseudo-devices used for interprocess communication. Whenever a DSM user process performs a SET or KILL on a global variable, the journal process makes a record of it in one of many possible journal files. In the event of database degradation, these files can be used to restore the database.

System and Library Utilities
The VAX-11 DSM software package includes a number of utility routines written in the DSM language. These routines help the application programmer to develop and maintain the software and data for his or her application, and allow the system manager to control the running of DSM applications.

The utilities are divided into two categories: library utilities and system utilities. Library utilities perform general services in three categories: procedures affecting routines; procedures affecting globals; and miscellaneous operations such as numeric conversion. System utilities perform services in the areas of: journaling control; job control, and other maintenance operations; and system information.

Generally, the system and library utilities are accessed through a menu-driven utility package. Most utilities in the package are interactive, that is, they prompt for required user input. In addition, most utilities provide extensive online documentation that explains how to use them.

VAX-11 MACRO
The VAX-11 MACRO assembler accepts one or more source modules written in the MACRO assembly language and produces a relocatable object module and symbol table and optional assembly listing. The VAX-11 MACRO language is similar to the PDP-11 MACRO language, but its instruction mnemonics correspond to the VAX native instructions. The VAX-11 MACRO assembly language is characterized by the following:

- Relocatable object modules
- Global symbols for linking separately assembled object programs
- Global arithmetic, global assignment operator, global label operator, and default global declarations
- User-defined macros
- Multiple macro libraries
- Program sectioning directives
- Conditional assembly directives
- Assembly and listing control functions
- Alphabetized, formatted symbol table listing
- Default error listing on command output device
- An optional Cross Reference Table (CREF) symbol listing

Symbols and Symbol Definitions
Three types of symbols can be defined for use within MACRO source programs: permanent symbols, user-defined symbols, and macro

symbols. Permanent symbols consist of the VAX instruction mnemonics and MACRO directives; they do not have to be defined by the user. User-defined symbols are those used as labels or defined by direct assignment. Macro symbols are those symbols used as macro names.

MACRO maintains a symbol table for each type of symbol. The value of a symbol depends on its use in the program. To determine the value of a symbol in the operator field, the assembler searches the macro symbol table, user symbol table, and permanent symbol table, in that order. To determine the value of the symbol used in the operand field, the assembler searches the user symbol table and the permanent symbol table, in that order. These search orders allow redefinition of permanent symbol table entries as user-defined or macro symbols.

User-defined symbols are either internal or external (global) to a source program module. An internal symbol definition is limited to the module in which it appears. Internal symbols are temporary definitions which are resolved by the assembler.

A global symbol can be defined in one source program module and referenced with another. Global symbols are preserved in the object module and are not resolved until the object modules are linked into an executable program. With some exceptions, all user-defined symbols are internal unless explicitly defined as being global.

Directives
A program statement can contain one of three different operators: a macro call, a VAX instruction mnemonic, or an assembler directive. The MACRO assembly language includes directives for:

- Listing control
- Functional specification
- Data storage allocation
- Radix and numeric usage declarations
- Location counter control
- Program termination
- Program sectioning
- Global symbol definition
- Conditional assembly
- Macro definition
- Macro attributes
- Macro message control
- Repeat block definition
- Macro libraries

Listing Control Directives
Several listing control directives are provided in MACRO to control the content, format, and pagination of all listing output generated during assembly. Facilities also exist for titling object modules and presenting other identification information in the listing output.

The listing control options can also be specified at assembly time through command qualifier options included in the listing file specification in the command string issued to the MACRO assembler. The use of these command qualifiers overrides all corresponding listing control directives in the source program.

Conditional Assembly Directives
Conditional assembly directives enable the programmer to include or exclude blocks of source code during the assembly process, based on the evaluation of stated condition tests within the body of the program. This capability allows several variations of a program to be generated from the same source module.

The user can define a conditional assembly block of code, and within that block, issue subconditional directives. Subconditional directives can indicate the conditional or unconditional assembly of an alternate or non-contiguous body of code within the conditional assembly block. Conditional assembly directives can be nested.

Macro Definitions and Repeat Blocks
In assembly language programming, it is often convenient and desirable to generate a recurring coding sequence by invoking a single statement within the program. In order to do this, the desired coding sequence is first established with dummy arguments as a macro definition. Once a macro has been defined, a single statement calling the macro by name with a list of real arguments (replacing the corresponding dummy arguments in the macro definition) generates the desired coding sequence or macro expansion. The MACRO language automatically creates unique symbols where a label is required in an expanded macro to avoid duplicate label specifications. Macros can be nested; that is, the definition of one macro can include a call to another.

An indefinite repeat block is a structure that is similar to a macro definition, except that it has only one dummy argument. At each expansion of the indefinite repeat range, this dummy argument is replaced with successive elements of a specified real argument list. This type of macro definition does not require calling the macro by name, as required in the expansion of conventional macros. An indefinite repeat block can appear within or outside of another macro definition, indefinite repeat block, or repeat block.

Macro Calls and Structured Macro Libraries

A program can call macros that are not defined in that program. A user can create libraries of macro definitions, and MACRO will look up definitions in one or more given library files when the calls are encountered in the program. Each library file contains an index of the macro definitions it contains to enable MACRO to find definitions quickly.

Program Sectioning

The MACRO program sectioning directives are used to declare names for program sections and to establish certain program section attributes. These program section attributes are used when the program is linked into an image.

The program sectioning directive allows the user to exercise complete control over the virtual memory allocation of a program, since any program attributes established through this directive are passed to the linker. For example, if a programmer is writing multi-user programs, the program sections containing only instructions can be declared separately from the sections containing only data. Furthermore, these program sections can be declared as read-only code, qualifying them for use as protected, shareable programs.

HOST DEVELOPMENT LANGUAGES

PDP-11 FORTRAN IV/VAX TO RSX

The FORTRAN IV language is an extended FORTRAN implementation based on American National Statndard (ANSI) FORTRAN, X3.9-1966. PDP-11 FORTRAN IV code is executed in compatibility mode under the VAX/VMS operating system. The FORTRAN IV language includes the following extensions to the ANSI standard:

- Generated expressions allowed in all meaningful contexts
- Mixed-mode arithmetic
- BYTE data type for character manipulation
- ENCODE, DECODE statements
- PRINT, TYPE, and ACCEPT input/output statements
- Direct-access, unformatted input/output DEFINE FILE statement
- Comments allowed at the end of each source line
- PROGRAM statement
- OPEN and CLOSE file access control statements
- List-directed input/output

Additionally, virtual arrays are supported on target systems with memory management directives. Virtual arrays are memory-resident and require enough main memory to contain all elements of all arrays.

The PDP-11 FORTRAN IV compiler is a fast, one-pass compiler. Compiler options allow program size versus execution speed (threaded code versus inline code) trade-offs. FORTRAN IV compiler optimizations include:

- Common subexpression elimination
- Local code tailoring
- Array vectoring
- Optional inline code generation for integer and logical operations

MACRO-11 subroutines may be called from FORTRAN IV programs. The FORTRAN IV language also includes a set of object modules, called the Object Time System (OTS), that are selectively linked with compiler-produced object modules to produce an executable program.

MACRO-11

MACRO-11, the PDP-11 assembly language, is included in the compatibility mode environment. Programs written in the MACRO-11 assembly language can be assembled to produce relocatable object modules and optional assembly listings. The following features are provided:

- Relocatable object modules
- Global symbols for linking separately assembled object programs
- User-defined macros
- A comprehensive system macro library
- Program sectioning directives
- Conditional assembly directives
- Assembly and listing control functions at program and command levels
- Alphabetized, formatted symbol table listing
- Default error listing on command output device

Symbol and Symbol Definitions

Three types of symbols can be defined for use within MACRO-11 source programs: permanent symbols, user-defined symbols, and macro symbols. Accordingly, three types of symbol tables are maintained: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST).

Permanent symbols consist of the PDP-11 instruction mnemonics and MACRO directives. The PST contains all the permanent symbols automatically recognized by MACRO and is part of the assembler itself. Since these symbols are permanent, they do not have to be defined by the user in the source program.

User-defined symbols are those used as labels or defined by direct assignment. Macro symbols are those symbols used as macro names. The UST and MST are constructed during assembly by adding the symbols to the UST or MST as they are encountered.

Directives

A program statement can contain one of three different operators: a macro call, a PDP-11 instruction mnemonic, or an assembler directive. Directives are included for:

- Listing control
- Function specification
- Data storage
- Radix and numeric usage declarations
- Location counter control
- Program termination
- Program boundaries information
- Program sectioning
- Global symbol definition
- Conditional assembly
- Macro definition
- Macro attributes
- Macro message control
- Repeat block definition
- Macro libraries

CHAPTER OVERVIEW

This chapter describes the wide range of capabilities supported by VAX/VMS systems for managing data, forms, records, and databases. It begins with a brief overview of the set of VAX information management products. The rest of the chapter includes detailed descriptions of the individual products.

Topics include:

- The Structure of the Architecture
- Overview of the Products
- VAX-11 DATATRIEVE
- VAX-11 FMS
- VAX-11 Common Data Dictionary
- VAX-11 DBMS

CHAPTER 6
INFORMATION MANAGEMENT

INTRODUCTION

The VAX/VMS operating system supports a set of software tools that provides a full range of information management capabilities. With these tools, data can be organized, maintained, retrieved and manipulated quickly, easily, and cost-effectively. The layered structure, called the VAX information architecture, includes:

- VAX-11 DATATRIEVE data management facility
- VAX-11 FMS (Forms Management System)
- VAX-11 CDD (Common Data Dictionary)
- VAX-11 RMS (Record Management Services)
- VAX-11 DBMS (Database Management System)

The architecture of the VAX information products was developed on the principle that no single approach to information management is appropriate for the typical user's combination of application needs. The layered, modular design of the VAX information architecture makes it possible to apply the appropriate technology to each level of an application. The components are not just a collection of independent point products, but a series of interlocking building blocks that fit into a well-defined software structure.

THE STRUCTURE OF THE ARCHITECTURE

The components of the architecture are arranged in layers above the operating system, as seen in Figure 6-1. Each layer has specific capabilities. The layered structure of the architecture makes it possible for the components on one level to use the facilities of the other components.

The top or outside layer provides a user interface where the architecture supports interactive and language-callable video forms, English-like queries, hardcopy reports, and graphics. At this level, users such as application programmers, data entry clerks, and management personnel can work with data, not as records and files or bits and bytes, but as meaningful information formatted to their specifications.

On the next level is the data dictionary and a facility for high-level access to local and remote data.

The data dictionary provides a facility for storing logical-to-physical data definitions. This facility integrates the other VAX information management products. For example, high-level data access facilities

PRODUCTS OF THE ARCHITECTURE

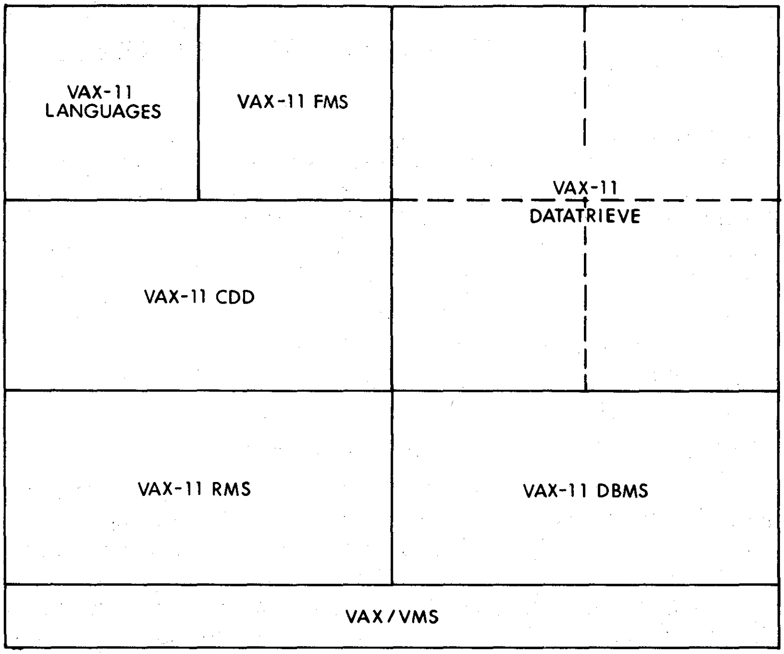


Figure 6-1

use data dictionary information to access locally and remotely stored data. The database management system uses the data dictionary to store the database data definitions it shares with the languages and the high-level data access facility.

The high-level data access facility lets the application program or interactive user state an access request in terms of a desired result rather than having to specify the operations required to achieve that result. For example, the user could request a printout of all employees with an annual salary equal to or greater than their age times a thousand dollars. The high-level access facility performs all data access, selection, formatting, and output operations required to produce the desired report--all in response to one simply stated English-like query request. The English-like syntax is results oriented. Users ask for what they want; the high-level data access facility determines how to locate the data.

High-level data access also supports a relational join capability for dynamically linking related records of different types. Users do not have to determine in advance the records they want to link. Using a relational join, the access facility is capable of making these associations on the fly.

For instance, in the above example, employee stock purchase information might be stored in a separate file from the main employee records. However, stock-purchase information could be included in the initial retrieval by using a request in the form "PRINT EMPLOYEES CROSS STOCK-PURCHASE OVER EMP-NUMBER WHERE ANNUAL SALARY IS EQUAL TO OR GREATER THAN AGE TIMES 1000." In this case, the associated stock purchase information is appended to each printed record because the CROSS operator performs a join of employee master records with employee stock-purchase records.

The distributed data access facility retrieves data from remote VAX-11 DATATRIEVE nodes. The process is transparent. A remote query looks just like a local query to the user.

The lowest level consists of two online, multiuser, data management facilities; one for traditional file structures and one for pointer-based database structures. Sequential, relative, and multikey-ISAM (Indexed Sequential Access Method) file organizations are supported by the file management system. The database management system is CODASYL-compliant and supports the network data model.

The Components of the Architecture

Figure 6-2 maps the VAX information architecture capabilities over the product set presented in Figure 6-1.

Programming Languages — The VAX languages are a basic part of the VAX information architecture. The architecture provides language support for high-level and direct access to RMS (Record Management Services) files and VAX-11 DBMS (Database Management System) data structures. Through the VAX Procedure Calling Standard, languages can use the VAX-11 DATATRIEVE data management facility for high-level access to data. The syntax for calling the VAX-11 DATATRIEVE facility is exactly the same as that for using it interactively.

For VAX-11 DBMS access, a data manipulation language (DML) is provided for VAX-11 FORTRAN and VAX-11 COBOL programs. All other application languages are supported through a callable VAX-11 DBMS utility called DBQ (database query).

VAX-11 FMS — VAX-11 FMS (Forms Management System) is a programmer productivity tool that provides a forms managememt capability for application languages and the VAX-11 DATATRIEVE data management facility. FMS forms are defined interactively and then

FEATURES OF THE ARCHITECTURE ②

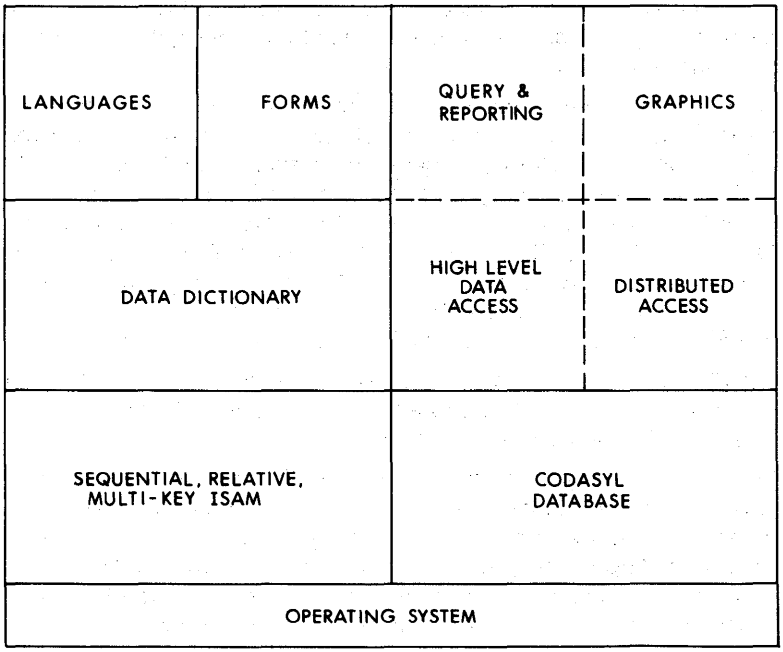


Figure 6-2

stored in an FMS forms library. At runtime, VAX-11 FMS works as a forms management software front end. It passes data between user programs and a video terminal on a per-field or per-form basis.

The process works exactly the same way when FMS forms are used with the VAX-11 DATATRIEVE facility. If a form name is used as part of a DATATRIEVE definition, the VAX-11 DATATRIEVE facility will automatically use the form to collect or display the associated data. From the point of view of VAX-11 FMS, VAX-11 DATATRIEVE is just another user program.

VAX-11 DATATRIEVE — VAX-11 DATATRIEVE is a comprehensive data management tool. It provides both interactive and program-callable access to data in RMS file organizations or in more complex, interrelated database structures. It is a comprehensive query and report writer with full update capabilities. It also includes an integrated graphics capability. Forms support is provided through VAX-11 FMS (Forms Management System).

The VAX-11 DATATRIEVE facility consists of four major subcomponents: at the user interface level are a query and report writing facility and a business graphics capability; below that, the local and distributed high-level data access facilities.

The VAX-11 COMMON DATA DICTIONARY — The VAX-11 CDD is in many respects the keystone of the architecture and is essential to the operation of VAX-11 DATATRIEVE and VAX-11 DBMS. VAX-11 DATATRIEVE statements refer to data entities defined in the VAX-11 CDD. The VAX-11 CDD is also used to store series of VAX-11 DATATRIEVE statements as procedures that can be invoked interactively or from application programs.

The VAX-11 CDD is used to store the database data definitions (schemas and subschemas) VAX-11 DBMS needs to create, access, and maintain databases. Application languages access these definitions at compile time.

VAX-11 RMS — VAX-11 RMS (Record Management Services) is an access method with an extended syntax interface to all high-level languages. It supports sequential, relative, and multikey indexed-sequential file organizations, as well as concurrent file access with record-level locking.

VAX-11 DBMS — The VAX-11 DBMS facility is a full-scale CODASYL-compliant database management system based on the March 1981 Working Document of the ANSI Data Definition Language Committee. It has many special ease-of-use and performance features and is suitable for both small and large database applications. Because it uses the powerful network-type data model, it can accommodate complex data relationships. The VAX information architecture allows for DBMS data to be accessed directly from applications languages or through the VAX-11 DATATRIEVE facility. Included with the VAX-11 DBMS facility is an important productivity tool, DBQ, an interactive and program-callable database query language, that makes it easy to write and check out VAX-11 DBMS data access statements.

The remainder of this chapter is divided into five sections that cover in detail the features and functions of VAX-11 DATATRIEVE, VAX-11 FMS, the VAX-11 CDD, VAX-11 RMS, and VAX-11 DBMS. The Language/VAX information architecture interface is integrated into each of the information management components. The use of DECnet-VAX communications software is covered under Distributed Data Access in the VAX-11 DATATRIEVE section.

VAX-11 DATATRIEVE

The VAX-11 DATATRIEVE facility is a multi-faceted data management facility that can store, update, and retrieve information and generate reports. The major commands include:

- CROSS—which allows multiple files to be accessed using a common field
- DECLARE—which defines global and local variables to be used within a DATATRIEVE query
- DEFINE—which provides a consistent mechanism for creating domain, record, table, and view definitions in the VAX-11 Common Data Dictionary
- DROP—which allows records to be deleted from a collection (subset) only, while not modifying the actual data file
- EDIT—which invokes an editor that inserts, modifies, or deletes text from procedures defined in the VAX-11 CDD, or from the last line entered in an interactive session
- ERASE—which deletes one or more records corresponding to the appropriate domain (file)
- FIND—which establishes a collection (subset) of records contained in either a domain or a previously established collection based on a Boolean expression
- FOR—which executes a subsequent command once for each record in record collection providing a simple looping facility
- HELP—which provides a summary of each DATATRIEVE command
- MODIFY—which alters the values of one or more fields for either the selected record or all records in a collection. Replacement values are prompted for by name, or shown on a pre-defined form
- PLOT—which allows a collection of records to be displayed/printed in graphic representation
- PRINT—which prints one or more fields of one or more records. Output can be optionally directed to a lineprinter or disk file. Format control can be specified. A column header is generated automatically
- READY—which identifies a domain for processing and controls the access mode to the appropriate file
- SELECT—which identifies a single record in a collection for subsequent individual processing
- SORT—which reorders a collection of records in either the ascending or descending sequence of the contents of one or more fields in the records

- STORE—which creates a new record. The value for each field contained in the record is prompted for by name, or indicated on a predefined form

VAX-11 DATATRIEVE also provides a subset capability to allw novice users to learn about DATATRIEVE while using it productively. This facility is called "Guide" mode. It provides explicit help at all decision points in processing a DATATRIEVE subset. The "Guide" mode subset includes:

SHOW
READY
FIND
PRINT
SORT
SELECT

VAX-11 DATATRIEVE can be used interactively from a terminal or called from an application program. Data can be accessed in VAX-11 RMS files and VAX-11 DBMS database structures. VAX-11 DATATRIEVE features integrated editing and graphic output facilities and it supports the forms management facility of VAX-11 FMS.

The VAX-11 DATATRIEVE facility also provides a distributed data access capability using DECnet-VAX communications software. This capability makes it possible to use VAX-11 DATATRIEVE to retrieve data on remote VAX systems, just as if the data were stored locally. A single DATATRIEVE command is capable of accessing data from RMS or DBMS files local or remote simply depending on its definition in the VAX-11 CDD.

Designed to be used by both novices and computer professionals, VAX-11 DATATRIEVE operates effectively in commercial, technical, scientific, industrial, or educational environments. Typical applications range all the way from producing a complex report to answering a casual question. For example, using VAX-11 DATATRIEVE, a personnel file could be queried to determine how many employees held bachelor's degrees, or the same data file could be used to produce a report with a complete statistical analysis of the employee education versus compensation.

Another typical environment where VAX-11 DATATRIEVE would be useful is a distributorship with an order processing system. In this setting, sales data could be extracted by territory, then the results could be plotted in the form of a pie chart. Order backlogs might be retrieved, sorted by supplier, and plotted in the form of a bar graph.

Implementing a VAX-11 DATATRIEVE application is a two-phase process. In the first phase, the appropriate statements are used to define all data that will be accessed by the application. This need be done only once to establish a foundation on which to build the application. In the second phase, VAX-11 DATATRIEVE statements are used to process the data associated with these definitions.

Data Definition
The data definition process involves establishing special VAX-11 DATATRIEVE constructs called *domains*.

Domains — The domain concept is central to DATATRIEVE. Domains represent relationships between actual physical data and descriptions of data. VAX-11 DATATRIEVE performs all data management in terms of domains. Domains must be defined before DATATRIEVE can manage the data associated with them.

In the simplest form, a VAX-11 DATATRIEVE domain definition consists of a domain name, the name of the VAX-11 RMS file, and the name of a record format description. A record format description defines the fields within a record, assigning each field a name and specifying its length, data type, and other vital parameters. All VAX-11 DATATRIEVE domain definitions and record format descriptions are contained in the VAX-11 Common Data Dictionary.

Record format descriptions can specify data validation criteria on a per-field basis. VAX-11 DATATRIEVE automatically uses the validation parameters to screen data at the time of input so that only data defined as valid is accepted. Supported validation parameters include range checks, missing value checks, default values, must-match tables, and argument-function conversion tables.

Domains can span multiple VAX-11 RMS files or VAX-11 DBMS record types. Domains can also include the name of an associated VAX-11 FMS form or a VAX-11 DATATRIEVE table. Domains can also be defined as remote. This means that the actual data definition and the data exist on a remote VAX-11 DATATRIEVE node and can be accessed through DECnet-VAX communication software using the distributed data access facility. These more complex domains are explained in more detail below.

Data Management
Data management involves creating and maintaining data in a current and correct state by adding, eliminating, and modifying records. The STORE, ERASE, and MODIFY statements are used to perform these relatively straightforward functions.

Populating Files — When an application requires the creation of new files, the new files must be filled with data. This process is called "populating" the file. A series of successive STORE statements is used for this purpose. With the STORE statement, VAX-11 DATATRIEVE prompts the user for each field value and, before accepting input, performs any validation checks specified by the record format description.

Data Retrieval

Maintaining an accurate database, however, is not an end in itself. Data is used to make decisions, generate reports, initiate transactions, and generally facilitate the operational processes of an enterprise. VAX-11 DATATRIEVE allows stored data to be retrieved in an easily understood form regardless of underlying data structure (RMS or DBMS) or location (local or remote via DECnet).

The data retrieval statements of VAX-11 DATATRIEVE are simple, and particularly powerful statements with English-like syntax. They consist of verbs modified by a Record Selection Expression (RSE). The RSE defines a subset of the records in the domain. These records are then selected by VAX-11 DATATRIEVE for retrieval. One statement can get the answer to a casual query or produce a long detailed report.

"EMPLOYEES WITH SALARY GREATER THAN 20000," " ACCOUNTS WITH UNPAID-BALANCE GREATER THAN 600," or "DONORS WITH BLOODTYPE EQUAL O-NEG" are examples of typical RSEs. Multiple conditions can be combined in a single RSE--for example, "DONORS WITH BLOODTYPE EQUAL O-NEG AND LAST-DONATION-DATE LESS THAN "4/30/81."" The VAX-11 DATATRIEVE SORT operator can be appended to the RSE to order the records being retrieved.

Ad hoc information retrieval with VAX-11 DATATRIEVE is normally performed as an iterative process using a series of statements to progressively narrow down the group of records to be retrieved. This works by using a FIND request with a specified domain as its object to establish what is called the current collection. Subsequent FIND requests progressively narrow down the current collection until the user is satisfied with the results. For example, the statement "FIND DONORS WITH BLOODTYPE EQUAL O-NEG AND LAST-DONATION-DATE LESS THAN "30/4/81"" might yield the VAX-11 DATATRIEVE response "200 RECORDS FOUND." In this case, the user could narrow down the current collection with the statement "FIND CURRENT WITH ZIP-CODE EQUAL 23016." VAX-11 DATATRIEVE might then respond with "16 RECORDS FOUND" and the user could PRINT these records to get telephone numbers for soliciting blood donations to help an accident victim.

Reports
The PRINT statement is used to output information to a display terminal, a printer, or a VAX-11 RMS file. Though there are some formatting options possible with the PRINT statement, they are limited. The REPORT command provides a more comprehensive set of formatting options for producing standard printed reports with page and column headings, page numbers, totals, and subtotals.

Graphics
The VAX-11 DATATRIEVE graphics capability includes histograms, bar charts, pie charts, xy scatter diagrams, and time series graphs. Plots use the VT125 video terminal for display and can be printed using an attached printer. The syntax for the plot statement consists of the PLOT verb followed by plot type and the fields to be plotted.

Forms
VAX-11 DATATRIEVE can be used with VAX-11 FMS to provide forms input and output. See "Using FMS With VAX-11 DATATRIEVE.

Stored Procedures
With the DEFINE PROCEDURE command, users can define sequences of VAX-11 DATATRIEVE commands and statements and store them for later use. PROCEDURES can be invoked to run by themselves or can be embedded in other sequences of commands and statements. PROCEDURES can be invoked by interactive users or application programs.

Ease-Of-Use Features
Guide Mode — VAX-11 DATATRIEVE provides a self-teaching facility, called "Guide" mode. In this mode of operation, users are guided through their VAX-11 DATATRIEVE sessions with a series of prompts. This enables the user to work productively with DATATRIEVE while learning to use it.

To invoke guide mode, the user issues a SET GUIDE command. VAX-11 DATATRIEVE immediately responds with "ENTER COMMAND, TYPE ? FOR HELP." If "?" is typed at this point, VAX-11 DATATRIEVE will present the user with the possible responses--in this case, READY, SHOW, or LEAVE. If one of the alternatives is selected, VAX-11 DATATRIEVE then procedes to guide the user through the syntax of the selected statement. In the case of READY, VAX-11 DATATRIEVE prompts with "DOMAIN NAME, END WITH SPACE."

VAX-11 DATATRIEVE Editor — The VAX-11 DATATRIEVE editor closely resembles the standard VAX editor, EDT. It can be used in either the line or character mode, with or without keypad commands.

The editor lets the user correct typing or syntax errors in VAX-11 DATATRIEVE statements without having to completely retype the statements.

To get into editor mode, the user types EDIT and a carriage return. VAX-11 DATATRIEVE places the last command or statement in the main buffer of the editor, and the user edits this just like any other text. If EXIT is used to leave the editor, VAX-11 DATATRIEVE performs the edited statement or command. If QUIT is used to leave the editor, VAX-11 DATATRIEVE ignores the last statement.

Application Design Tool — The Application Design Tool (ADT) is a VAX-11 DATATRIEVE utility that simplifies the process of defining domains, record formats, and creating VAX-11 RMS files. Operating in an interactive mode, ADT presents the user with a series of simple questions. The user's responses provide ADT with information to generate the proper definitions. For RMS files, ADT will prompt the user to get a full set of parameters pertaining to organization, index keys, etc. ADT will then create a VAX/VMS indirect command file that the user can execute immediately or at some later time to create the desired file.

Advanced Features

View Domains — VAX-11 DATATRIEVE allows domains to be defined that can subset the fields of a record and can span multiple VAX-11 RMS files or VAX-11 DBMS record types. These are called view domains because they provide a user's logical view of the data. Once view domains have been established, they can be used in much the same way as simple domains.

This facility is basic to high-level data access. It makes it possible for a single statement to retrieve a set of related records. For example, in an employee records application there might be an employee master file with company confidential information pertaining to salary that could be masked out using a view domain. Other information in the master file such as addresses and telephone numbers could then be combined in another view domain with a special file of records used in a car-pooling application.

View domains can also be used with VAX-11 RMS files for domains containing records related in a hierarchical fashion. For example, in an order processing application there might be an account master file and an order file. These files could be combined in a view to produce billing statements with data drawn from both files. A single record in this view domain could be defined to contain one account master record and all the orders applying to that account.

Joining Multiple Records Using CROSS — VAX-11 DATATRIEVE also has a similar and equally important relational facility for linking multiple record types dynamically. Using the CROSS operator, records from separate VAX-11 RMS files or from different VAX-11 DBMS data structures can be joined dynamically in a single retrieval statement.

This is an especially powerful facility that makes it possible for users to join records from any files related to one another by a common field. For example, in a student records application, a school might maintain a current academic status file and a registration file. When mailing grade scores the academic status file could be joined with the registration file to get the home address. The VAX-11 DATATRIEVE statement would take the form "CROSS ACADEMIC-STATUS WITH REGISTRATION OVER STUDENT-ID."

DBMS Domains — DBMS domains are a VAX-11 DATATRIEVE feature to take advantage of the record format descriptions and interrecord relationships defined in VAX-11 DBMS subschemas. When DBMS domains are used, a schema, subschema and a record type are simply identified. VAX-11 DATATRIEVE uses the record format description and relationships defined for the record type in the VAX-11 CDD.

Tables — VAX-11 DATATRIEVE tables can be defined to reside in the VAX-11 Common Data Dictionary or exist as DATATRIEVE domains. Tables can be used as a must-match list for field validation or for argument-function type conversions. For instance, a must-match list of valid U.S. Mail state abbreviations could be used to check an address field or a argument-function table could be used to convert from state abbreviation codes to the spelled out state name.

Calling VAX-11 DATATRIEVE from Applications Languages — All the functions of VAX-11 DATATRIEVE with the exception of the editor, ADT, and guide mode can be called from all application languages using the standard VAX/VMS call interface. Using the specialized power of FORTRAN, for instance, complex computational operations can be performed on records retrieved by VAX-11 DATATRIEVE. With COBOL, specially formatted reports can be generated from VAX-11 DATATRIEVE collections.

Distributed Data Access — The VAX information architecture works with DECnet-VAX communications software to provide a distributed data access facility that makes it possible for users to access remote data just as if it were stored locally.

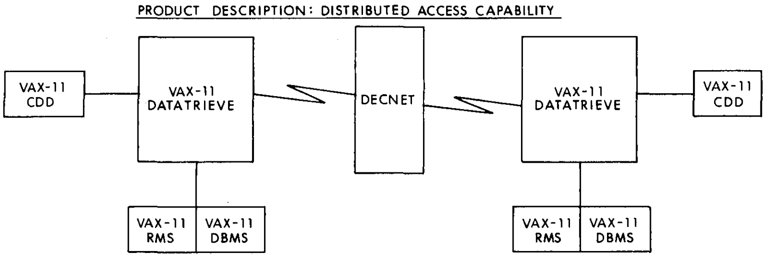


Figure 6-3

Figure 6-3 illustrates the distributed data access process. A remote domain is defined with a simple statement that identifies the host node and domain name. When the query request is executed, VAX-11 DATATRIEVE uses DECnet-VAX software to forward the request to the appropriate node. The response to the request is then returned over the line and presented to the user just as if the data had been stored locally.

Since VAX-11 DATATRIEVE uses the record stream concept, its distributed data access facility is extremely efficient. Line utilization is optimized because only the records that satisfy a query are returned over the transmission line. And since the data description is maintained with the data, the complex problems normally associated with distributed data management are minimized.

VAX-11 FMS

VAX-11 FMS (Forms Management System) provides video form support for applications on VT100, VT125, and VT52 video terminals. Because the VAX-11 FMS facility is integrated into the structure of the VAX information architecture, it can serve both application programs and the VAX-11 DATATRIEVE data management facility. This improves productivity by reducing required training time, since there is only one forms package to learn.

The many special features of the VAX-11 FMS facility improve the productivity of application programmers and application maintainers. It is easy to use and self-teaching, meaning that it can be used effectively by entry-level programmers. However, FMS also has the flexibility and power required by experienced systems designers when implementing complex forms-oriented applications. In addition, FMS provides the means for easily building applications that exploit the full power of the VT100 and VT52 video terminals.

FMS Subcomponents

As shown in Figure 6-4, FMS consists of forms definition software, runtime forms management software, and form library data structures that contain the internal representation of all forms definitions.

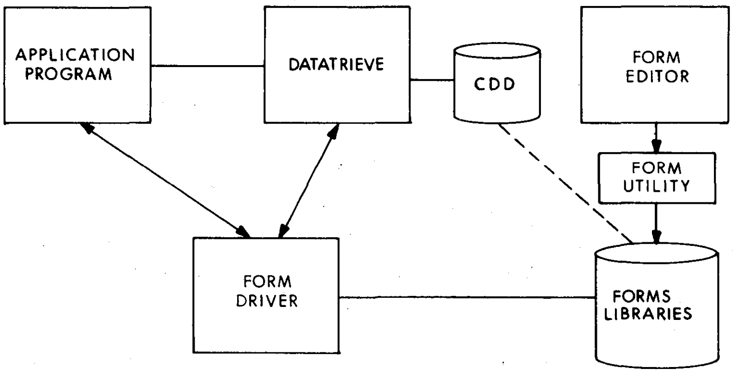


Figure 6-4

Forms definition software performs the functions associated with creating forms and managing the forms libraries. It consists of an interactive editor for forms definition and modification and a library manager.

Runtime forms management software performs interactive forms management functions for application programs and the VAX-11 DATATRIEVE data management facility.

The FMS Form Editor

The interactive FMS Form Editor is quick, easy, and natural. The forms designer does not have to learn a forms definition language or lay out forms on a paper grid. Instead, forms are constructed directly on the VT100 video terminal, and all screen and field attributes are defined with function keys or by filling in simple questionaires. The form appears just as it looks to the terminal operator at runtime. Since forms developers can watch the end product evolve, a single session is all that is required to get a form right, regardless of complexity. The independence of forms from programs allows the forms to be approved early in the development process by the end user, eliminating changes late in the development cycle.

Interactive forms definition is more efficient than a forms-language-oriented process that requires form definition source code to be written and then compiled in an iterative procedure. With a forms language, forms have to be compiled and tested before a forms developer can actually see the results of the process.

Function-Key Logic Using the Keypad
The keypad provides function-key logic for cut-and-paste editing. The user defines a rectangular area of the form by indicating a pair of opposite corners with the cursor. This piece of the form can then be picked up and moved in one operation.

Function keys allow deletion of the entire current line or only the portion to the right or left of the cursor. An undelete function key inserts the most recently deleted line, allowing for error recovery from accidental deletions or fast replication of a single line.

Character attributes (bold, reverse video, underline, and blink) can be defined for any desired rectangular portion of the screen, using a similar technique.

Attributes are assigned to individual fields, and to the form as a whole, through an interactive process in which the FMS forms compiler uses the FMS runtime system to present the operator with a series of simple fill-in-the-blank forms.

The Form Editor creates an internal representation of the form in a work file. The Form utility is then used to create and manage the libraries in which the form definitions reside until they are requested by an application program. The Form utility can also be used to list the names of forms in a library to a generate printable description of a form suitable for use in end-user or system documentation, or to generate COBOL Data Division code reflecting the content of the form.

Forms are not compiled or linked with the application program; the association between the application program and the form is made at runtime. This scheme provides for a high degree of data/program independence, with consequent savings in application maintenance costs.

Using FMS With VAX-11 DATATRIEVE
When the VAX-11 FMS facility is used with application programs, the developer defines a form, then writes a program to use it. In the case of FMS used with the VAX-11 DATATRIEVE data management facility, the developer defines the form and names it as part of a VAX-11 DATATRIEVE data definition. VAX-11 DATATRIEVE does the rest. It will automatically generate the proper FMS calls when data associated with a form definition is input or retrieved.

Runtime Forms Management
The FMS Forms Driver (runtime system) operates as a program-callable software front end to facilitate application-program/terminal-user interaction. This approach is appropriate for interactive source data entry and transaction processing applications where the terminal user is familiar with the data and can make decisions during the input process.

Application-program calls to FMS specify forms and fields by using strings of plain ASCII characters. This simple interface design eliminates the need for any pre-runtime binding through a compilation or linking process. This type of architecture encourages data/program independence and, as a result, improves programmer productivity by making applications easier to develop and maintain. Fields can be moved; some attributes can be changed; and, in some cases, even the order of fields can be changed without requiring a recompilation of the associated application programs.

The FMS forms driver is powerful and flexible. It provides a broad range of calls--from the simple and straightforward to the complex and sophisticated.

Programmers of any level of proficiency can write applications easily. An entry-level programmer could implement an entire application using just two main calls: one to display a designated form and the other to let the operator enter data onto the form and return input data to the program upon completion.

A more experienced programmer could use a much greater repertoire of calls to write an application. Certain calls, for instance, provide for program/operator interaction on a per-field basis. The application program can look up and display an item description, unit price, or quantity on hand, at the moment a part number is entered. Other calls make it possible for the programmer to use scrolling, multiple overlapped forms, and function-key input, to achieve special effects.

Another feature of VAX-11 FMS is its Named Data capability. By storing abitrary strings of data as part of the form definition, to be retrieved by the application program at runtime, the application programmer can create highly general, parameterized, easily maintainable applications. Examples of the types of parameters that can be stored in this manner include names of successor forms for a chained or menu-driven application, names of data files, boundaries for range check logic in the application, small tables of validation data, etc. An installation could easily write a library of subroutines extending FMS validation capabilities and drive them with Named Data.

THE VAX-11 COMMON DATA DICTIONARY

The VAX-11 Common Data Dictionary (CDD) is a central repository for data about data. It is the hub of the VAX information architecture. It ties the components together by making it possible for them to use a single set of data descriptions as a common resource.

The VAX-11 DATATRIEVE data management facility uses the VAX-11 CDD for descriptions of data stored in VAX-11 RMS files or VAX-11 DBMS database structures. VAX-11 DBMS (Database Management System) uses the VAX-11 CDD to store its schemas, subschemas, and storage schemas. Application languages and VAX-11 DATATRIEVE can share subschema definitions stored in it.

Figure 6-5 is a schematic representation of the VAX-11 CDD showing the categories of data it can contain.

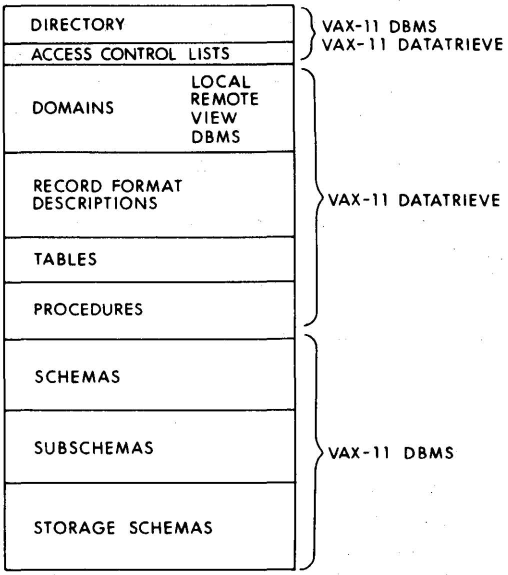


Figure 6-5

The VAX-11 CDD Directory

The VAX-11 CDD has one integrated directory that is an index to both the VAX-11 DATATRIEVE and VAX-11 DBMS data definitions it contains. The directory is organized as an n-level hierarchy that has a structure closely resembling that of the VAX/VMS system directory.

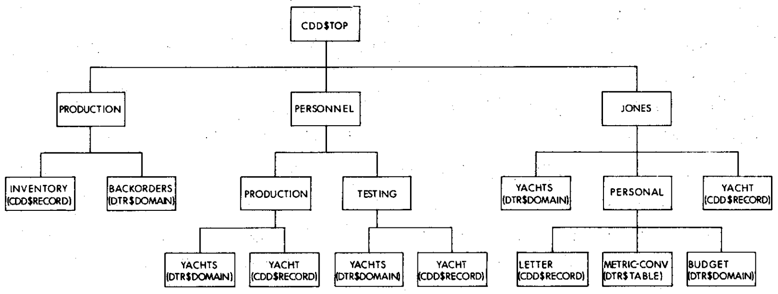


Figure 6-6

Figure 6-6 shows how a typical VAX-11 CDD directory might look. The terminal nodes are called leaves. They always reference VAX-11 CDD objects. The other nodes form a hierarchical access path structure that provides security control through the use of access control lists. There is one access control list per path node. Each entry in the list specifies a user or class of users and their access privileges with respect to dictionary objects below the associated path node.

VAX-11 DATATRIEVE Data Definitions
For VAX-11 DATATRIEVE, the VAX-11 CDD stores record format descriptions and domain definitions for VAX-11 RMS files and VAX-11 DBMS database structures. The VAX-11 CDD also contains VAX-11 DATATRIEVE procedures and tables.

VAX-11 DBMS Data Definitions
For VAX-11 DBMS, the VAX-11 CDD stores database data descriptions of three types—schema, subschema, and storage schema. These definitions are used by VAX-11 DBMS to build and later reference database structures.

The schema is the master data definition for a logical database. It contains all record, data item, and interrecord (set) relationship definitions. There is one schema per database. The subschema definitions are application program views of the data of which there can be many for one database. They are used by the language compilers to produce data definition source code during the compilation process. The storage schema is the master data definition for the physical structures of a database. There is one storage schema per database.

Common Data Definition Language
The Common Data Definition Language (CDDL) utility provides a generic facility to enter, modify, and display record definitions for the VAX

languages. Once a record definition has been entered in the Common Data Dictionary, it can be used by the language compilers or VAX-11 DATATRIEVE. This means only one definition per record need be stored in the VAX-11 CDD.

The Dictionary Management Utility

The Dictionary Management Utility (DMU) is a utility for managing the VAX-11 CDD. The DMU can be used to backup and restore the VAX-11 CDD, to create and delete VAX-11 CDD objects, to create and delete directory nodes on any level, and to create and delete access control lists.

The DMU can also be used to display all or part of the directory structure. It can also display selected information about a particular node.

The VAX-11 CDD can maintain a history of activity and will display on command the access history of specified nodes.

VAX-11 RMS

The VAX-11 RMS facility is the standard DIGITAL data management services software that provides an interface at the application-program level to record/file management functions. VAX-11 RMS provides capabilities to facilitate the definition, creation, population, access, and general maintenance and management of files and records within files. It supports sequential, relative, and multikey indexed-sequential file organizations.

For information on VAX-11 RMS file organization, record access modes, file and record attributes, and utilities, see Chapter 12 of this handbook.

VAX-11 DBMS

Introduction

VAX-11 DBMS (Database Management System) is a CODASYL-compliant general purpose database management system based on the March 1981 Working Document of the ANSI Data Definition Language Committee.

VAX-11 DBMS provides multiuser support with data security and performance features that are required for large-scale database applications. However, VAX-11 DBMS also has ease-of-use features that make it equally suitable for implementing small and medium-scale applications.

There are three stages to the process of implementing a database application using VAX-11 DBMS: data definition, application development, and database creation.

The following step-by-step account of the process of implementing a database application provides an overview of VAX-11 DBMS at work. It identifies the major VAX-11 DBMS subcomponents and describes what they do and how they work together as a system.

Data Definition

VAX-11 DBMS provides three levels of data definition languages (DDLs):

- The schema DDL defines the logical structure of the database
- The storage schema DDL defines the physical structure of the database
- The subschema DDL defines an application program view of a schema

The schema DDL is the only DDL that must be written. The DDL compiler will produce a default subschema and a default storage schema for each compiled schema.

The schema DDL defines the records, sets, and areas composing the database. A record is a collection of data items. A set is a relationship between records having one owner record and one or more member records in some specified order. An area is a logical subdivision of the database that contains records.

The storage schema DDL defines the representation of storage records, storage sets, and storage areas (which are equivalent to VMS files). It also defines the placement of records within the database, the storage set parameters, and the representation of data items within a storage record.

The subschema DDL defines a logical subset of the database in terms of records, sets, and realms (a collection of one or more areas). Many subschemas can be written to provide different views of the database for different application programs.

As shown in Figure 6-7, one DDL compiler compiles all DDL source code and stores schema, subschema and storage schema information in the CDD.

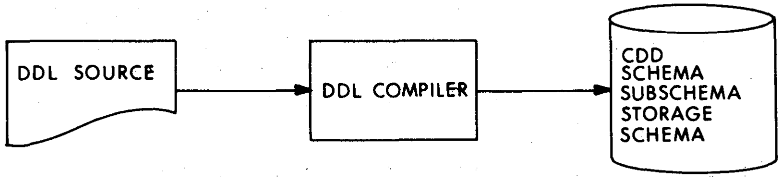


Figure 6-7

Application Development
The application development stage is illustrated by Figure 6-8. Application program source code is compiled by a language processor. The source code must reference a previously defined subschema. The language processor gets the subschema definition from the CDD in coded form and uses it to create DML (Data Manipulation Language) application program data definitions and calls to the Database Control System (DBCS).

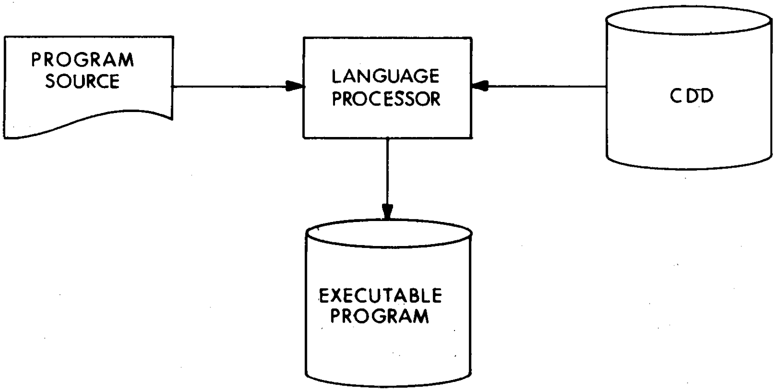


Figure 6-8

Application programs must call the DBCS for all database record and set operations. The specific operations supported are:

- COMMIT—which establishes a run unit quiet point
- CONNECT—which inserts a record into selected sets
- DISCONNECT—which removes a record from selected sets
- ERASE—which deletes a record from the database
- FETCH—which combines FIND and GET
- FIND—which establishes current of run unit
- FREE—which removes a dbkey value from a keeplist
- GET—which gets contents of current record
- KEEP—which inserts a dbkey value into a keeplist
- MODIFY—which changes the contents of a record
- READY—which prepares selected realms for use
- RECONNECT—which combines DISCONNECT and CONNECT
- ROLLBACK—which nullifies changes since last quiet point
- STORE—which puts a record into the database

Direct Language Access to the VAX-11 DBMS Database
Each application program accesses a subdivision of the database through a simple set of commands that acts as an extension to COBOL and FORTRAN programs or as a call from BASIC and MACRO programs. The database can be accessed by application programs directed by the subschema (that has been predefined by the database administrator). The subschema, which is first NAMED in the program, includes record descriptions. The programmer can, therefore, logically manipulate the information in the database using one of two methods provided by VAX-11 DBMS:

- **Data Manipulation Language** — a set of high-level statements that create syntactical and logical extensions to FORTRAN and COBOL. General types of DML statements are Control Statements (READY,COMMIT, ROLLBACK); Retrieval Statements (GET, FETCH, FIND); and Modification Statements (CONNECT, DISCONNECT, MODIFY, RECONNECT,STORE).
- **Call Statement Interface** — used for any VAX-11 language that supports the VAX/VMS calling standard. Programs written in these languages call DBQ to access a database. The same set of statements are available as in the host language DMLs. VAX-11 DBMS provides a User Work Area (UWA) generator to facilitate program development.

Database Creation Using the Database Operator Utility (DBO)
The database creation stage is illustated by Figure 6-9. The Database Operator Utility (DBO) converts coded schema, subschema, and storage schema information from the VAX-11 CDD into data files and control information for the Database Control System (DBCS) to use at runtime.

The result of the database creation stage is called a null database. The null database does not contain data. The process of filling a database with data is called database population. Database population is done with user written programs that are usually modified versions of regular application programs for adding data records in an operational context.

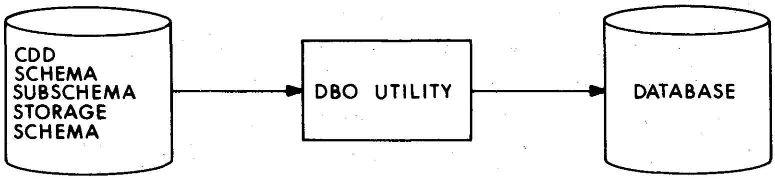


Figure 6-9

Runtime Operation with the Database Control System and the Database Monitor

Figure 6-10 illustrates the runtime operation of VAX-11 DBMS. It shows how multiple applications can access the same database and multiple databases can be supported online at one time using the Database Control System (DBCS) and the database monitor. DBCS is a reentrant shared system program that performs all database accesses. It uses the Database Monitor to control all system lock conflicts and perform journaling and recovery functions.

Though not shown here for the sake of simplicity, database development and database runtime operations can function concurrently.

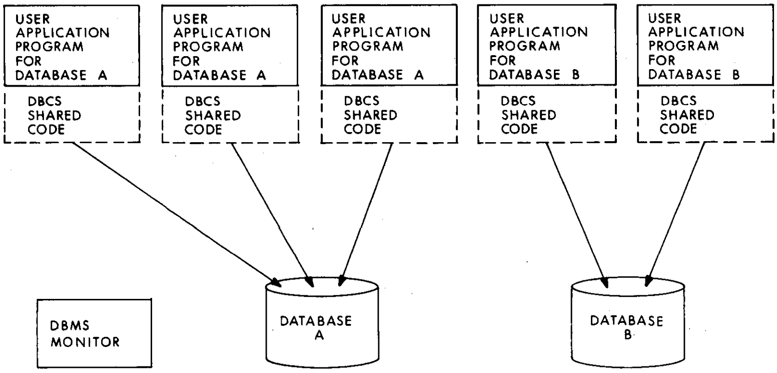

USER APPLICATION PROGRAM FOR DATABASE A
DBCS SHARED CODE
USER APPLICATION PROGRAM FOR DATABASE A
DBCS SHARED CODE
USER APPLICATION PROGRAM FOR DATABASE A
DBCS SHARED CODE
USER APPLICATION PROGRAM FOR DATABASE B
DBCS SHARED CODE
USER APPLICATION PROGRAM FOR DATABASE B
DBCS SHARED CODE
DBMS MONITOR
DATABASE A
DATABASE B

Figure 6-10

EASE-OF-USE FEATURES

Default Subschemas and Storage Schemas

Many applications do not warrant a full-scale design effort involving subschemas and an optimized storage schema. To accommodate these situations, VAX-11 DBMS provides an automatic default subschema and storage schema generating facilility. When a schema is compiled by itself, the DDL compiler creates default subschema and storage schema information in the VAX-11 CDD. This information can then be extracted from the Common Data Dictionary in source DDL form and subsequently edited with the standard VAX/VMS editor (EDT) to achieve a level of customization appropriate to specific applications.

Interactive Data Manipulation with DBQ

VAX-11 DBMS includes an interactive data manipulation tool called

DBQ. DBQ lets the user interactively retrieve, update, and store any database record. It executes COBOL-like data manipulation commands and automatically generates VT100 displays of easy-to-follow schematic diagrams that illustrate access paths. Using DBQ, data manipulation operations can be tested against actual data structures. This is particularly useful when checking out application program designs.

Integrated Database Administration with DBO

The Database Operator utility provides the Database Administrator with all of the functions required to create, maintain, delete, and control VAX-11 DBMS databases. It provides the following:

- Creation, initialization, and deletion of databases
- Reports on VAX-11 DBMS information in the VAX-11 CDD
- Extraction of DDL source from the VAX-11 Common Data Dictionary
- Deletion of DDL information in the VAX-11 CDD
- Online verification of the integrity of internal database structures
- Modification of the contents of corrupted database storage areas (This function is not recommended nor required for normal usage)
- Formatted database dumps
- Offline full and incremental database backup
- Database restoration from backup and long-term journals
- Control and display of the status of the DBCS Monitor process
- Database access statistics
- Generation of a User Work Area (UWA) for application programs using the high-level call interface

Simple Restructuring without Reload

Fields, records, and new set relationships (provided they do not require retrofit database modifications) can be added without having to unload and reload the database. This feature is especially useful for applications that tend to grow over time.

Database Verification With DBO/VERIFY

VAX-11 DBMS has a database verification utility called DBO/VERIFY. This utility can be used to check the integrity of a database that a user suspects might be corrupted. It checks for valid set linkages and database page formats.

ADVANCED FEATURES

Multiuser Support with Concurrency Control
VAX-11 DBMS provides full concurrent access and update capabilities with automatic record-level locking. The application developer does not have to be concerned with multiuser contention for data records by declaring and releasing data locks, because this is performed automatically in a totally transparent, efficient manner. In addition, users will always see a consistent view of the database.

Transaction Backout
VAX-11 DBMS performs record journaling with automatic transaction rollback. A transaction is a logical unit of work in an application program bounded by program quiet points. A program quiet point occurs when a program is first activated or when a COMMIT or ROLLBACK command is executed. If a process aborts or issues a ROLLBACK command, all updates not yet committed will be backed out automatically.

Journaling/Recovery
VAX-11 DBMS has the facility for record-level journaling that keeps long-term after-image journals of all database updates and before-images of all uncommitted updates. These journal records are used to recover from program, system, or hardware failures.

After a program or system malfunction, VAX-11 DBMS will rollback all uncommitted transactions. If data has been destroyed, VAX-11 DBMS can roll forward from a backup copy of the database, using after-images in the journal to reapply all committed transactions.

Multiple Databases
VAX-11 DBMS allows multiple databases to be online at the same time. This is useful when totally independent data must be maintained in separate databases with different schemas. It is also useful in the more commonly encountered situation of a single schema that applies to both a test and production database. A single VMS process can only access one database at a time.

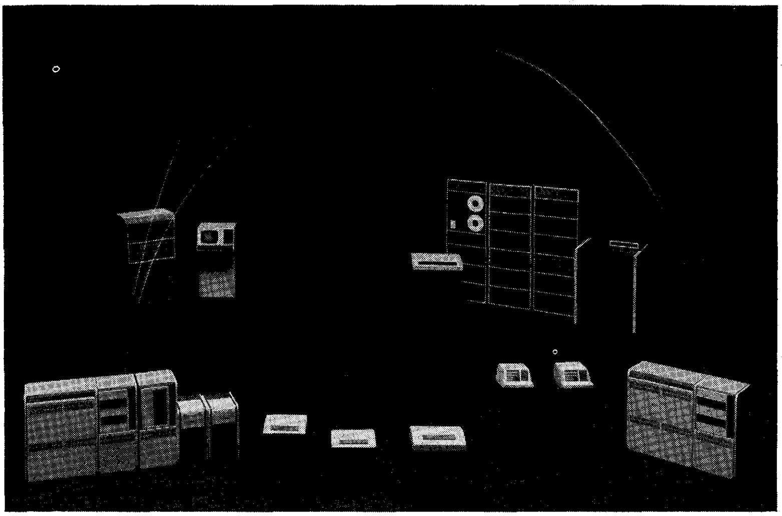
Performance Optimization
VAX-11 DBMS uses its own optimized access method to take advantage of the VAX/VMS architecture. Many other design features have been included to improve performance.

Journaling at the record level greatly reduces the amount of data that must be written to the journal. Data buffer caching and I/O transfer clustering are also performed to increase the efficiency of the system.

An indexed tree structure is used for sorted sets which greatly reduces the overhead of access to sorted sets.

Data Security

VAX-11 DBMS uses standard VMS system-owner-group-world file security logic at the storage-area level. Data is further protected by the standard CODASYL mechanism of subschemas. A subschema defines down to the data item just what subset of the total database a program can access. The association between subschemas and programs must be controlled by the Database Administrator. Data security is also provided by protecting data definitions in the VAX-11 CDD with access-control lists at each node of the hierarchical directory.



CHAPTER OVERVIEW

DIGITAL offers a range of products to link tasks or processes together, whether they are running on the same or different systems. This capability allows computers to operate together in data communications networks for distributed processing. Specifically, DIGITAL offers three ways to interconnect computer systems: DIGITAL to DIGITAL (with DECnet communications software), DIGITAL to other manufacturers (with DIGITAL's Internet products), and DIGITAL to public packet switched networks (with DIGITAL's Packetnet products).

Topics include:

- Digital Network Architecture
- Description of a DECnet Network
- DECnet-VAX Phase III features
- Internet Products
- Packetnet Products
- Command Language, FORTRAN, MACRO
- Task Messages
- Programming Procedures

CHAPTER 7
DATA COMMUNICATIONS

INTRODUCTION

DIGITAL computers can communicate with other computers either locally or remotely via a network. A network is a configuration of two or more independent computer systems, called nodes, that are linked together to facilitate remote communication, share resources, and perform distributed processing. Network software can run on different operating systems and communicate with non-DIGITAL equipment. Within the scope of a single network, several nodes with different operating systems and different features can interact to provide increased processing flexibility.

Adjacent network nodes are linked together via carriers known as physical links. Physical links can be relatively permanent bonds, such as telephone lines or cables laid between nodes, or they can be temporary connections that change with each use, such as dial-up telephone links.

In a network, several tasks (programs) can use the same physical link to exchange data simultaneously. Each data path is known as a logical link.

A variety of computer networks can be implemented using DECnet communications software, Internet products, and Packetnet products. They typically fall into one of three classes:

- Communications Networks. These networks exist to move data from one, often distant, physical location to another. The data may be file-oriented (as is often the case for remote job entry systems) or record-oriented (as occurs with the concentration of interactive terminal data). Interfaces to common carriers, using both switched and leased-line facilities, are normally a part of such networks. Such networks are often characterized by the concentration of all user applications programs and databases on one or two large host systems in the network. Figure 7-1 illustrates such a network.

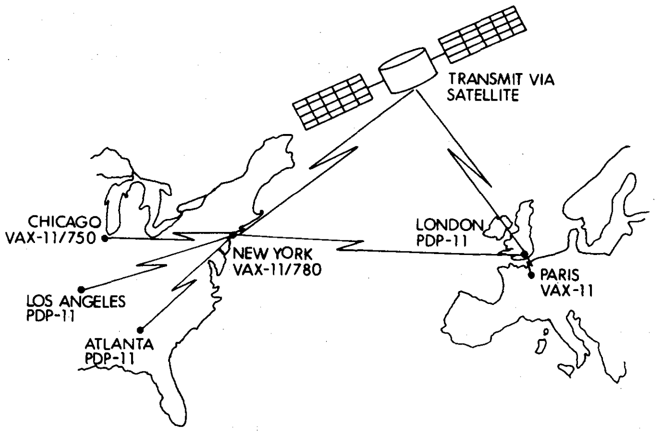


Figure 7-1 Communications Network

- Resource-Sharing Networks. These networks exist to permit sharing expensive computer resources among several computer systems. Shared resources can include not only peripherals such as mass storage devices, but also logical entities, such as centralized databases available to other systems in the network. These networks are often characterized by the concentration of high-performance peripherals, extensive databases, and large programs on one or two host systems in the network. Typically, the satellite systems have less expensive peripherals and smaller programs. Figure 7-2 illustrates a resource-sharing network.

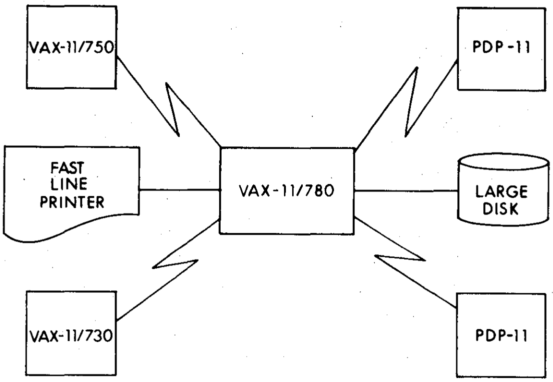


Figure 7-2 Resource-Sharing Network

- Distributed Computing Networks. These networks coordinate the activities of several independent computing systems and exchange data among them. Networks of this nature can have specific geometries (star, ring, hierarchy), but often have no regular arrangement of links and nodes. These networks are usually configured so that the resources of a system are close to the users of those resources. Distributed computing networks are usually characterized by multiple computers with applications programs and databases distributed throughout the network. Figures 7-3 and 7-4 illustrate two such networks.

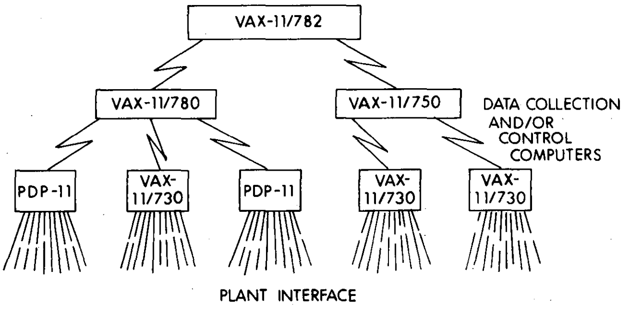


Figure 7-3 Typical Manufacturing Network

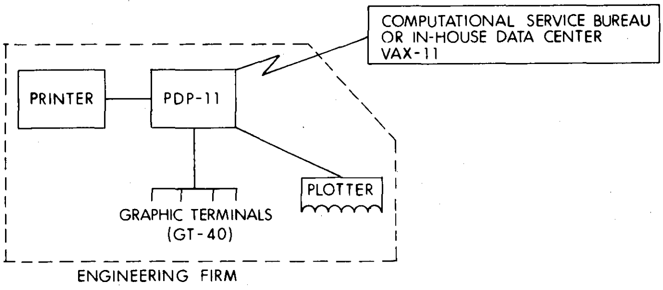

COMPUTATIONAL SERVICE BUREAU OR IN-HOUSE DATA CENTER VAX-11
PRINTER
PDP-11
PLOTTER
GRAPHIC TERMINALS (GT-40)
ENGINEERING FIRM

Figure 7-4 Computational Network

DIGITAL NETWORK ARCHITECTURE

The DIGITAL Network Architecture (DNA) is a set of protocols (rules) governing the format, control, and sequencing of message exchange for all DECnet implementations. DNA controls all data that travel throughout a DECnet network and provides a modular design for DECnet software. The DNA structure is similar to the ISO (International Standards Organization) model for Open Systems Architecture. It permits substitution of functional layers as new technologies become standards.

The functional components of DNA are defined within six distinct layers: User Layer, Network Application Layer, Network Service Layer, Transport Layer, Data Link Layer, and Physical Link Layer. Each layer performs a well-defined set of network functions (via network protocols) and presents an additional level of capability to the layer above it.

USER LAYER: This layer contains all user-supplied functions. It is the highest layer in the DNA structure.

NETWORK APPLICATION LAYER: This layer provides the network functions for the user layer. Modules in this layer include network remote file access modules, a remote file transfer utility, and a remote system loader module. The protocol used for remote file access and file transfer is the Data Access Protocol (DAP).

NETWORK SERVICE LAYER: This layer provides a location-independent communication mechanism for both the user layer and the network application layer. The means by which they communicate is called a **logical** link. Two network application modules may communicate with each other by means of the network service layer regardless of their network locations. The protocol used between network service modules is the Network Services Protocol (NSP).

TRANSPORT LAYER: This layer provides a mechanism for the network service layer to send a unit of data (a packet) from any node in a network to any other node in the network.

DATA LINK LAYER: This layer provides the transport layer with an error-free communication mechanism between adjacent nodes. The data link module specified for this layer implements the DIGITAL Data Communications Message Protocol (DDCMP). The functions provided by this layer are independent of communication facility characteristics.

PHYSICAL LINK LAYER: This layer, the lowest layer in the DNA structure, provides the data link layer with a communication mechanism between adjacent nodes. Several modules are specified for this layer, one for each type of communication device that can be used in a DECnet network.

DNA is system independent. It enables a variety of DIGITAL computers running a variety of DIGITAL operating systems to be tied together in a DECnet network.

A DECnet network can grow both in size and in the number of functions it provides. It can, therefore, be adapted to new technological developments in both hardware and software. Existing DECnet implementations can take advantage of these new technologies. A DECnet network can accommodate the change of a function from software into hardware.

DECNET COMMUNICATIONS SOFTWARE

DECnet communications software provides user interfaces consistent with those provided by DIGITAL's operating systems. To program task-to-task communication or remote file access, programmers use calls formatted for the operating system in which the program will run. The logical link between two programs is like an I/O channel over which programs can send and receive data. Using DECnet software for task-to-task communication is like doing I/O with other peripheral drivers. Terminal users invoke DECnet utilities consistent with local operating system conventions.

Product Capabilities

The network functions available to a DECnet-VAX user depend, in part, on the configuration of the rest of the network. Each DECnet product offers its own functions and its own set of features to the user. Networks consisting entirely of DECnet-VAX Phase III nodes have all the functions described in the DECnet-VAX Phase III section of this chapter. Networks that combine DECnet-VAX nodes with other DECnet products may limit the functions available to the DECnet-VAX user because some DECnet-VAX features may not be supported by all DECnet products. Conversely, a user of another DECnet implementation will not necessarily have access to all DECnet-VAX functions.

The goal of DECnet-VAX is to provide a network capability that is extremely easy to use and can grow with the user. Task-to-task communication and file access between systems is virtually transparent. In fact, VAX/VMS provides the same interfaces as those used by DECnet/VAX to communicate over the network. This means that a user can begin with a single VAX system, develop their applications using these interfaces in the programs and procedures (incidently, at no cost in performance), then expand to a multiple VAX network using DECnet/VAX without having to re-develop their applications software, even if communicating processes are no longer running on the same system.

Programs executing in VAX native mode can make use of the following network facilities.

- Interprocess (Task-to-Task) Communication: Programs executing on one system can exchange data with programs executing on other systems
- Intersystem File Transfer: A program or user can transfer an entire data file from one system to another
- Intersystem Resource Sharing: Programs executing on one system can access files and devices physically located at other systems in the network. Access to devices in other systems is provided through the file system of the target node and is subject to that system's file system restrictions
- Routing: Intermediate nodes will direct data packets to the correct target node if the source node and target node are not directly connected
- Network Command Terminal: A terminal on one VAX system can appear to be connected to another VAX system in the network
- Downline System Loading: Initial load images for RSX-11S systems in the network can be stored on the host VAX system and loaded into adjacent PDP-11 systems configured for the RSX-11S operating system
- Downline Task Loading: Program images for RSX-11S systems in the network can be stored on the host VAX system and loaded on request into PDP-11 systems configured for the RSX-11S operating system
- Downline Command File Loading: Command language users can send command files to a remote node to be executed there. However, no status information or error messages are returned

DECNET-VAX PHASE III COMMUNICATIONS SOFTWARE

With DECnet-VAX Phase III communications software, a suitably configured VAX/VMS system can participate as a routing or end node, performing point-to-point or multipoint communication in a DECnet computer network. The VAX/VMS system can communicate with other DECnet systems on: VAX/VMS systems; PDP-11 computer systems running RSTS/E, RSX-11M, RSX-11S, RSX-11M-PLUS, RT-11, and IAS operating systems; or DECSYSTEM-20 systems running the TOPS-20 operating system.

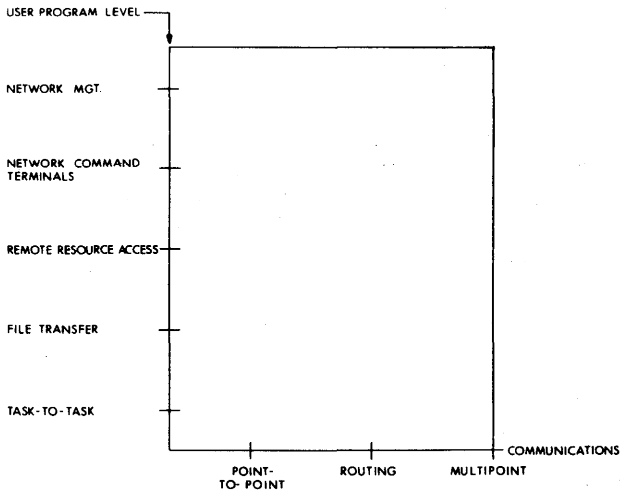


Figure 7-5 DECnet-VAX Phase III Capabilities Matrix

The following functions are supported by DECnet-VAX Phase III software.

Access Control
Access control is the method by which network users are screened before gaining access to network facilities. With the appropriate access control information, a user program can log into a remote system and access any of the remote system's resources. The accessing program must have either an account or access to a guest account on the remote system to login successfully.

Remote File Access
All DECnet systems support exchange of sequential ASCII files. The DECnet software handles compatibility issues among operating systems by translating the file syntax of the sending node into a common network syntax and then retranslating at the receiving end appropriately for that node. The transfer of file types other than sequential ASCII may also be supported between particular operating systems. Between two VAX/VMS systems, for example, sequential or relative files with fixed length, variable length, or variable length with fixed control field records can be transferred. Similarly, multikeyed indexed files with variable or fixed length records are supported.

The Remote File Access capability is implemented by such features as: file transfer, remote command file submission/execution, downline task loading, and terminal-to-terminal communication.

DECnet-VAX software supports file transfers between locally supported File Control Services (FCS) devices and the file system of other DECnet nodes. Wildcards can be used for the user identification code, filename, filetype, and version number for local-to-remote file transfers. Directory listings are also a supported feature.

Additional facilities on DECnet-VAX software allow system command files to be submitted to a remote node. The list of commands must be in a format acceptable to the node responsible for the execution. Similarly, command files can be received from other systems and then executed.

Downline task loading of memory-based RSX-11S nodes is another useful tool provided by some DECnet products. Initial task images for DECnet-11S nodes can be stored on VAX/VMS file system devices and subsequently loaded into remote DECnet-11S nodes. Programs already executing on DECnet-11S nodes can be checkpointed to the host file system and later restored to main memory in the DECnet-11S node. Overlays for DECnet-11S tasks can also be stored on VAX/VMS file system devices. These features simplify the operation of network systems that do no have mass storage devices.

File Handling Using a Terminal
By using DECnet-VAX DIGITAL Command Language (DCL) commands, the user can copy files from one node to another, delete files stored on a remote node, and transfer a command file to another node and then execute the command file on the remote node.

File Handling Using Record Management Services
A wide range of VAX-11 Record Management Services (RMS) can be used to handle files and records stored on remote nodes. At the file level, these operations include opening, closing, creating, deleting, and updating files stored on a remote node. Indexed Sequential Access Method (ISAM) files are supported by DECnet-VAX software as part of its RMS support, thereby allowing remote-node manipulation of files organized by this very useful file structure. Also, at the record level, RMS can be used to read, write, update, and delete records stored on a remote node.

Network Command Terminal Facility
With the Network Command Terminal facility, local users can log onto and use remote VAX systems as though they were local. Network

Command Terminals are a software capability and require no special hardware. They provide virtual terminal communication between VAX/VMS systems. Intermediate nodes (i.e., nodes that are neither the source nor destination nodes but are in the message path) can be running DECnet-VAX or other DECnet Phase III software.

Adaptive Routing

Adaptive routing is a key feature of any DECnet Phase III network. With DECnet-VAX Phase III software, a VAX system can act as a hub node, in which it routes all messages to their proper destination without the need for a physical line directly between the originating node ('A' in the Phase III illustration of Figure 7-6) and the terminating node ('B').

To fully interconnect the four-node network with Phase II DECnet would require 6 lines and 12 modems, with the associated line usage charges and hardware costs. The DECnet-VAX Phase III software can do the same interconnection with potentially half the lines and modems. In addition, the larger the network, the greater the savings in network costs that adaptive routing can provide.

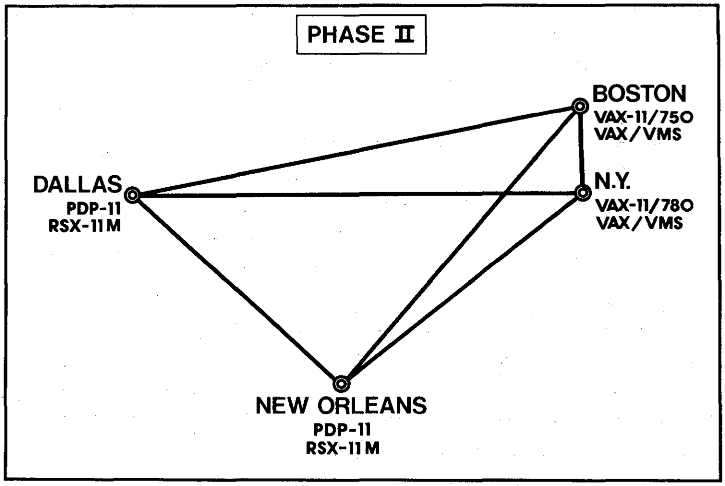
If a line goes down, A DECnet Phase III system will automatically reroute the communication over another line, transparently to the user. This feature enables network managers to easily reroute traffic to avoid a troublesome line or to run diagnostics on such a line. Adaptive routing also makes it possible to install back-up links (which might be dial-up connections) with the result being still fewer connections than with traditional point-to-point networks.

Network Management

The Network Control Program (NCP) performs three primary functions: displaying statistical and error information, controlling network components, and testing network operation. These functions can be performed locally or executed at remote Phase III nodes that support these functions.

An operator can display the status of DECnet activity at any Phase III node in the network. The user can choose to display statistics related to the node itself or the communication lines, including traffic and error data. The local operator can also perform many network control functions such as starting and stopping lines, activating the local node, and downline loading DECnet-11S systems.

DECnet-VAX provides network event logging to a terminal device or disk file. The NCP utility can be used to enable or disable the event logging facility.



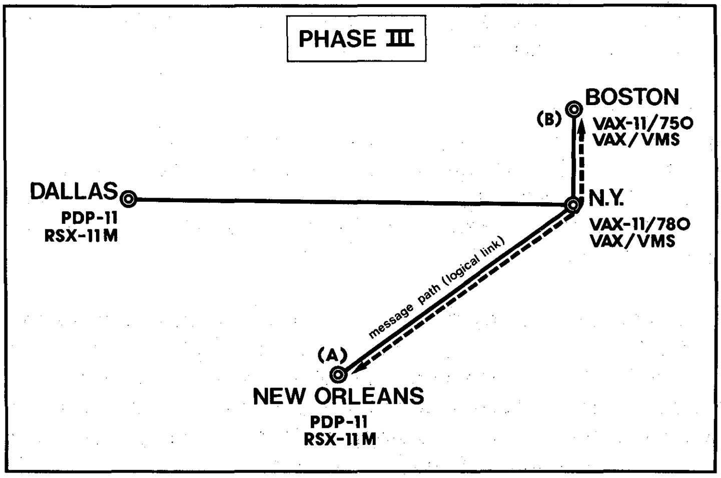


Figure 7-6 Phase II and Phase III Communications

The NCP can also be used to test components of the network. It enables transmission and reception of test messages over individual lines either between nodes or through other controller loopback arrangements. The messages can then be compared for possible errors. The NCP allows performance of a logical series of tests that will aid in isolating network problems.

Task-to-Task Communications

DECnet-VAX software provides task-to-task communication, enabling cooperating programs to exchange data. Task-to-task communication is a method of creating a logical link between two tasks, exchanging data between the tasks, and disconnecting the link when the communication is complete. Any VAX language programmer can write programs that perform task-to-task communication.

Intertask communication routines can be coded using one of two methods: transparent calls or nontransparent calls.

Transparent Intertask Communication — The program opens the network interchange as if it were preparing for device access, and then performs a series of reads and writes just as it would to a pair of serial devices, one for input and the other for output.

By its very nature, transparent access has no calls specifically associated with DECnet software. The calls used for interprocess communication are the same as the calls used for accessing a sequential file in a high-level language: OPEN, CLOSE, READ, WRITE, etc. The programmer can choose to include the target node name in the OPEN statement, or can defer assignment using logical names.

Nontransparent Intertask Communication — In nontransparent access, a program can obtain information about the network status to control the nature of its communication with other processes or tasks. This method of coding intertask communications is available to the MACRO programmer. And if one does no AST processing nor attempts to accept multiple connects, one may program in any language. Nontransparent access is available only through calls to operating system service procedures. A program can issue the following requests:

- CONNECT—Establish a logical link (the analog of OPEN)
- CONNECT REJECT—Reject a connect initiate
- RECEIVE—Receive a message (the analog of GET or READ)
- SEND—Transmit a message (the analog of PUT or WRITE)
- SEND INTERRUPT MESSAGE—Transmit a high-priority message
- DISCONNECT—Terminate a conversation (the analog of CLOSE)

The process can send optional data along with the connect request (for example, the size or number of messages that it wants to send). The receiving process or task can accept or reject the connect initiate. A process can accept multiple connect requests.

A process can send or receive mailbox messages to or from another process or task. Mailbox message traffic is essentially no different from data message traffic except that it uses a mailbox associated with the I/O channel over which the logical link was created. (This is the same mechanism used, for example, for telling programs that unsolicited terminal data is available.) A logical link, therefore, has two subchannels over which messages can be transmitted: one for normal messages and another for high-priority messages. An interrupt message is written to a mailbox that a process supplies for that purpose.

In a DECnet-VAX network, a program using nontransparent access normally opens a control path directly to a Network Ancillary Control Process (NETACP), and designates one or more mailboxes for receiving information from the NETACP about the logical or physical links over which the process is communicating. The NETACP can notify a process when (a partner being a source and destination node with a logical connection):

- A partner requests a synchronous disconnect
- A partner requests a disconnect abort
- A partner exits
- A physical link goes down
- An NSP protocol error is detected

DIGITAL COMMAND LANGUAGE (DCL) FILE HANDLING

A VAX/VMS DIGITAL Command Language (DCL) user can transfer files from one node to another and delete files at other nodes. However, to perform operations on files stored on a remote node, the user must prefix the file specification with the remote node's name, and an optional login string as follows:

```
    nodename"access-control-string"::filename.filetype;version
```

where:

nodename	Nodename is a 1- to 6-character name (numerics or upper case alphabetics) identifying the remote network node. This can be followed by an optional quoted string used for logging in at the remote node.

access-control-string	If the "access-control-string" is omitted, default login information comes from an entry (for the remote node) in the local configuration data base. Thus, by using the access-control-string, the user overrides the default login information. One of the following formats is used for the access-control-string: "username password" "username password accountname" The double colon (::) following the nodename separates the nodename from the file specifier.
filename filetype version	See the Chapter 2, The System User, for details of these three. But note that the way in which a file on a remote node is identified depends on the *remote node's* operating system. The following format is used if the remote node is a DECnet-VAX node: device:[directory]filename.filetype;version If, however, the remote node is not a DECnet-VAX node, enclose the file specifier between quotation marks. The file specifier is passed to the remote node without syntax checking.

DECnet-VAX software supports the following subset of VAX/VMS (DCL) commands:

APPEND
ASSIGN
COPY
DEASSIGN
DEFINE
DELETE
DIRECTORY
OPEN/CLOSE
READ/WRITE
SUBMIT
TYPE

The following examples illustrate the COPY and SUBMIT commands:

```
$ COPY BOSTON::DBA1:TEST.DAT DENVER::DMA2:
```

transfers a file named TEST.DAT from the disk (DBA1:) at the node named BOSTON to the disk (DMA2:) at the node named DENVER.

Using the VAX/VMS command SUBMIT, a terminal user can have a command file executed at another node in the network. For example, the command:

```
$ SUBMIT/REMOTE WASHDC::INITIAL.COM
```

preceded by a DCL COPY command will transfer the command file named INITIAL.COM from the host system to the node named WASHDC, where the command file is executed. The SUBMIT command assumes that the file already exists at the remote node. Command files must be written in the command language of the system. No status information or messages are returned to the sender.

RECORD MANAGEMENT SERVICES FILE HANDLING

By using a subset of the VAX-11 Record Management Services (RMS), the user can manipulate records and files stored on remote DECnet nodes. However, before using VAX-11 RMS to perform operations on files and records stored on a remote node, the user must prefix the file specification with the node name of the remote node, and an optional login string just as with any other remote file application.

Much of the VAX-11 RMS functionality, including the management of sequential, relative, and indexed file organizations, is supported by DECnet-VAX communications software. A large number of the VAX-11 RMS macros are available to network users.

The following MACRO program illustrates the transfer of a sequential file from one device to another. Note the use of VAX-11 RMS macros.

```
                        .TITLE DEMO1 - RMS FILE TRANSFER EXAMPLE
;
;                       This program transfers a sequential file with variable length
;                       records from one device to another. The devices are specified
;                       by the logical names SRC and DST. For example, to display file
;                       INVENTORY.DAT residing at node ALBANY on the line printer at node
;                       BOSTON, execute the following procedure:
;
;                       $ DEFINE SRC ALBANY::DBB3:[XYZCO.STOCK]INVENTORY.DAT
;                       $ DEFINE DST BOSTON::LPA0:
;                       $DEMO1
;
                        .SBTTL CONTROL BLOCK AND BUFFER STORAGE
        .PSECT                          IMPURE NOEXE.LONG
;
;                       Define the source file FAB and RAB control blocks.
;
SRC_FAB:
        $FAB                            FAC=GET,- ; GET ACCESS
                                        FOP=SQO,- ; SEQUENTIAL ONLY
                                        FNA=SRC_NAM,- ; ADDRESS OF FILENAME STRING
                                        FNS=SRC_NAM-SIZ ; SIZE OF FILENAME STRING
```

```
SRC_RAB:
         $RAB                  FAB=SRC_FAB,- ; ADDRESS OF FAB
                               RAC=SEQ,- ; SEQUENTIAL RECORD ACCESS
                               UBF=BUFFER,- ; ADDRESS OF USER BUFFER
                               USZ=BUFFER_SIZ ; SIZE OF USER BUFFER
;
;             Define the destination file FAB and RAB control blocks.
;
DST_FAB:
         $FAB                  FAC=<PUT>,- ; PUT (WRITE) ACCESS
                               FOP=SQO,- ; SEQUENTIAL ONLY
                               FNA=DST_NAM,-
                               FNS=DST_NAM_SIZ,-
                               ORG=SEQ,- ; SEQUENTIAL FILE (DEFAULT)
                               RFM=VAR,- ; VARIABLE LENGTH RECORDS
                               RAT=CR ; PRECEDE LINE BY LF, FOLLOWED BY CR
DST_RAB:
         $RAB                  FAB=DST_FAB,-
                               RAC=SEQ
;
;             Define logical names for the source and destination files.
;
SRC_NAM:
         .ASCII                /SRC/

              SRC_NAM_SIZ==.-SRC_NAM
DST_NAM:
         .ASCII                /DST/

              DST_NAM_SIZ==.-DST_NAM
;
;                   Allocate buffer space to be size of largest record.
;
BUFFER:
           .BLKB                              132
                   BUFFER_SIZ=.-BUFFER

           .SBTTL                             MAINLINE

           .PSECT                             CODE NOWRT


           .ENTRY                             DEMO1,↑M<>


;
;                   Put FAB and RAB addresses in registers for efficiency.
;
           MOVAB                              W↑SRC_FAB,R6
           MOVAB                              W↑SRC_RAB,R7

           MOVAB                              W↑DST_FAB,R8

           MOVAB                              W↑DST_RAB,R9

;
;                   Open the SRC and DST files.
;
           $OPEN                              FAB=(R6)
           BLBC                               R0,30$

           $CONNECT                           RAB=(R7)
```

```
                BLBC                        R0,30$

                $CREATE                     FAB=(R8)
                BLBC                        R0,30$

                $CONNECT                    RAB=(R9)

                BLBC                        R0,30$

;
;                       Transfer records until end-of-file is encountered.
;
10$:            $GET                        RAB=(R7)
                CMPW                        R0, #<RMS$W_EOF&↑XFFFF>

                BEQL                        20$

                MOVL                        RAB$L_UBF(R7),RAB$L_RBF(R9)

                MOVW                        RAB$W_RSZ(R7),RAB$W_RSZ(R9)
                $PUT                        RAB=(R9)

                BLBS                        R0,10$

                BRB                         30$

;
;                       Close the SRC and DST files.
;
;                  Note: in this example, the $DISCONNECT calls below are not
;                  necessary because $CLOSE performs an implied
;                  $DISCONNECT. They are included for symmetry.
;
20$:        $DISCONNECT     RAB=(R9)
            BLBC            R0,30$
            $CLOSE          FAB=(R8)
            BLBC            R0,30$
            $DISCONNECT     RAB=(R7)
            BLBC            R0,30$
            $CLOSE          FAB=(R6)
;
;                       Exit to VMS. Also, enter here on detection of an error.
;
30$:        $EXIT_S R0      ; R0 = RMS completion code to
                            ; display on error condition
                .END                        DEMO1
```

The following VAX-11 FORTRAN program illustrates the transfer of a sequential file from one device to another.

```
          PROGRAM DEMO1.FOR
C
C         This program transfers a sequential file with variable length
C         records from one device to another. The devices are specified
C         by the logical names SRC and DST. For example, to display file
C         INVENTORY.DAT at node ALBANY on the line printer at node
C         BOSTON, execute the following procedure:
```

```
C
C           $ DEFINE SRC ALBANY::DBB3:[XYZCO.STOCK]INVENTORY.DAT
C           $ DEFINE DST BOSTON::LPA0
C           $ RUN DEMO1
C
            LOGICAL*1 BUFFER(132)
C
100         FORMAT      (Q,132A1)
200         FORMAT      (132A1)
C
C           Open the SRC and DST files.
C
            OPEN        (UNIT=1,NAME='SRC',TYPE='OLD',ACCESS='SEQUENTAL',
            1           FORM='FORMATTED')
            OPEN        (UNIT=2,NAME='DST',TYPE='NEW',ACCESS='SEQUENTIAL',
            1           FORM='FORMATTED',CARRIAGECONTROL='LIST',DISPOSE='SAVE')
C
C           Transfer records until end-of-file is encountered.
C
10          READ        (1,100,END=20)  NCHAR,(BUFFER(K),K=1,NCHAR)
            WRITE       (2,200,END=20)  (BUFFER(K),K=1,NCHAR)
            GOTO        10

C
C           Close the SRC and DST files.
C
20          CLOSE       (UNIT=2)
            CLOSE       (UNIT=1)
C
            END
```

SAMPLE VAX-11 FORTRAN INTERTASK COMMUNICATION

This section describes how to code a program to perform intertask communication using the normal VAX-11 FORTRAN I/O instructions. The user communicates with another task in much the same way as an access to a sequential file, i.e., via OPEN, READ, WRITE and CLOSE statements. This is only a sample—similar capabilities exist in any of the native mode languages.

Three major steps in VAX-11 FORTRAN intertask communication will be performed:

1. Create a logical link between tasks
2. Send and receive messages (each message can be 1 to 512 bytes in length)
3. Destroy the link at the end of the message dialogue

Creating a Logical Link Between Tasks

A logical link between tasks can be created only if they agree to co-operate with each other. That is, one task must request that a logical link be created, and the other must agree to accept the request. The task requesting the logical link is called the source task; the one agreeing to accept the logical link request is called the target task.

The source task issues a logical link connect request by including a task specifier in the source task's open statement. The task specifier identifies the remote node and target task to be connected to. The normal file specification in the OPEN statement's NAME argument should be replaced with a task specifier. The following format should be used:

```
nodename"access-control-string"::"TASK=taskid"
```

where:

nodename	(Refer to DIGITAL Command Language File Handling section of this chapter.) Use one of the following formats for access-control-string: "username password" "username password accountname" The double colon (::) following the nodename separates the nodename from the file specifier.
TASK=	Specifies that what follows is the task identifier.
taskid	taskid is the target task's identifier.

Example of source task OPEN statement:

```
OPEN (UNIT=7,NAME='DENVER::"TASK=ACC"' , ERR=200)
```

The NAME argument in the OPEN statement requests a logical link connection to target task ACC on node DENVER.

Note that the logical name feature can be used to represent the task specifier. For example:

```
OPEN (UNIT=7,NAME='TARGET',ERR=200)
```

permits node and target independence when you assign the logical name before program execution (as in the following DCL command):

```
$ASSIGN DENVER::"""TASK=ACC """ TARGET
```

The local node passes the logical link connect request to the remote node (using DECnet-VAX services). The remote node creates a process for the target task, and places the source task identifier in the process logical name table under the logical name SYS$NET.

The target task identifies the source task requesting the logical link connect by specifying SYS$NET as the NAME argument in the OPEN statement.

Example of target task OPEN statement:

```
OPEN (UNIT=2,NAME='SYS$NET:',ERR=700)
```

Sending and Receiving Messages

After the logical link has been created, the tasks must "cooperate" with each other. That is, for each message sent by a task (WRITE statement), the receiving task must issue a corresponding receive (READ statement).

In addition, the tasks must ensure that enough buffer space is allocated for messages, must ensure that the end of dialogue can be determined, and must determine which of the tasks will disconnect the logical link (CLOSE statement).

Disconnecting the Logical Link

Either task can disconnect the logical link by calling CLOSE. CLOSE aborts all pending sends and receives, disconnects the link immediately, and frees the channel number associated with the logical link.

MACRO TRANSPARENT INTERTASK COMMUNICATION

This section describes how to code a MACRO program for transparent intertask communications using a subset of the existing macro calls available under VAX/VMS system services. These macro calls allow the user to perform intertask communications in much the same way as normal I/O operations are performed.

The term "transparent" simply implies that the calls are identical in format to all other I/O calls.

Thus, communication with another task is performed in much the same way as an I/O channel is assigned to a device ($ASSIGN). Reads and writes are then performed as if to a pair of sequential devices (that is, $QIO with the _WRITEVBLK function or $OUTPUT, and $QIO with the IO$_READVBLK function the JO$ or $INPUT). Finally, $DASSGN the device when communication is complete.

There are three major functions in transparent intertask communication:

1. Create a logical link between tasks

2. Send and receive messages (each message can be 0 to 65535 bytes long)

3. Delete the link at the end of the message dialogue

Creating a Logical Link Between Tasks

A logical link between tasks can be created only if the tasks agree to cooperate with each other. That is, one task must request that a logical link be created, and the other task must agree to accept the request.

A logical link is requested by including a task specifier in the source task's $ASSIGN call.

A task specifier identifies the remote node and the target task to which it is to be connected. Replace the normal file specification in the $ASSIGN call's devnam argument with a task specifier.

The local node passes the logical link connect request to the remote node (using DECnet-VAX services). A remote VAX node creates a process for the target task, and places an equivalence string containing the source task identifier in the process's logical name table for the logical name SYS$NET.

The target task identifies the source task requesting the logical link connect request by specifying SYS$NET as the devnam argument in the $ASSIGN statement. This action completes the creation of the logical link.

Sending and Receiving Messages

After the logical link is created, the tasks must "cooperate" with each other. That is, for each message sent by a task ($QIO with the IO$_WRITEVBLK function or $OUTPUT), the receiving task must issue a corresponding receive ($QIO with the IO$_READVBLK function or $INPUT).

In addition, the tasks must ensure that enough buffer space is allocated for messages, must ensure that the end of dialogue can be determined, and must decide which of the tasks will disconnect the logical link ($DASSGN).

Disconnecting the Logical Link

Either task can disconnect the logical link by calling $DASSGN. $DASSGN aborts all pending sends and receives, disconnects the link immediately, and frees the channel number associated with the logical link.

MACRO CALLS

Listed below are the VAX/VMS system service macro calls that can be used for transparent intertask communications.

$ASSIGN—Assign I/O Channel

$QIO—Send a Message to a Remote Task $QIO (IO$_WRITEVBLK)

$QIO—Receive a Message from a Remote Task $QIO (IO$_READVBLK)

$DASSGN—Disconnect the Logical Link

MACRO NONTRANSPARENT INTERTASK COMMUNICATION

As with transparent intertask communication, nontransparent intertask communication consists of two tasks interacting to establish a logical link. After establishing the logical link, the tasks exchange messages over the link, then disconnect the link when communication is completed.

The MACRO system service calls discussed in this section provide the user with greater flexibility and control over network operations. The following features can be used when performing nontransparent intertask communication:

- Associate a mailbox with the I/O channel (over which the logical link will be created). The mailbox can then receive unsolicited messages sent by a remote task, or notifications affecting the status of the logical link. For example, status returned through a mailbox includes whether the remote task accepted or rejected a connect, or the cooperating task disconnected or destroyed the link
- A task can declare itself as a network task to accept multiple logical link connect requests
- A source task can send a logical link connect request to the target task. The source task can optionally send up to 16 bytes of data to the target task at the same time it issues the connect request
- The target task can accept or reject the connect request. It can send up to 16 bytes of optional data back to the source task at the same time it accepts or rejects the connect request
- A task using the nontransparent interface can also accept or reject connect requests received from tasks written using transparent intertask communication system service calls
- Either task can send or receive a 1- to 16-byte interrupt message after the logical link is created
- Either task can abort the link immediately, or issue a synchronous disconnect. The task disconnecting or aborting the logical link can send up to 16 bytes of optional data to the remote task at the same time it disconnects or aborts a logical link

TASK MESSAGES

There are two types of messages in nontransparent intertask communications: data messages and mailbox messages.

Data Messages

A data message is a message sent by one task, and expected by the cooperating task. That is, for each message sent by a task $QIO with the IO$_WRITEBLK function or $OUTPUT, the receiving task must issue a receive $QIO with the IO$_READVBLK function or $INPUT.

Thus, a data message in nontransparent intertask communications is the same as a data message sent in transparent communication.

Mailbox Messages

All other messages received by a task employing a nontransparent interface are classified as mailbox messages. These include any one of the following message types:

1. A logical link connect request—This message is received by the target task. It requests a logical link connection to the source task
2. A connect accept—This message is received by the source task. The message confirms that the target task accepted the logical link connect request
3. A connect reject—This message is also received by the source task. The message informs the source task that the target rejected the logical link connect request
4. An interrupt message—Either task can receive a 1- to 16-byte interrupt message sent by a cooperating task. The 1 to 16 bytes of data are placed in the task's mailbox
5. A synchronous disconnect—This message informs the task that the cooperating task synchronously disconnected the logical link
6. An abort disconnect—This message informs the task that the cooperating task aborted the link. The link is destroyed immediately
7. A network status message—This message informs the task of some unusual network occurrence. For example, the data link has been restarted

After a logical link is created between cooperating tasks, DECnet communications software places a received mailbox message into the mailbox associated with the channel representing the logical link to which the mailbox message applies.

In the case of a task that can accept multiple inbound connect requests, inbound connect requests are placed into the mailbox associated with the I/O channel over which the network name was declared.

Note that the mailbox was previously created using the $CREMBX system service call. The task must then explicitly retrieve the mailbox message from the mailbox using the $QIO(IO$_READVBLK) system service call.

PROGRAMMING PROCEDURES

The following sections outline the procedures to follow when using the system service calls for intertask communications.

Creating a Logical Link

Both the source and target tasks must call the $ASSIGN system service call to:

1. Assign to device_NET0:
2. Request a channel number for the logical link
3. Associate a previously created mailbox with the channel

After creating the logical link, DECnet communications software places any mailbox message associated with the logical link in the mailbox associated with the channel.

Note that the $ASSIGN (to device NET0:) does not transmit a logical link connect request to the remote task (as in transparent intertask communications).

Source Task Requests a Logical Link Connection

The source task calls $QIO(IO$_ACCESS) to request a logical link connection to the target task. The source task may optionally send up to 16 bytes of data to the target task at the same time it sends the connect request.

The target task is identified in the $ASSIGN call by specifying the target task's identifier in the network control block.

The Network Connect Block (NCB) contains the information necessary to request a logical link connection, or to accept or reject a logical link connection request.

The optional data to be sent to the target task are also specified as part of the network connect block.

The source task must then call $QIO(READVBLK) to read its mailbox to determine whether the target task accepted or rejected the connect request.

Target Task Receives Connect Request

The target task determines whether to accept or reject a connect request, possibly by reading the received connect block. The received connect block contains the source task identifier, as well as up to 16 bytes of optional data sent by the target task.

The manner by which the target task retrieves the received connect block depends on whether the target task can receive single or multiple connect requests.

A target task can accept multiple connect requests only if it declares itself as an active network task. Thus, it assigns an I/O channel to NET0: first, then calls $QIO(IO$_ACPCONTROL) to assign a network number and declare itself eligible to accept multiple connect requests.

After this is done, DECnet places the first and all other connect requests in the task's mailbox. The target task then retrieves a connect request from its mailbox by calling QIO(IO$_READVBLK).

If the target task can accept only one connect request, it need not declare itself as a network task. The target task retrieves the connect block by translating the logical name SYS$NET using the $TRNLOG system service call.

Accepting or Rejecting a Connect Request
The target task accepts or rejects the connect request by:

1. Calling $QIO(IO$_ACCESS) to accept the logical link connect request, or
2. Calling $QIO(IO$_ACCESS!IO$M_ABORT) to reject the logical link connect request [Note that ! is OR]

In both cases, an unsolicited message is sent back to the source task's mailbox confirming or rejecting the connect request. The target can send up to 16 bytes of optional data back to the source task at the same time it accepts or rejects the logical link connect request.

Sending and Receiving Data Between Tasks
DECnet delivers a solicited message only if it has been sent by one task and solicited by the cooperating task. Thus, after the logical link is created, the tasks must "cooperate" with each other. That is, for each message sent by a task ($QIO(IO$_WRITEVBLK) or $OUTPUT), the receiving task must issue a corresponding receive ($QIO(IO$_READVBLK) or $INPUT).

Note that the term "cooperating" here implies that the tasks:

1. Create buffers large enough to send and receive data messages
2. Have agreed upon a protocol for sending and receiving data

Sending an Interrupt Message
Either task can send a 1- to 16-byte interrupt message to a cooperating task using the $QIO(IO$_WRITEBVLK!IO$M_INTERRUPT) system service call.

In this case, "interrupt message" is the term for an unsolicited message sent to the cooperating task's mailbox, and should not be confused with a hardware or software interrupt. It is a method that can be used to send a message to a remote task outside the normal flow of data messages. A task's instruction sequence is interrupted only if it issued a request to read its mailbox with AST (Asynchronous System Trap) notification.

Disconnecting or Aborting the Logical Link
Either task can disconnect or abort the logical link by:

- Calling $QIO(IO$_DEACCESS!IO$M_SYNCH) to synchronously disconnect the logical link. All pending solicited and unsolicited messages must have been transmitted to the remote node before the link will be disconnected. DECnet returns an error if the user tries to disconnect the link before all pending transmits are transmitted. Any pending receives are terminated.
- Calling $QIO(IO$_DEACCESS!IO$M_ABORT) to abort the logical link. This system service call destroys the link immediately. No further I/O operations are permitted on the link.

Deassigning the I/O Channel
The user can issue $DASSGN after all communication between the tasks is complete. $DASSGN releases the I/O channel and disassociates the mailbox from the channel. Also, if a synchronous disconnect or abort was not previously issued, $DASSGN destroys the link immediately.

MACRO CALLS

This section lists the VAX/VMS system service macro calls that can be used for nontransparent intertask communication coding. These calls are:

$ASSIGN—Assign I/O Channel

$QIO—Request a Logical Link Connection $QIO (IO$_ACCESS)

$QIO—Accept a Logical Link Connection Request $QIO (IO$_ACCESS)

$QIO—Reject a Logical Link Connection Request $QIO (IO$_ACCESS!IO$M_ABORT)

$QIO—Send a Message to a Remote Task $QIO (IO$_WRITEVBLK)

$QIO—Receive a Message from a Remote Task $QIO (IO$_READVBLK)

$QIO—Send an Interrupt Message to a Remote Task $QIO (IO$_WRITEVBLK!IO$M_INTERRUPT)

$QIO—Synchronously Disconnect the Logical Link $QIO (IO$_DEACCESS!IO$M_SYNCH)

$QIO—Abort a Logical Link $QIO (IO$_DEACCESS! IO$M_ABORT)

$QIO—Declare a Network Name $QIO (IO$_ACPCONTROL)

$DASSGN—Disconnect the Logical Link

INTERNET PRODUCTS

DIGITAL's Internet family of products supports the interconnection of DIGITAL computers and DIGITAL networks to systems built by other manufacturers. Internets give data processing managers the freedom to choose mainframes and minicomputers on the basis of application needs, with the assurance that reliable links can be established between systems.

Internet products emulate common communications protocols. They are data transfer facilitators rather than hardware emulators. While they appear to one another vendor's computers to be supported devices, they are, in fact, parts of powerful DIGITAL systems. They provide transparent communication with the equipment of other vendors, and at the same time, offer the flexibility of local file systems, many different languages, and a wide selection of computing power.

VAX-11 2780/3780 Protocol Emulator

The VAX-11 2780/3780 protocol emulator provides the VAX/VMS user with a mechanism for transferring files between a VAX system and another system equipped to handle IBM 2780 or 3780 communications protocols. It does this by emulating the IBM Binary Synchronous Communications (BISYNC) protocol used by a 2780 or 3780 Remote Batch Terminals.

The emulator may be invoked either interactively or by a command procedure. The emulator's command set is designed to facilitate sharing a communication line among several users. With the appropriate modem options, the emulator is capable of automatically answering incoming calls.

Sophisticated operations can be performed by a combination of command procedures, allowing, for example, unattended operation. This would include the capability to detect an incoming call, establish the connection, and then transmit and receive files and recover from transmission failures, all without the intervention of the operator.

Several data formats are supported with the use of a particular format selected by user command. Users may select various data format translation schemes—for example, translation to and from EBCDIC, converting variable-length records to and from card images, and BISYNC transparency. All file I/O is performed through the VAX/VMS record management facility. Print and punch stream recognition is implemented in such a way that the data manipulation scheme can differ with each stream.

The following remote batch terminal features are supported:

- 2780 Extended and Multiple Record Option
- Variable Horizontal Forms Control
- BISYNC Transparency
- 3780 Space Compression

All of the above features are supported on a simultaneous, multiline basis. The product can concurrently run up to four physical lines, each with a different set of attributes (e.g., some may be 2780, the others, 3780) at speeds up to 9600 bits per second per line.

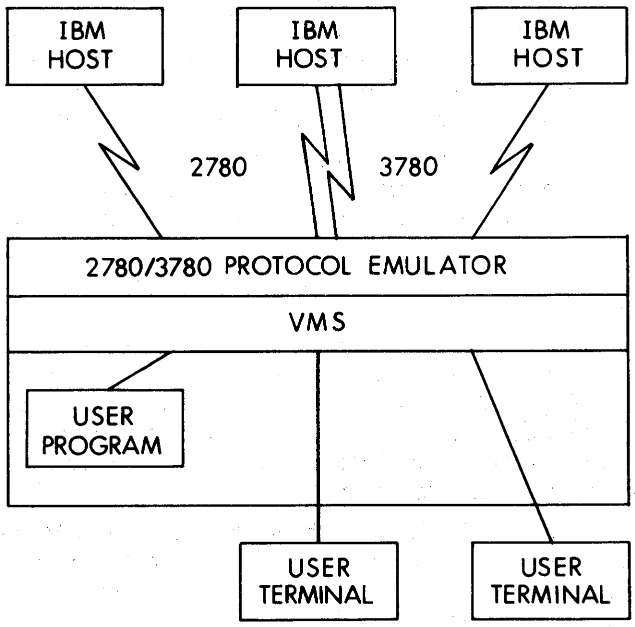


Figure 7-7 VAX-11 2780/3780 Protocol Emulator

Additionally, the VAX-11 2780/3780 protocol emulator can be used in conjunction with DECnet networks, meaning that VAX systems in a DIGITAL network can also communicate with IBM systems. For example, files can be transferred across a DECnet network to a DECnet-VAX node and from there to a mainframe bisynchronously.

VAX-11 3271 Protocol Emulator

The VAX-11 3271 protocol emulator provides VAX system users with an interactive program-to-program link to an IBM mainframe. This emulator enables a user application program on a VAX system to exchange data with a program running under CICS or IMS on an IBM host (System/370). Using two application programs -- one for the DIGITAL side, and one for the IBM side -- the VAX system user can both send and receive data.

The user-written program on the VAX system communicates with the IBM application program by issuing I/O requests to the VAX-11 3271 protocol emulator. The emulator, appearing to the host as an IBM 3271 Model 2 Control Unit, interacts with the IBM system to perform the actual mechanics of the data exchange. It manages the protocol and the physical communications in a manner that is transparent to the VAX system user. As part of its protocol management, for example, it frames the data with appropriate BISYNC link-control characters: Start of Text, Unit Indentification, CRC's, and End Text.

Application programs that can be used for 3270 operations on the VAX system include programs to access IBM databases and transaction-oriented applications and programs to emulate 3270-class terminals.

The communications discipline used by the VAX-11 3271 protocol emulator is the 3271 subset of IBM's Binary Synchronous Communications (BISYNC) protocol using EBCDIC code. Specifically, this subset of BISYNC supports operation of full- and half-duplex leased lines in either point-to-point or multipoint configurations at transmission speeds up to 9600 bits per second. The VAX-11 3271 protocol emulator does not support switched facilities, contention line control, or transparent BISYNC capability.

The multidrop BISYNC capability enables the VAX-11 3271 protocol emulator to coexist with other 3271 controllers on the same line, thus reducing the required number of communications links. On a VAX-11/780 system, the emulator can be connected on up to four IBM systems, or can have four lines to a single system for higher throughput. On a VAX-11/750 system, the emulator supports two lines.

The VAX-11 3271 protocol emulator can support up to 32 logical devices for each "control unit" being emulated. A user application program can control one or more logical devices. A maximum of 32 user application programs, one per logical device, can exist for each control unit.

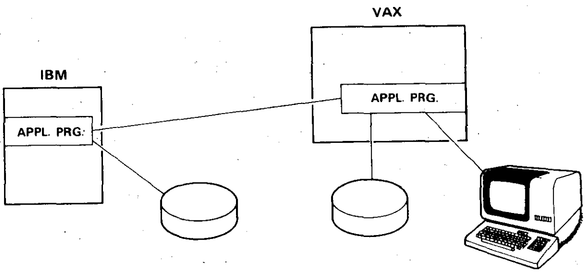


Figure 7-8 VAX-11 3271 Protocol Emulator

The VAX-11 3271 protocol emulator performs the following key activities:

- Provides all device handling for DUP11 Synchronous Line Interfaces
- Monitors the communications line via DUP11 Synchronous Line Interfaces for line errors
- Performs blocking of user data during transmission
- Supports an I/O interface with the user program
- Maintains the multipoint BISYNC protocol with the IBM host: ensures data integrity of transmitted and received data; processes polling, selection addressing sequences, and other protocol functions normally handled by an IBM 3271
- Translates incoming data from EBCDIC to ASCII and outgoing data from ASCII to EBCDIC
- Supports multiple lines

The VAX-11 3271 protocol emulator can also be used to complement a DECnet-VAX network. Data can be transferred across a DECnet network to a user application program in a DECnet-VAX node and from there to a mainframe bisynchronously.

MUX200/VAX Multiterminal Emulator

The MUX200/VAX multiterminal emulator is a VAX-based software package which provides communication with a CDC6000, CYBER series, or other host computer systems capable of using 200 UT mode 4A communications protocols.

Any VAX interactive terminal may be used to control remote job entry or to communicate at command level with the host system. Input files may be sent from, and output files received onto, any VAX-supported mass storage, unit record, or terminal device.

The MUX200/VAX emulator communicates with the host using the Mode 4A communications protocol as defined in CDC publication 82128000. The software package can be configured to support either the ASCII or the external BCD versions of the protocol.

The MUX200/VAX emulator provides for one synchronous communication circuit to a host computer system. The product supports a single switched or dedicated leased line two- or four-wire common carrier facility at speeds up to 9600 bits per second.

With the MUX200/VAX emulator, several users can communicate simultaneously with a host system over a single line. The VAX/VMS system, though using a single physical drop, appears to the host as a number of multidrops and terminals on the circuit.

MUX200/VAX features include:

- Output received from the host system may be spooled to the line printer upon detection of a text string predefined by the user
- Up to eight VAX/VMS files may be specified for transmission to the host in a single command
- VAX/VMS terminals may be detached for other use while the software package is operating. Data received from the host directed to a terminal are saved for printing until the terminal is reattached
- In many applications the host system can be offloaded by taking advantage of the local processing power of the VAX/VMS system. This reduces host processing and line costs; for example, file editing can be performed locally rather than on the host

See Figure 7-8 for a schematic of the MUX200 use.

PACKETNET PRODUCTS

In the 1970's the International Telephone and Telegraph Consultative Committee—CCITT—developed a series of recommendations for standard communications protocols that could be used by the PTTs and other commmon carriers to provide data communications services. Known as X.25, this recommendation developed for the public data networks defines the interface between the computer and the network. The CCITT has also developed a second set of protocols (X.3, X.28, and X.29) that specify how to control asynchronous terminals connected directly to a network. Together, these protocols define the Interactive Terminal Interface (ITI).

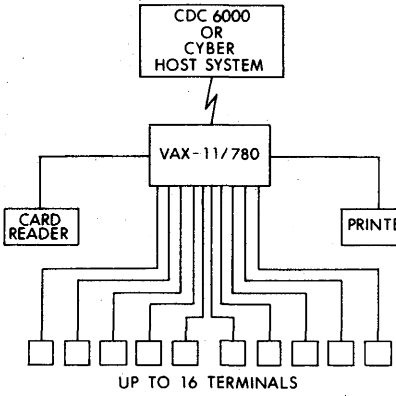


Figure 7-9 MUX200 Schematic

The fundamental technology used in public data networks is called packet switching. With it, user data and control information needed to assure delivery to the correct location are formed into discrete entities—packets. The network dynamically interleaves the packets of many users over shared transmission facilities, and routes packets to their destinations. Unlike conventional telephone setups, where the user is charged for both connect time and distance, regardless of the amount of data passed, charges in public data networks are determined so that the person who uses the line the most pays the most.

Rapidly, X.25 is becoming the standard international communications protocol. As another advantage, X.25 allows computers from different manufacturers to work together: with appropriate security validation, any system on the network can send data to any other system on the network. X.25 ensures data integrity, while at the same time relieving users of any concern about input and output speeds of the various processors in the network.

Public data networks are currently operational in the United States (the Telenet and Tymnet networks), Canada (the Datapac network), France (the Transpac network), Germany (the Datex network), and the United Kingdom (the PSS network). Within the near future, the DNI network will be available in Holland. DIGITAL is committed to support public data networks using the full X.25 communications protocol, and is therefore engaged in the development of products to link DIGITAL systems to these networks.

VAX-11 PSI (Packetnet System Interface)
VAX-11 PSI (Packetnet System Interface) software allows a suitably configured VAX/VMS system to connect to public data networks. This enables a VAX system to converse with any other computer (DIGITAL or non-DIGITAL) that has implemented the X.25 protocol.

The VAX-11 PSI interface offers process-to-process communications via the PSS network. It also permits remote terminal access to the VAX/VMS operating system using an Interactive Terminal Interface. Access to VAX-11 PSI software is supported for VAX/VMS user programs written in the VAX-11 MACRO assembly language and in the VAX higher-level languages such as VAX-11 FORTRAN.

For interprocess communication, application programs use VAX/VMS system services to set up and break connections with the network, to send and receive data, and to issue control and synchronization requests.

X.25 User Interface
The X.25 User Interface allows application programs to use the X.25 functions, including:

- Establishing and clearing connections
- Sending and receiving data
- Sending interrupt messages
- Receiving unsolicited messages
- Controlling errors and reporting status

For the occasional network user, the PSI software can be loaded into memory only when needed.

Interactive Terminal Interface
Remote terminals have the same access privileges to VAX programs as they would if they were local. Thus, it is possible to run applications programs across the PSS network with no modification, unless the network itself imposes restrictions which are beyond DIGITAL's control.

VAX-11 PSI software supports access by remote terminals according to CCITT recommendations X.3, X.28, and X.29. Terminals are supported in 'Network Terminal' mode, in which code conversions between ASCII and the actual code used by the terminal are performed by the network.

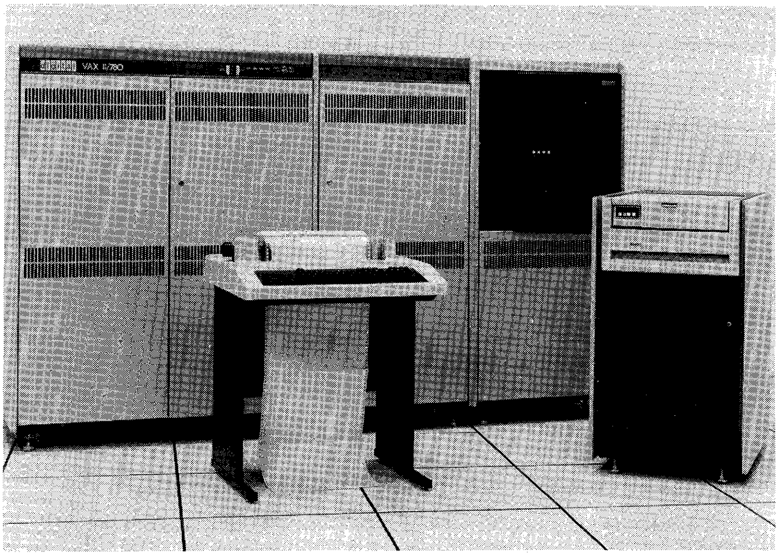
Line Discipline
The communications discipline used is the CCITT recommendation X.25. Specifically, the product supports V.24 (EIA-RS-232) at the hard-

ware level, the symmetric LAPB variant of the X.25 frame level protocol and the X.25 packet level protocol over point-to-point, 4-wire, synchronous, full-duplex lines, at speeds up to a maximum of 9600 bits per second.

Network Management

A system management command interface is provided for the control of the operation of the X.25 software. This includes loading and unloading the software, defining the lines and network to which the system is connected, specification of addressing information for incoming calls, and access to error counters and other maintenance functions. This interface provides a subset of the DIGITAL Network Architecture (DNA) specification for Network Management.

PART III
VAX/VMS SYSTEM DESIGN AND APPLICATION



CHAPTER OVERVIEW

Sophisticated techniques of memory management and concepts and details of virtual memory design are explained in this chapter. The division of virtual address space into various regions plus the kinds of information that can be resident in each are also covered. The software algorithms for paging are given, as are more detailed definitions of the terms "context," "image," and "process."

Topics include:

- Virtual Memory Management
- Division of Virtual Address Space
- A Process and Its Context
- Paging

CHAPTER 8

VIRTUAL MEMORY AND MEMORY MANAGEMENT

INTRODUCTION

The function of an operating system is to manage the system's available resources. The VAX/VMS operating system is a multiuser, multiprogramming operating system. To accommodate multiprogramming, physical memory must be shared by more than one process. Therefore, physical memory is a fundamental resource requiring allocation, deallocation, and associated management.

Virtual address space is divided into 512-byte sections called pages. The virtual page corresponds exactly in size to a block on a disk and to a page frame in physical memory. The term "page" is used interchangeably with these and is interpreted in context. The page is the basic unit of relocation and protection. Memory management utilizes page tables as the database to contain the status and location of virtual pages of processes. Each individual page of a process has associated with it an entry in an appropriate page table to describe that page. The functions of memory management are to map virtual pages into physical address space, to control the paging of those pages in active use by the process, and to provide process and interprocess memory protection.

VIRTUAL MEMORY

The memory management technique utilized by the VAX/VMS operating system is known as virtual memory. Virtual memory is the set of storage locations in both physical memory and secondary disk storage that are referenced by virtual addresses (see below). The size of virtual memory is the total of available virtual addresses. Additional features of the VAX virtual memory usage are:

1. Only a portion of the program (those pages which are being actively referenced) need reside in physical memory during execution
2. Programs (processes) are allowed to exceed the maximum amount of physical memory available

Virtual Address Space
Because of the VAX family 32-bit architecture, a longword (32 bits) is required to specify the address of any byte of memory. Therefore, VAX virtual address space consists of 2^{32} or 4.3 billion bytes. Virtual address space is divided into system and process address space, each consisting of 2^{31} bytes. The process address space is further divided into a program and control region. Virtual address space is described in Figure 13-1.

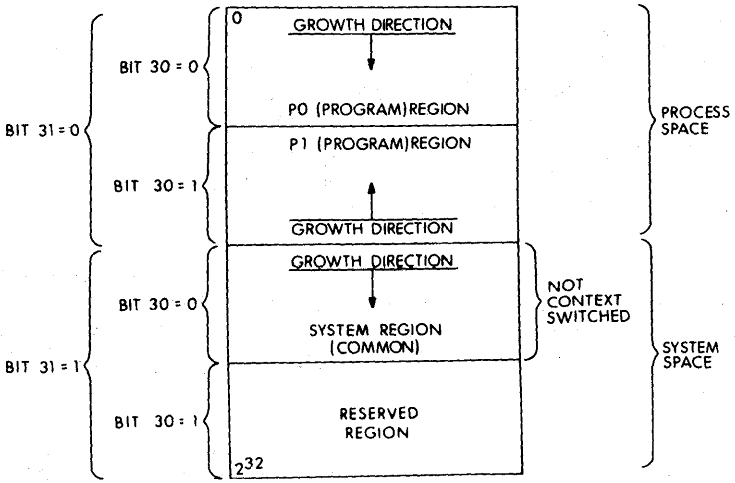


Figure 8-1 Virtual Address Space

The program region contains the native or compatibility mode image to be executed by the process, possibly the application migration executive (AME), and additional user code referenced by the image. (Some technical terms are defined below or in other chapters. See the Glossary and the Index.) The program region of a process's address space originates at virtual address location 0 and extends in the direction of increasing address locations. P0BR and P0LR are hardware registers that describe the page table containing references to each individual page in the program region. Virtual addresses in the program region are translated to physical address using the page table described by the registers P0BR and P0LR. The program region corresponds to P0 space.

NOTE

To translate from a P0 virtual address to a physical one, the P0 page table is used. The registers do not give the translation, but rather point the user to the page table.

One page table has many page table pages. Each process has one P0 page table. The registers can only describe one process' page table at a time.

The control region of a process's address space contains process-related information maintained by the system and process control structures such as the the kernel, executive, supervisor, and user stacks, and the process I/O database. The control region originates at location 2^{31} and extends toward lower addressed locations. P1BR and P1LR are hardware registers that describe the page table containing references to each individual page in the control region. The base address and length of the control region are described by the registers P1BR and P1LR respectively. The control region corresponds to P1 space.

System virtual address space occupies the first half of locations 2^{31} through 2^{32} as described in Figure 13-2. System space originates at location 2^{31} and extends toward increasing address locations. The remaining locations are reserved for future use. System space is that area of total virtual address space that is shared among all processes and contains the VAX/VMS executive and those software data structures required to control the process and to maintain the status of all physical and virtual pages within the system.

The addresses used to locate, interpret, and execute instructions are virtual addresses. As the process executes, the system translates virtual addresses to physical addresses. A virtual address consists of a 21-bit virtual page number and the number of a byte (location) within the page, as illustrated in Figure 13-3.

Bit 31 of the virtual address is used to distinguish between a process virtual address and a system virtual address. When bit 31 is set (i.e., has the value 1), the address is system virtual. Bit 30 is used in conjunction with process virtual addresses to distinguish between the program region and the control region. When bit 30 is set, the control region is referenced.

A physical address consists of a page frame number and the number of a byte within the physical page, as illustrated in Figure 13-4. The page frame number is the number of a physical page in physical memory.

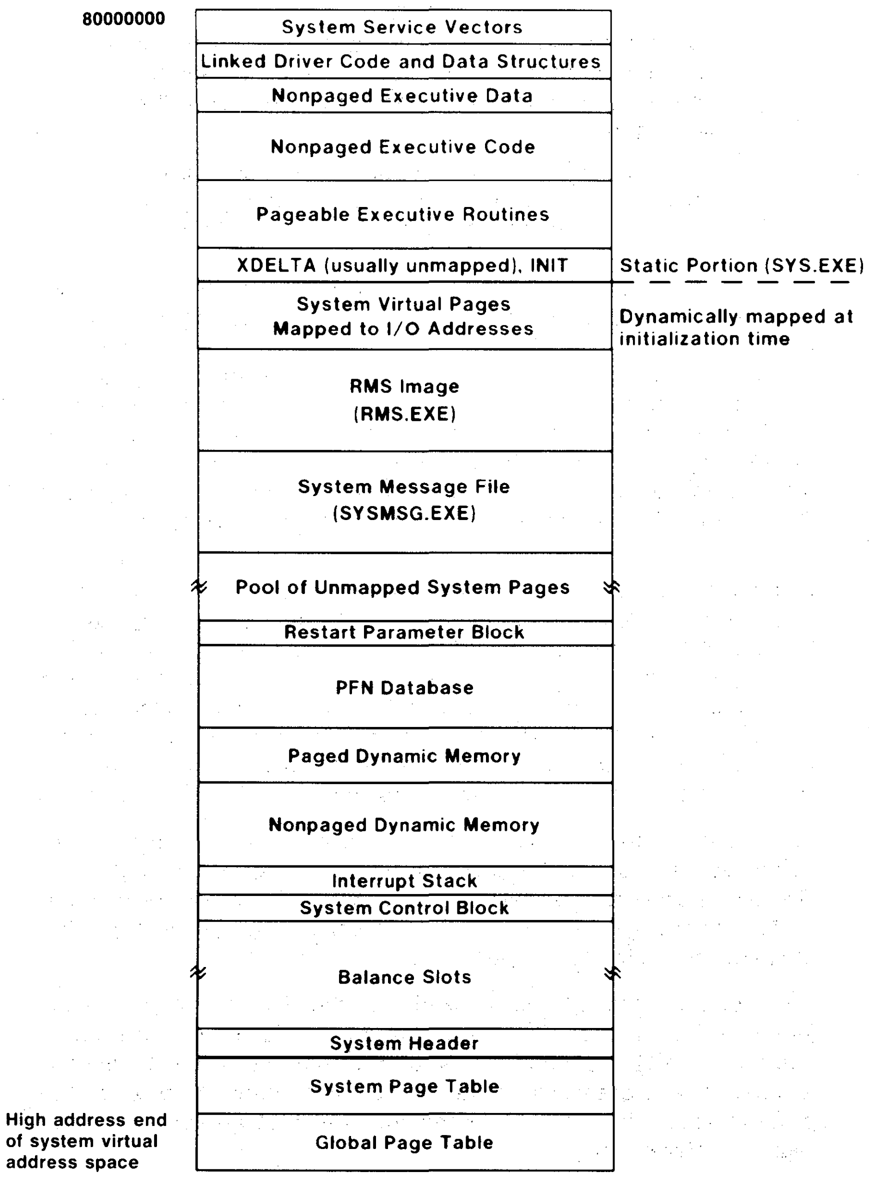


Figure 8-2 System Virtual Address Space

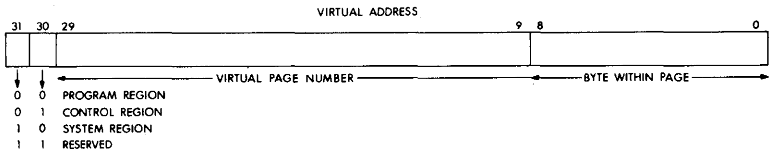


Figure 8-3 Virtual Address

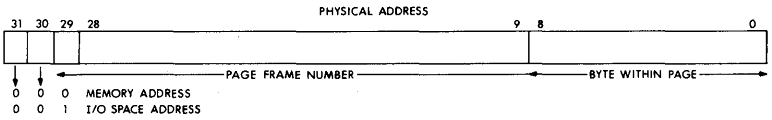


Figure 8-4 Physical Address

Dynamic Page Tables

Memory management software is responsible for creating and maintaining the mapping structure required by the processor to translate virtual addresses referenced by a process to physical memory addresses. The basic unit of mapping and protection is the page. A page is a block of 512 contiguous byte locations in physical memory on a 512-byte boundary. Within physical memory each page is unique, and no pages overlap. Virtual addresses are also grouped into 512-byte pages, where each virtual page may be mapped to a physical page or a page (block) of secondary storage. Any number of virtual pages can be mapped to the same physical page.

Unlike some systems, in which portions of physical memory are statically allocated or partitioned, the VAX system supports complete dynamic allocation of physical memory. Dynamic allocation of physical memory may result in the noncontiguous physical location of a process's pages in physical memory, but they remain virtually contiguous in the process's address space.

The VAX/VMS operating system maintains unique translation maps called page tables for each process. Process virtual address space is described in two page tables: the P0 page table corresponding to the program region and the P1 page table corresponding to the control region. Each portion of the process space is described by a virtually contiguous vector of page table entries. Process page tables reside in system virtual memory when the process is resident. Being themselves virtual pages, the page tables may be mapped to physically discontiguous areas of memory and are resident only when required. When a virtual page is in memory, the page table entry contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage. From the process page tables contained in system virtual space, it is possible to locate all process virtual pages.

System virtual space is described in a data structure called the system page table (SPT). The system page table contains one page table entry (PTE) for each page of system virtual memory. The hardware system base register (SBR) and system length register (SLR) provide the physical address and the length (in longwords) of the system page table. The system page table resides in system virtual memory, but is physically based and physically contiguous. Given the contents of the SBR and SLR, it is possible to locate all other system virtual pages.

PROCESS

A process is the basic schedulable entity in the VAX/VMS system. A process consists of a virtual address space and hardware and software contexts. The hardware context of a process is defined by values that are loaded into processor registers when the process is scheduled for execution. The hardware registers consist of the following:

1. The four registers mapping user process virtual address space P0BR, P0LR, P1BR, and P1LR
2. A set of 19 user registers (R0-R13, program counter, and user, supervisor, executive, and kernel stack pointers)
3. The processor status longword (PSL)

The VAX processor contains one set of processor registers used to maintain the hardware context of a single process. While a process is executing, its hardware context is continually being updated in the processor registers. When a process is not being executed by the VAX processor, its hardware context is stored in a software data structure called the hardware process control block (PCB). The VAX/VMS operating system maintains the hardware PCB in a software structure known as the process header (PHD). Saving the contents of the VAX processor registers into the hardware PCB of the currently executing process and then loading a new set of context from another hardware PCB is called context switching. Context switching occurs as one process after another is scheduled for execution by the VAX/VMS operating system.

The VAX/VMS operating system maintains the software context for each process. The software information regarding the process is maintained in a data structure called the software process control block (software PCB). The combined hardware and software description is referred to as the process's context.

Working Set

When a process executes, a subset of its pages resides in physical memory. This subset is referred to as the process's working set. A process's working set consists of all the pages of a process's virtual memory which are residing in physical memory and which the process can access directly without incurring a page fault. A page fault is a reference to a page not currently in the process's working set. This condition is handled automatically by the operating system as discussed under the subheading "Paging."

The working set is a dynamic characteristic of a process that has both minimum and maximum size limits. The system designates a required minimum number of pages that have to be in a process's working set, and the system manager defines the maximum number of pages allowed in any one job's working set in the user authorization file. The size of the working set determines the amount of physical memory needed to run a process, and directly affects its paging and swapping performance.

Balance Set

The collection of processes residing in physical memory at any one moment is called the balance set. During the execution of a process, conditions may occur that require the movement of the process's working set to secondary storage, thereby freeing physical memory for another process to use. This method of controlling memory use by removing processes from and adding other processes to the balance set is called swapping. The swapper utilizes three conditions to determine which processes should be swapped in and which should be swapped out:

1. Process priority
2. Process status (which processes are executable and which are not)
3. Expiration of process's balance set time quantum (process has used up assigned CPU time slice without completing and must wait for another turn)

For example, a process's working set can be written to secondary storage while the process is waiting for I/O completion on a slow device, making room for another outswapped process which can execute immediately. The working set is brought back into the balance set after I/O completion.

For more information on swapping, see Chapter 14.

PROCESS CONTROL STRUCTURES

VAX hardware defines a process by using registers and a hardware process control block (PCB). The VAX/VMS operating system, however, provides each process with additional definition that is used to control the process, its working set, and the balance set. The two most important structures that define a process are the software process control block (PBC) and the process header (PHD).

The system also provides each process with a unique name called the process identification.

Software Process Control Block

The system defines a software PCB for every process when the process is created. The software PCB is the central control mechanism for the process. It includes the following kinds of information about the process:

1. Current state of the process (executable, in one of several types of wait states, swapped out, etc.)
2. Storage address of the process if it is swapped out of memory
3. Unique identification of the process
4. Software priority of the process
5. Additional status and control information

Software PCBs for all processes reside in the system virtual address space. However, because the software PCB contains the information needed to schedule a process and retrieve a swapped process from secondary storage, it is always resident in memory.

Process Header

The system defines a process header for every process when the process is created. When a process is swapped into memory, i.e., brought into the balance set, the header for the process is placed in one of the process header slots reserved in the system virtual address space. The software PCB for each process contains the virtual address of the process's header. The number of process header slots defined for the system determines the number of processes that can be in the balance set. However, since processes are subject to outswapping, the system can maintain a greater number of PCBs than process header slots.

A process header, illustrated in Figure 13-5, contains the following information:

1. Privilege mask for the process
2. Hardware PCB
3. Indices to the working set list and the process section table portions stored lower down in the PHD
4. Accounting
5. Working set list
6. Process section table
7. P0 and P1 page tables

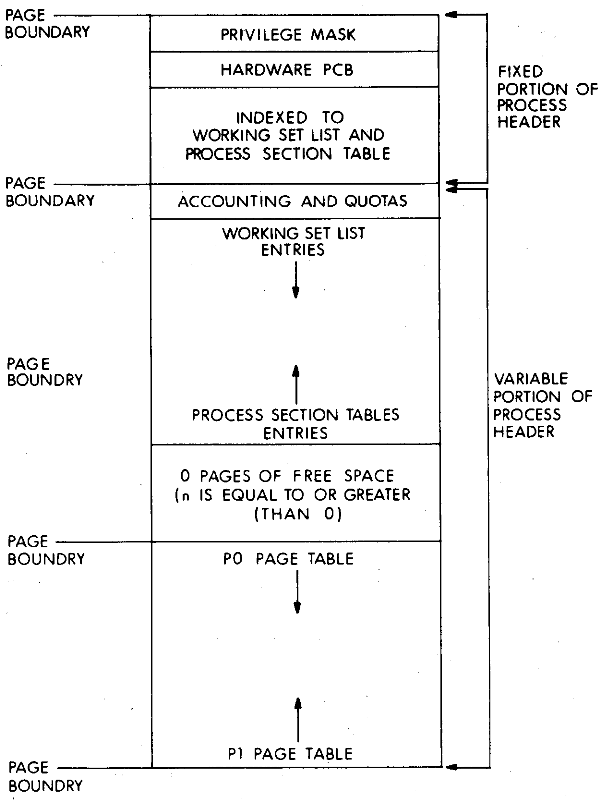

PAGE BOUNDARY
PRIVILEGE MASK
HARDWARE PCB
INDEXED TO WORKING SET LIST AND PROCESS SECTION TABLE
FIXED PORTION OF PROCESS HEADER
PAGE BOUNDARY
ACCOUNTING AND QUOTAS
WORKING SET LIST ENTRIES
PAGE BOUNDRY
PROCESS SECTION TABLES ENTRIES
VARIABLE PORTION OF PROCESS HEADER
0 PAGES OF FREE SPACE (n IS EQUAL TO OR GREATER (THAN 0)
PAGE BOUNDRY
P0 PAGE TABLE
P1 PAGE TABLE
PAGE BOUNDRY

Figure 8-5 Process Header

The working set list contains entries to describe that portion of the process's virtual address space that is resident in physical memory. This database is maintained by the pager and is also used and modified by the working set swapper. It starts at a page boundary, and expands toward higher addresses.

The process section table contains entries to describe the process-private image sections that are mapped by the process's page tables. The image activator fills in this table. The process section table starts at a page boundary and extends in the direction of the working set list, as illustrated in Figure 13-5.

The page tables contain the one entry needed to locate every virtual page of the process. The page tables are initialized by the image activator and dynamically maintained by the pager and, after inswap, by the swapper as well. The page table for the program region of the process starts on a page boundary and extends to higher addresses. The address of the page table is found using a pointer (P0BR) in the hardware PCB.

The page table for the control region of the process starts on a page boundary and extends toward lower numbered pages. As illustrated in Figure 13-5, the P1 table starts at the last page of the process header and extends in the direction of the P0 page table. The P1 page table also is addressed through a pointer (P1BR) in the hardware PCB.

A process header is in memory only when the associated process's working set is in the balance set.

IMAGE

An image consists of procedures and data that have been bound together by the linker. Binding (or linking) refers to the resolution of symbolic references between modules and the assignment of virtual address space.

An image results from the linking of one or more object modules together. It is the program entity that is executed by a process. When a user logs onto the system, the system creates a process dedicated to that user. That process executes all of the images needed to perform the user's requests. Images are referred to by file name. Examples of images are the linker, the assembler, and user programs.

The unit of virtual memory allocation associated with the image is known as the image section (isect). An isect is a group of program sections (psects) with identical attributes. For example, the psects in a given isect might have the read-only and relocatable attributes.

Image Activation

The image activator is a set of procedures that execute in the system address space to prepare an image for execution. The procedures, however, run in the context of the process requesting execution of the image.

Opening the Image File

The command interpreter passes the file name of the image to be executed to the image activator. Using defined search rules, the image activator locates the file and starts processing the image header. At this point, the image activator determines whether the image is native or compatibility mode, using information in the image file header.

If the requested image is a native mode image, the image activator sets up the process section table entries and the process page table entries in the P0 and P1 page tables. Once the process section table and the page tables are set up, the image is ready to execute.

If the image is a compatibility mode image, its name and the number of the channel on which the image file is open are saved in a known place in the process control region for the application migration executive (AME). The image activator locates the image name of the AME in the process control region and activates the AME image instead of the requested image. The AME then maps in the compatibility mode image.

Setting Up the Process Section Table

The image activator uses information produced by the linker to create process section table entries for the image. When the linker produces an image file, it places an image section descriptor (ISD) in the image header for every image section in the file. The image activator uses the ISDs to create process section table entries.

Once the image activator has read the image header and created the process section table entries for the image, it can set up the P0 and P1 page table entries for the image.

Setting Up Page Table Entries

The image activator uses the number of pages for each image section (isect) as specified in the ISDs to determine the total number of pages needed for the image.

Before an image can execute, the image activator must fill the page table entries for that image with an index to the process section table entry for the section that contains that page of the image. The process section table contains the information needed to locate that page in the image file.

Once all of the page table entries for the image are set up, the image is ready to execute.

PAGING

Paging is the action of bringing pages of an executing process into physical memory when referenced. When a process executes, all of its pages reside in virtual memory. Only the actively used pages, however, need be in physical memory. The remaining pages can reside on disk until they are needed in physical memory. In the VAX/VMS operating system, a process's pages are paged out only when the process references more pages than it is allowed to have in its working set. When the process refers to a page not in its working set, a page fault occurs. This causes the operating system's pager to read in the referenced page if it is on disk (and, optionally, other related pages), replacing the oldest removable pages as needed.

Page Table Paging

To reduce the amount of memory required to run a process, the only pages of process page tables that are required to remain in memory are those containing one or more page table entries that refer to a page frame number (i.e., the identification of a page actually in memory). Page table pages are faulted into memory by faulting a page in the page table or by faulting the page table entry itself, as happens when creating a new page with the Create Virtual Address Space system service. In either case, the page table looks like a normal data page in the process's working set list and is subject to working set replacement.

Whenever a page table entry has a page frame number placed in it, the reference count for the page table page is increased. If the page frame number is the first one in the page table, (i.e., the reference count went from 0 to 1), the pager locks the page table page in the working set list.

As each page frame number is taken out of the page table page and replaced by its backing store address, or by 0 if the page has been deleted, the reference count for that page of the page table is decreased. When the last page frame number is taken out (i.e., when the reference count goes from 1 to 0), the pager unlocks the page table page from the working set, thereby making it eligible for working set replacement.

Pager

The pager is a set of routines that execute as a result of a Translation Not Valid Fault, that is, a page fault. It is, therefore, an exception service routine. The pager runs in kernel mode in the context of the process that caused the fault. The pager's function is to make the page for which the fault occurred available in physical memory so that the process can continue execution. The page can be in the image file, in memory but not in the process's working set, or in the paging file when a fault occurs.

The pager uses the paging file to maintain modified pages from either an image or a global section.

A backing store (secondary storage) address in the paging file is assigned to a page under either of the following conditions:

1. The page was a demand-allocate zero-fill page that was not in a process section or global section
2. The page was a copy-on-reference page

Other pages maintain their original backing store addresses. These pages are the following types:

- Read-only image file pages
- Read/write process section pages
- Read/write global section pages

In order to make pages available as needed, the pager maintains and manipulates the following databases:

- Page Frame Number (PFN) database
- Free page list
- Modified page list
- Working set lists
- Page table entries

The paging philosophy implemented in the VAX/VMS operating system is called "paging against the process." Each process is allocated a maximum number of pages in its working set. A page fault to a filled working set requires that one page be removed for each page brought in.

Page Frame Number (PFN) Data Base

The page frame number (PFN) database consists of 18 bytes of information for each page of physical memory. The 18 bytes for each page are not grouped together to form a table per page; rather, the various categories of information are organized as arrays. Items within each array are indexed using physical page frame numbers (PFNs).

Every physical page has an entry in the following arrays:

- System virtual address array of longwords

- Backing store address array of longwords
- Reference count array of words
- Forward and backward link arrays of words
- Swap virtual block number array of words
- State array of bytes
- Type array of bytes

Free Page List

The free page list is a doubly linked list of physical memory pages that are available for use. Pages are linked at the end of the list and removed from the head. When a page is removed from a process's working set, the page is placed on the free page list if its reference count in the PFN database is zero and the modify bit in the PFN state byte is clear. If the modify bit is clear, the page has not been written into (altered or modified), and the disk retrieval information in the PFN database is valid.

If the process faults a page on the free page list that does contain valid data, the pager can unlink the page from the free page list and make it available to the faulting process. Thus, the free page list acts as a page cache for the most recently discarded pages in addition to being a source of available pages. Therefore, increasing the size of the free list (through a SYSGEN parameter) can minimize the number of pages that must be faulted from the disk. At the time a page is removed from the top of the free page list and filled with a new virtual page, the PFN database and the page table entries for both the old and the current virtual pages are updated.

Modified Page List

The modified page list is a doubly linked list of physical memory pages whose contents must be written to backing store before the pages can be linked onto the free page list. When a page is removed from a process's working set, the page is placed on the modified page list when its reference count in the PFN database is zero and the modify bit in the page table entry is set. When the modify bit is set (i.e., =1), it indicates that the page has been altered since it was last read into memory from either its image file or the paging file.

Just as pages can be faulted from the free page list for reuse, pages can be faulted from the modified page list. If the write has not been started, the page is unlinked from the list. If the write is in progress, i.e., the write-in-progress state in the PFN database is set, the page is no longer on the list. However, the page can still be made available to the faulting process by signaling the write completion routine that it should not place the page on the free page list. This also helps to reduce accesses to secondary storage and to speed up process execution.

Working Set List

The working set list provides the mechanism required by the pager to keep track of and limit the process's use of physical memory. It specifies the number of physical pages of memory that the process can have resident at any time. It also contains a linear list of the virtual page numbers of the resident pages.

Each working set list pointer in the fixed portion of the process header is a word containing the longword index (sometimes called an offset) from the beginning of the process header to its respective working set list entry.

The working set list not only limits the number of physical pages that a process can have in memory, it also provides the complete list of those virtual pages that are actively being used. This information is required by the balance set swapper so that it can swap the entire working set.

Page Faults For Process-Private Pages

Every virtual page of a process has an associated page table entry that represents its current state. A process-private page can be in any one of the following states:

1. Out of the process's working set and in the image file. The page either has never been faulted into memory or has been faulted into memory and out again without being modified. In either case, its backing store disk address locates it in an image file
2. Out of the process's working set and available as a demand-allocate zero-fill page. The page has never been faulted into memory. When it is, the pager supplies a physical page filled with zeros
3. Out of the process's working set and in the paging file
4. In transition. That is, in the free or modified page list or currently being read into or written from memory
5. In the process's working set

The format used for a page table entry to describe a given page indicates which state the page is in. All page table entry formats use bit 31 as the valid/invalid bit. When bit 31 is set, it indicates that the virtual page associated with that page table entry is in a physical page of memory and is in the process's working set. That is, it is an active page and has an entry in the process's working set list. A page fault occurs when reference is made to a page whose page table entry has bit 31 cleared. The page can be in any of the first four states listed above.

SHARING PAGES OF PHYSICAL MEMORY BETWEEN PROCESSES

The sharing of procedures and data among many processes is accomplished through the use of global sections. A global section becomes available for sharing as a result of one of two steps:

1. Creation of a shareable image and installation of that image as a shared known image
2. Creation of a data file and issuance of a Create and Map Section system service to make that file globally available

Known Images

Once the image is installed as a shared known image, it is available for mapping into the virtual address space of many processes. The installation procedure results in the creation of the database required for the sharing of global sections, but only if it is /SHARE specified when installed. Not all known files are shareable.

Global Section Database

Sharing sections in the process address space requires the creation of a global section database. The global section database is created as a result of a Create and Map Section system service issued when a shareable image is installed as a shared known image or issued by a process to create a global data section. The database consists of the following data structures:

- Global section descriptor
- Global section table
- Global page table

Global Section Descriptor

The global section descriptor (GSD) provides the naming and access protection mechanisms for a global section. One GSD is defined for each global section. Dynamic memory is allocated for GSDs. There are two doubly linked lists of GSDs in the system: one for system-wide global sections and one for group global sections.

The owner user identification code (UIC) is the UIC of the creator of the global section. Protection for the section is specified when it is created.

The global section table index is an index to this section's global section table entry.

During the installation of a shared known image (created by INSTALL), the section identification and name are taken from the image section descriptor (ISD) for the section. The ISD is produced by the linker and placed in the header of the linkable image file. The image activator uses this information when it prepares to execute an executable image that is bound to a shareable image.

Global Section Table

The global section table is a parallel structure to the process section table. It is the section table in the system process header. It contains one entry for each global section. A global section table entry describes the disk area that the corresponding global section occupies. The global section table index is an offset to the associated global section table entry.

Global Page Table

The global page table is the master page table for the pages of a global section. One global page table entry is required for each page in the global section. The global page table entries for a section must be contiguous. Global page table entries have formats that are similar to those of process page table entries. The pager manages both types of page table entries.

The initial format of a page table entry for a global section is the same as the initial format for a page of a private section. That is, both contain a section table index.

Read/write global section pages are written back into the image or data file; they are not placed in the paging file. Writing them back to the image or data file provides cooperating processes with a common read/write area. Such cooperating processes must synchronize their access to read/write shared files. None is provided automatically.

The global section table index contained in the page table entry is the offset into the global section table for this section. The global section table entry is used in the same way as a process section table entry to locate the page in the section on disk.

Image Activation

A process can map to global sections in either of two ways:

1. By running an executable image that hasn't been installed as a shared known image or by running an image that is linked against a shareable image that has been installed as a shared known image
2. By issuing a request for the Map Global Section system service

If the first method is used, the image activator takes the steps needed to map the global section in the process's address space. When it encounters an image section descriptor (ISD) in the image file referring to a global section, the image activator calls the Map Global Section system service to scan global section descriptors for the section name specified in the ISD. If the service locates the specified section name, it compares version information in the global section descriptor for the section with version information in the ISD for the section in the executable image file.

If the globally available version of the section is appropriate, the Map Global Section system service maps the global section into the process's address space. The result is that a specified range of page table entries in the process space is filled with indirect pointers to the corresponding global page table entry for the global section.

If the second method is used, the process itself requests the mapping, rather than having the image activator request the mapping for it. The result of the Map Global Section system service is identical in either case. The second method is used to map a data file that has been made a global section.

A process page table entry that contains a global page table index is a pointer to the global page table entry and its associated database that provides central control of the global section.

Page Faults

When a page fault occurs in a process and the process page table entry for the page contains a global page table index, the pager uses the content of the global page table entry pointed to by the process page table entry. The global page table entry provides the pager with the information needed to determine what action is necessary to make the process page table entry valid. The result of the action is a process page table entry that contains the page frame number that is the physical address of that page of the global section.

When a fault occurs in a process for a page of a global section, the page can be in any one of the following states:

1. In a section of a file on disk
2. In memory but not in the working set of the faulting process
3. In the free page list or modified page list
4. In the page file
5. Doesn't exist yet, as it is a demand-zero page

SWAPPING

The swapper is the process responsible for moving entire working sets between main memory and secondary storage. The swapper process serves two major functions:

1. Process scheduling
2. Process creation

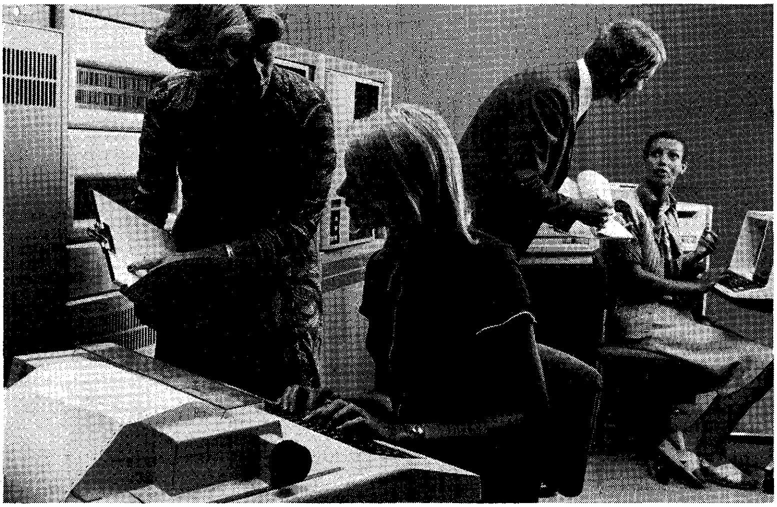
Process scheduling and the swapper are discussed in Chapter 14, Process Scheduling and Swapping.

PAGING IN SYSTEM SPACE

A considerable amount of code that can be paged exists in the system address space, including many system services. Paging in system space is essentially the same as paging in process space. Data structures parallel to those used for a process are used for system space to provide the information needed to page system space. The following structures are defined:

- System header (parallel structure to process header)
- System working set list
- System section table
- System page table (parallel to a process's P0 page table)

Working set replacement in system space functions in the same manner as in a process. That is, pageable system pages are paged against each other.



CHAPTER OVERVIEW

Central to the effectiveness of multiuser computers is the algorithm that controls swapping of processes and the allocation of CPU time. The VAX/VMS operating system provides an advanced swapping technique that reduces thrashing and helps minimize overload.

Scheduling is event-driven, pre-emptive, and priority-controlled. The upper sixteen priority levels are usually reserved for realtime processes. In order to balance the system load, the system modifies the priorities of processes in the lower sixteen levels. In addition, time quanta insure a rotation among computationally intensive processes of the same priority. This chapter examines swapping and scheduling.

Topics are:

- Scheduling
- Swapping
- Priorities

CHAPTER 9

PROCESS SCHEDULING AND SWAPPING

INTRODUCTION

The VAX/VMS scheduler performs normal and realtime process scheduling based upon the priority of the executable processes in the balance set. A normal process is also referred to as a timeshared or background process while a realtime process is sometimes referred to as time-critical.

The VAX/VMS operating system defines 32 distinct levels of software priority for the purpose of scheduling. Priorities range numerically from 0-31, where 31 represents the highest software priority. The operating system allocates priorities 0-15 to the scheduling of normal processes while priorities 16-31 are dedicated to the scheduling of realtime processes. Realtime processes are scheduled strictly by priority; when a higher priority process is ready to execute, it pre-empts the process currently running. Normal processes, on the other hand, are scheduled using a modified pre-emptive algorithm to achieve maximum overlap of computational and I/O activities.

As part of a process's total context, its software Process Control Block (PCB) maintains a link to the current state queue defining the process's status within the system. A state queue is a list of all those processes currently residing in a particular processing state. A single state queue exists for every state or condition in which a process may reside. Examples of possible process states are: computable, local event flag wait, hibernation, etc.

Regardless of which state queue a process is in, the process owns of a collection of pages that is referred to as its working set. The swapper is the process responsible for moving entire working sets between main memory and secondary storage. Moving a process from main memory to secondary storage is called outswapping; moving a process from a secondary storage device to main memory is called inswapping.

The swapping of processes is necessary for two reasons:

- To replace lower priority or nonexecutable resident processes with higher priority executable processes
- To keep the scheduler supplied with executable processes in configurations that do not provide sufficient main memory to contain all processes's working sets

SCHEDULING

Realtime processes take precedence over background processes in the queue for execution because they are of higher priority. The VAX/VMS scheduler performs process scheduling that takes into account the following variables:

1. The process priority
2. The occurrence of system events and resulting process state transitions
3. The expiration of in-memory time allowed to a non-realtime process. This is called quantum overflow

The process selected to execute is always the process with the highest priority in the executable resident state queue.

System events are occurrences that cause the state of one or more processes in the system to change. The scheduler reflects the change by removing the process's PCB from one state queue and placing it in the current state queue. An executing process can cause a system event by putting itself in a wait state, or it can cause a system event for another process. In addition, system components like the swapper and the timer can cause system events. Regardless of the source, all system events are reported to the scheduler.

System events can be synchronous with the process's execution (e.g., a wait request), or they can be asynchronous (e.g., an I/O completion event).

Process States

The state of a process is the condition of the process at a given instant. For example, a process can be in a hibernate state or a local event flag wait state. The possible states of a process are mutually exclusive. A process moves from one state to another as a result of system events. The state number of a process is defined by a field in the software PCB. Each state has a queue of processes that are in that state. The processes's software PCBs are linked into the appropriate state queue.

Some conditions have two associated state queues: one for resident processes and the other for nonresident processes. Others mix both resident and nonresident processes in the same queue. The separation into two queues is to optimize queue searching. In all cases, the residence of a process is indicated by a status bit in the PCB.

State Queue Headers

Each of the state queues in which a process can be linked is a standard linked circular queue that is suitable for use in INSQUE and

REMQUE* instructions. The header for all queues is a quadword that locates the head and the tail of the queue. If the queue is empty, the header points to itself. The header structure for wait state queues differs from that for executable process state queues in that the latter uses a subqueue structure. Figure 14-1 describes the general state queue header.

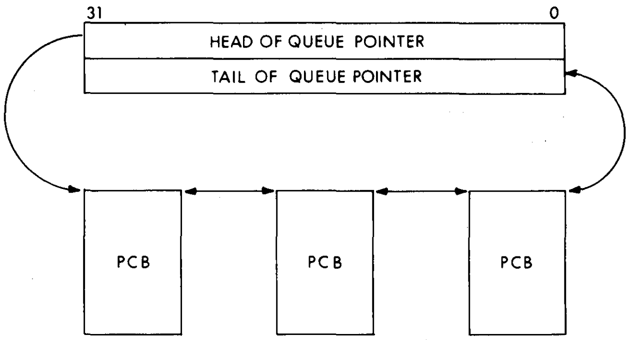


Figure 9-1 State Queue Header

Wait State Queue Headers

Wait queue headers have a count of PCBs associated with the queue in addition to the standard head/tail quadword. Figure 14-2 illustrates a wait queue header.

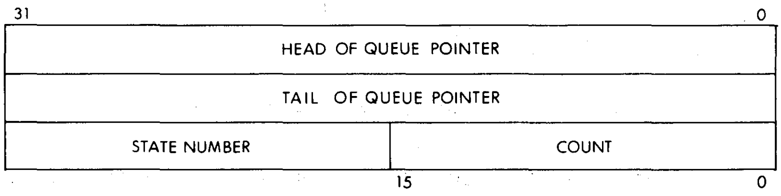


Figure 9-2 Wait Queue Header

* The **VAX-11 Architecture Handbook** gives detailed descriptions and examples of many VAX/VMS instructions.

Executable Process State Queues

The state queues for executable processes within and outside of the balance set are divided into 32 subqueues, providing one subqueue for each priority level. The state of a process and its priority provide the scheduler with the information needed to determine the subqueue for the process.

Each subqueue has a header that contains the head/tail quadword. Subqueue headers do not contain a count of PCBs linked into the queue. Instead, an array called the summary longword is maintained for the executable process state subqueues. Each bit in the longword corresponds to a subqueue, and if a bit is set, the corresponding subqueue contains entries. Refer to Figure 14-3 for an example of an executable process state queue.

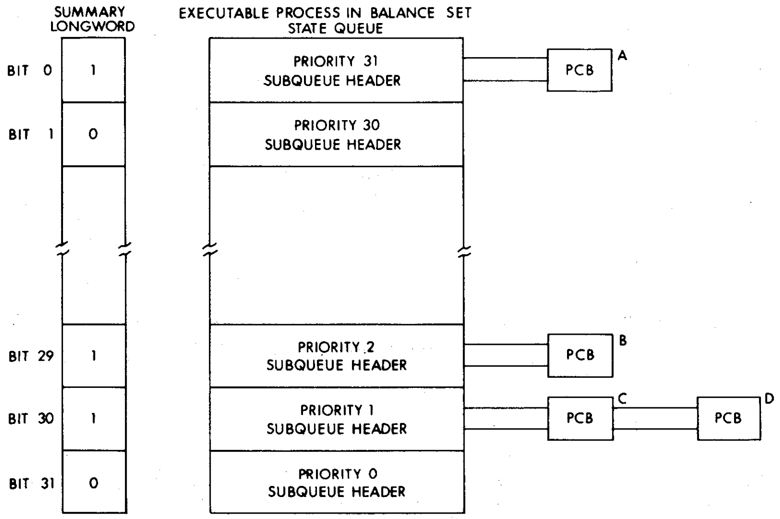

SUMMARY LONGWORD
EXECUTABLE PROCESS IN BALANCE SET STATE QUEUE
BIT 0
1
BIT 1
0
BIT 29
1
BIT 30
1
BIT 31
0
PRIORITY 31 SUBQUEUE HEADER
PRIORITY 30 SUBQUEUE HEADER
PRIORITY 2 SUBQUEUE HEADER
PRIORITY 1 SUBQUEUE HEADER
PRIORITY 0 SUBQUEUE HEADER
PCB
A
PCB
B
PCB
C
PCB
D

Figure 9-3 Executable Process State Queue

Processes are selected from the state queue in order of priority. Higher priority processes receive attention first. Processes are selected on a first-in/first-out basis within a priority subqueue. Referring to Figure 6-3, processes would be selected for execution in the order A, B, C, and then D. Processes are selected as if they were in one queue; the subqueue structure is used to simplify queue searching. That is, if the summary bit for a priority subqueue is clear, the scheduler does not need to consider that queue. A single instruction is required to locate the first non-empty subqueue, thereby locating the highest priority process.

Process State Transition

Transitions from one process state to another occur as the result of system events reported to the scheduler. The process state transition cycle is illustrated in Figure 14-4.

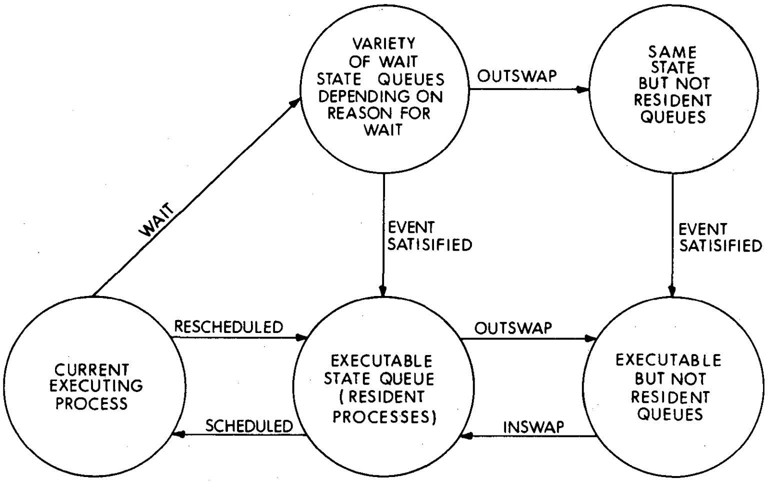

VARIETY OF WAIT STATE QUEUES DEPENDING ON REASON FOR WAIT
OUTSWAP
SAME STATE BUT NOT RESIDENT QUEUES
WAIT
EVENT SATISIFIED
EVENT SATISIFIED
RESCHEDULED
OUTSWAP
CURRENT EXECUTING PROCESS
EXECUTABLE STATE QUEUE (RESIDENT PROCESSES)
EXECUTABLE BUT NOT RESIDENT QUEUES
SCHEDULED
INSWAP

Figure 9-4 Process State Transition Cycle

When the current executing process ceases execution, it will enter one of the following states, depending upon the system event that caused it to stop:

1. Executable state queue in the balance set as the result of a reschedule event
2. A wait queue as a result of a suspend, hibernate, wait for local event flag (LEF), wait for common event flag (CEF), page fault wait (PFW), collided page wait (COLPG), or miscellaneous wait (MWAIT)

A process that is in the balance set and in any of the wait queues can make the transition to either of the following states:

1. Executable and in the balance set as a result of a system event that satisfied the wait condition. For example, if a process is waiting for a local event flag and that flag becomes set, it enters the executable state queue
2. In the same wait state but swapped out of the balance set. For example, in the case of suspend, hibernate, and wait for local event flag, making the transition from a process in the balance set to one out of the balance set causes the process to be placed in another wait queue.

   In the case of wait for common event flag, page fault wait, collided page wait, and miscellaneous wait, processes that are in the balance set and those that are out of the balance set are placed in the same queue

Asynchronous System Trap (AST) events are significant for processes in a variety of states, including hibernating and outswapped, local event flag and outswapped, page fault wait, common event flag wait, free page wait, and collided page wait. For a process in one of these states, issuance of an AST to the process or a request to delete the process results in the process's being placed in the executable state but not necessarily in the balance set.

A process that is out of the balance set and in a wait queue can make the transition only to the state of being executable and out of the balance set. It is placed in the appropriate subqueue according to its priority, as illustrated in Figure 14-3.

A process that is executable and out of the balance set can make the transition only to the state of being executable and in the balance set. Again, it is placed in a subqueue according to its priority. Once a process is executable and in the balance set, it is selected to execute according to its priority as a result of a system event indicating the need to reschedule.

When a process is created, it enters the nonresident executable state. When a delete request is issued for a process, the process is marked for deletion and placed in either the resident executable state queue or the nonresident executable state queue. The process executes termination procedures and is then removed from the system.

Dispatching A Process For Execution

Dispatching an executable process to the processor involves minimal decision making. The selected process is always the one at the head of the highest priority subqueue of the executable process in the balance set state queue. The real scheduling decisions are made as a result of those system events that cause the state transitions which make processes executable.

When a process is pre-empted to dispatch a process of higher priority, the pre-empted process is placed at the end of the proper priority subqueue. Placing it at the end forces a rotation of processes within a priority. The result is that available processor time is distributed more evenly among all processes of the same priority.

The interval between pre-emptions is random. Intervals are determined by the occurrence of system events. Quantum keeping and other timer events provide a minimum level of event activity. In practice, the average interval between events is determined by the duration of the typical I/O operation.

Placing a process at the end of a priority queue does not necessarily increase the likelihood that the process will leave the balance set. However, a process in the balance set has a significantly better chance of being executed than a process of the same priority that is not in the balance set.

Quantum Control

Every process, regardless of its priority, is assigned an execution time quantum that is maintained in the process header. The quantum serves two purposes:

- It attempts to provide a minimum amount of time in which the process can perform useful work before it is swapped out of the balance set
- It enforces a coarse rotation interval for compute-bound processes with a priority less than 16

Realtime processes are immune to quantum-end events.

Note that the quantum is a memory occupancy quantum, not a pure compute quantum.

A process can be pre-empted many times before it has received its full quantum. However, a process remains in the balance set until it completes its first quantum or until a nonresident higher priority process requires service, or until the process enters a wait state.

When a process is swapped into the balance set, its quantum is initialized. The process status flag in the software process control block (PCB) is set to indicate that the first quantum is in progress. If the quantum expires (i.e., reaches zero), the interrupt timer interrupt routine triggers a software level interrupt. A quantum-end event causes the scheduler to perform the following operations:

1. Set the current priority of the process one unit closer to its base priority if it is a normal process
2. Clear the first quantum flag
3. Reset the quantum value
4. Trigger a rescheduling interrupt

Each time a process executes a wait request (e.g., to wait for I/O completion), a fixed amount is added to the negative quantum value, making it that much closer to expiration. If this wait time addition causes the quantum to be satisfied, the first quantum flag is cleared and the quantum counter is reinitialized. Remember, the quantum is a memory occupancy quantum rather than a pure compute quantum.

Rescheduling Interrupts

The rescheduling interrupt is triggered when either of the following two conditions exists:

1. A process making the transition to the resident executable state has a higher priority than the current process
2. The timer detects quantum expiration for the current process

Rescheduling is requested by triggering the software-controlled Interrupt Priority Level (IPL) 3 interrupt. As a result of this interrupt, the state of the currently executing process is saved and the process is placed at the end of the proper compute queue. When the current process is placed into a wait state, the highest priority computable process is selected and placed into execution.

Scheduling Of Processes

Each process has a base priority assigned to it when it is created. The priority of a realtime process remains unaltered by the system during the process's execution. However, a normal process is subject to having the scheduler alter its priority during the course of its execution.

The scheduler uses a modified pre-emptive priority algorithm for normal processes. The algorithm floats the priority according to the process's recent execution history. Each process has a current priority in addition to its base priority. The scheduler dynamically changes the current priority as the process executes; however, the current priority is never less than the base priority.

Scheduling according to strict priority for realtime processes and using a modified priority for other processes allow the scheduler to achieve maximum overlap of compute and I/O activities while still remaining responsive to high-priority realtime requests. Figure 14-5 illustrates process priority scheduling.

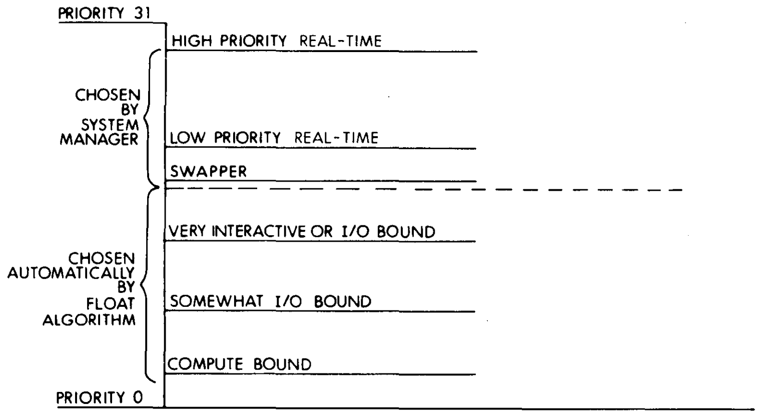


Figure 9-5 Process Scheduling

Priority Increments

The scheduler uses priority increments to modify the priority of a normal process. Each system event has an assigned priority increment that is a characteristic of the cause of the event. If the event causes a state change to an executable state for the process, the scheduler adds the increment to the base priority; the result becomes the current priority. The only restriction is that the current priority cannot be raised to a time-critical value, that is, to priorities 16 through 31.

When a wait condition is satisfied for a normal process, the scheduler increases the priority of the process in accordance with the priority increment of the satisfied condition. When a process is scheduled for execution, the scheduler decreases the process's current priority in the PCB by one unit. When the process is stopped, it is placed at the end of the next lower queue, thereby decreasing its priority. Thus, a process's priority is increased after a wait and is decreased each time it executes, as illustrated in Figure 14-6. A process's current priority is never decreased to a value below its base priority or increased above a priority of 15. A realtime process's priority is never modified.

The decrease of priority as a consequence of continued execution yields preferential treatment to processes that require only brief intervals of execution between the time that one wait condition is satisfied and the next is established. Compute-bound processes quickly fall to

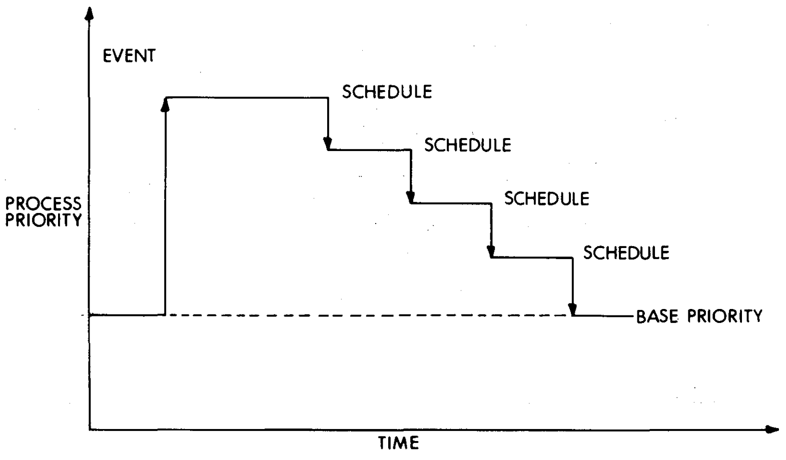


Figure 9-6 Priority Modification

their base priorities where they can be interrupted by more event-driven (I/O-bound) processes.

Priority increments are given for the following types of system events (ordered from greatest to smallest):

- Terminal read completion
- Terminal write completion and other buffered I/O
- Direct I/O (e.g., disk or magtape) completion and WAKE, common event flag wait, etc.

This gives treatment of processes with equal base priority in the following order of preference:

- Response to terminal input
- Terminal display
- File I/O and other interaction
- Compute bound

SWAPPING

Swapping is accomplished by a swapper process. All of its code and data areas are contained in system space.

The swapper performs the following functions:

1. Balancing the available page count
2. Modified page writing
3. Swap scheduling

4. Outswapping
5. Inswapping
6. Process creation

Although the functions performed by the swapper are essential to system operation, the swapper is a normally scheduled process to permit the assignment of an appropriate priority. The swapper priority is 16. That priority is the lowest of all realtime processes and higher than all normal processes. Process creation is discussed in Chapter 13, Virtual Memory and Memory Management.

Balancing the Available Page Count

The system maintains a number of physical pages that are not part of any process's working set and that are available for use by a user's process. The swapper utilizes these available pages when it brings a process's working set into memory and releases them when it swaps a process's working set to secondary storage. Likewise, memory management uses these pages to fault a virtual page of a process into memory and releases them when pages are removed from the process's working set.

Memory management maintains two lists of available pages: the free page list and the modified page list. Although modified pages are not immediately available for use, they become free pages after being written to backing storage. The modified page is written to backing storage only when that page is required as a free page. A page either list is referred to as a page in transition.

The number of free pages has a significant influence on system performance when a number of processes are actively paging. Therefore, the swapper attempts to keep the number of free pages within a specific range. The range is determined by the following two SYSGEN parameters:

- A desired number of free pages
- The lowest acceptable number of free pages

When the number of free pages falls below the lower limit, the swapper is initiated to balance the count. The swapper performs page count balancing by outswapping the process which, according to an algorithm, is the most desirable to outswap (see below for the outswap algorithm). The swapper also writes out modified pages.

The number of pages can fall below the lower limit for the following reasons:

1. A process that is resident acquired additional physical pages
2. A process was inswapped
3. A global section is deleted

Modified Page Writing
Modified pages are placed on the modified page list to be processed by the swapper and written to their backing storage address. After the backing storage copy of the page has been updated, the page is placed on the free list.

The writing of modified pages is not initiated immediately when a page is first placed on the modified page list. Rather, the swapper begins writing pages from the list when any of the following events occurs:

1. Adding a page to the list causes it to exceed a threshold size
2. The free list falls below its low limit
3. Space is needed for an inswap candidate

Deferring the writing of modified pages has two benefits:

1. Modified pages may be written in clusters, increasing the effective disk throughput
2. Modified pages may be faulted back into a process' working set, eliminating the need to write them altogether

Swap Scheduling Philosophy
Swapping normally is motivated by the need to inswap a process that would be executable if its working set were moved from secondary storage into main memory. The function of swap scheduling is to determine the highest priority process in the nonresident executable state and obtain sufficient memory to contain that process.

The needed memory is obtained by acquiring excess free pages. The number of excess free pages is determined by subtracting the desired number of free pages from the actual number of free pages. If the result is a sufficient number of pages, the nonresident process is swapped into memory. If the result is an insufficient number of pages for the process to be inswapped, additional pages are acquired by outswapping suitable processes or by writing modified pages. The pages released by outswapping are added to the count of free pages.

An executable resident process is not outswapped to acquire memory for a normal process unless it has completed its first quantum. The intent is to ensure that some useful execution occurs for a process once the inswap investment has been made.

Each time a process is swapped into memory, the swapper balances the available page count.

A nonresident process with a working set that currently cannot fit into available memory is not bypassed for a smaller process of lower priority.

Swap Scheduling Algorithm
The procedure for deciding to initiate an inswap is divided into two phases:

1. Inswap scheduling—the selection of the highest priority inswap candidate
2. Outswap scheduling—the selection of processes to be removed from main memory to enable the desired inswap to occur

Both phases are repeated each time a resident process is outswapped to permit changes that affect the choice in inswap and outswap candidates to be recognized as soor as possible.

Inswap Scheduling
Processes are selected for inswapping by choosing the highest priority process in the nonresident executable state queues. When the inswap is complete, the process is placed in one of the resident executable state queues.

Each time it is wakened from its normal hibernation state, the swapper process attempts to find an inswap candidate. The swapper is wakened after any of the following events:

1. A process is deleted
2. The free page list becomes too big or too small
3. The modified page list becomes too big
4. One second passes
5. A process is added to the nonresident executable state
6. A resident process is placed into a wait (only, however, if the process has no outstanding direct I/O and some outstanding buffered I/O)

Only the addition of a process to the nonresident executable state can alter the choice of an inswap candidate; the remaining conditions increase the availability of memory or outswap candidates.

If no swap is currently in process and an inswap candidate exists, the swapper is awakened to attempt the inswap, provided that it can obtain sufficient memory for the process.

Outswap Scheduling
Most of the swap scheduling effort involves obtaining memory required for the inswap candidate or balancing the free page count.

Occasionally, the number of excess free pages is sufficient to satisfy the inswap memory requirement. Normally, one or more resident processes must be outswapped to obtain the required memory.

The memory requirement to be satisfied by outswap scheduling has two components:

1. That required to reach the desired number of available pages
2. That required to contain the inswap candidate, i.e., the sum of private and global page counts for the candidate

Before attempting to obtain memory for a desired inswap candidate, the swapper adjusts the number of free pages, if necessary. The most suitable outswap candidate processes are outswapped until the number of available pages is greater than or equal to the desired number of pages required for the inswap. Once the count of available pages is balanced, the swapper attempts to obtain memory for an inswap candidate.

To select an outswap candidate, the outswap schedule checks a list of process states in a fixed order. The scheduler passes down the list until a candidate is found. That process is then outswapped.

Some states have constraints, others do not. For example, a process in its initial quantum is disqualified as an outswap candidate.

State queues that contain resident processes are examined for possible outswap candidates in the following order:

1. Suspended (SUSP)

   Local event flag wait with direct I/O count equal to zero (LEF)

   Hibernating (HIB)

   Common event flag wait with direct I/O count equal to zero (CEF)

   Mutex wait (MWAIT)

   Processes in the above wait states are considered to be outswap candidates regardless of their priority relative to that of the inswap candidate.

   Processes with a nonzero direct I/O count have a higher probability of their event flag wait being satisfied quickly.
2. Free page wait (FPG)

   Collided page wait (COLPG)

A process in one of the above states is outswapped only if the inswap candidate is of equal or greater priority.

3. Common event flag wait with nonzero direct I/O count (CEF)

   Local event flag with a nonzero direct I/O count (LEF)
   Page fault wait (PFW)

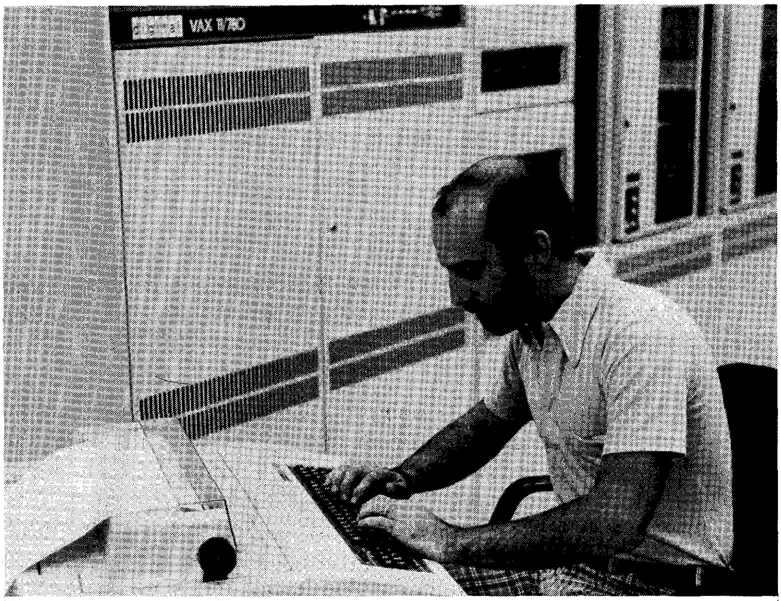
   Executable (COM)

   The above state queues contain the processes most likely to benefit from balance set residency. Both priority and the quantum flag are observed. The quantum flag indicates that the first quantum is in progress.

If an available page deficit is being corrected, the outswap is performed, and the scheduling procedure is repeated. Otherwise, the search for outswap candidates continues until the page count is balanced or all eligible outswap candidates have been examined. The most suitable outswap process is outswapped. The combined inswap/outswap scheduling operations are repeated. Eventually enough memory becomes available to perform the desired inswap.

Process Creation
The swapper performs a major portion of the process creation function by making copies of a predefined shell process, which provides the initial context and virtual address space for a process. The shell process is swapped into memory to create the process initially.

All processes that are swapped out of memory exist in a swap file as a swap image. The swap image of the shell process exists as part of the executive disk image. Using a shell process for process creation requires very little specific code because much of the normal swapping mechanism is used. However, it allows any degree of complexity for the shell process.



CHAPTER OVERVIEW

Dealing with exception, exit, and asychnronous conditions and events requires sophisticated software mechanisms such as those incorporated into the VAX/VMS operating system. The goal of condition handling is the efficient handling of conditions and events without shutting down the system or interfering with other processes is the goal of condition handlers and traps. In this chapter the VAX/VMS solution to such goals is examined.

Topics are:

- Condition Handlers
- Exit Handlers
- Asynchronous System Traps

CHAPTER 10
SPECIAL EVENT HANDLING

INTRODUCTION

During the execution of an image, both expected and unexpected conditions, called **exceptions**, can occur. An exception is any event that is detected by the hardware or software, and which interrupts the execution of the image. For example, arithmetic overflow or underflow and reserved opcode or operand faults are, for example, exceptions.

Condition handlers and exit handlers allow a process to respond synchronously to unexpected or expected conditions.

Asynchronous System Traps (ASTs), on the other hand, are interrupts (or at least reactions to an interrupt). Condition handlers are used to manage hardware-generated exceptions and software-generated signals, while exit handlers are used to clean up local databases during the termination of an image's execution.

Hardware generated exception conditions represent error conditions and must be corrected if program execution is to continue. Some software routines may generate exception conditions; these may be warning or error conditions. Software exceptions may also be caused when an error or severe error status is returned from a call to a system service.

CONDITION HANDLERS

A condition handler is a procedure that is executed in response to a hardware- or software-detected exception condition. Hardware-detected conditions cause the hardware to vector to a kernel mode routine that is responsible for interpreting the condition and dispatching control to the proper condition handler. When a software-detected condition occurs, the software signals the condition by calling a library procedure that is responsible for dispatching the condition to the proper condition handler.

Both hardware- and software-detected exceptions occur synchronously with the execution of a process. That is, they occur as the result of the execution of a specific instruction sequence; if that sequence were repeated, the same exception would occur again. Examples of hardware-detected exceptions include reserved operands, arithmetic traps, and access violations. Examples of conditions that result in the signaling of software-detected exceptions are an argument value that is out of range and the passing of an invalid argument to a called subroutine that does not return a status value, e.g., passing a negative number to a square root routine.

The VAX/VMS operating system provides two methods for specifying condition handlers:

- Specifying the address of a condition handler in the first longword of the procedure call frame
- Establishing exception vectors with the Set Exception Vector system service

The first method is the most common way to specify a condition handler; the second method—the Set Exception Vector system service—allows the specification of addresses for a primary and a secondary exception vector. There is also a last chance handler that is called after all stack handlers have been called. The exception vectors are used primarily for debuggers or program monitors.

If an exception occurs, and no user-specified condition handler exists, the default condition handler established by the command language interpreter takes control; it issues a descriptive message and optionally performs an exit on behalf of the image that incurred the exception, depending on whether a warning condition or error occurred.

Exception Dispatching

When a hardware-detected exception condition occurs within a process, the hardware vectors to a kernel mode routine after pushing PSL, PC, and arguments, if any, on the kernel stack. The actual number of arguments pushed depends on the type of exception. The kernel mode routine that gains control is called the exception dispatcher and is responsible for dispatching the exception to the proper condition handler. To locate a condition handler, the dispatcher examines only the stack and vectors for the access mode in which the exception occurred.

When a software-detected exception condition occurs, the detecting software signals the occurrence of the condition by constructing an appropriate argument list and calling a library procedure to perform the signal dispatching. The search sequence for dispatching conditions is the same whether the condition is detected by software or hardware.

SEARCHING FOR A CONDITION HANDLER

When an exception occurs, the primary exception vector and then the secondary exception vector are examined to determine if either contains the address of a handler. If either is nonzero, a condition handler has been found.

If both are zero and the exception was hardware-detected, the call stack for the appropriate access mode must be searched for a condition handler. The mode is the one at which the exception occurred or was signaled.

The call stack is searched by following the saved frame pointer (FP) register images backward through the stack. At the time of the exception, the FP points to the current call frame. Because the condition handler address is the first longword in a call frame, the FP also points to the longword that can specify a condition handler. Each call frame contains a saved copy of the previous call frame FP. Thus it is possible to trace backward through the call frames, examining the first longword in each frame to determine whether it is nonzero.

The search back through the call stack is terminated by finding a condition handler or detecting a previous frame pointer that is zero. The search of the call stack is performed at the access mode at which the exception or signal occurred. The stack frames are accessed, and if a frame is inaccessible, an exception occurs. The exception or signal dispatcher declares its own condition handler for access violations and processes any exceptions it causes.

FATAL ERRORS AND SYSTEM CRASHES

If the access mode incurring a hardware exception was kernel or executive and any of the following conditions exist, the system is shut down in a controlled fashion:

1. No condition handler could be found
2. All condition handlers that were found resignaled the condition
3. An access violation was detected while searching the stack

Not finding a condition handler for kernel or executive mode is considered a fatal system error. If the access mode was either supervisor or user, an error message is issued and an Exit system service is executed on behalf of the process at the access mode of the exception. The exit argument supplied to the system service is "absence of condition handler."

Argument List Passed To The Handler

If a condition handler is found in the primary or secondary vector or on the call stack, a complete argument list is constructed in preparation for reflecting the exception to the proper handler. The argument list consists of two addresses that point to longword arrays.

The first argument is an array containing the signal arguments and the second is an array containing the mechanism arguments. The signal array contains values describing the condition. The mechanism array contains the condition context. The first longword of each array specifies the number of arguments in the array. The depth parameter defines the frame number in which the condition handler was found.

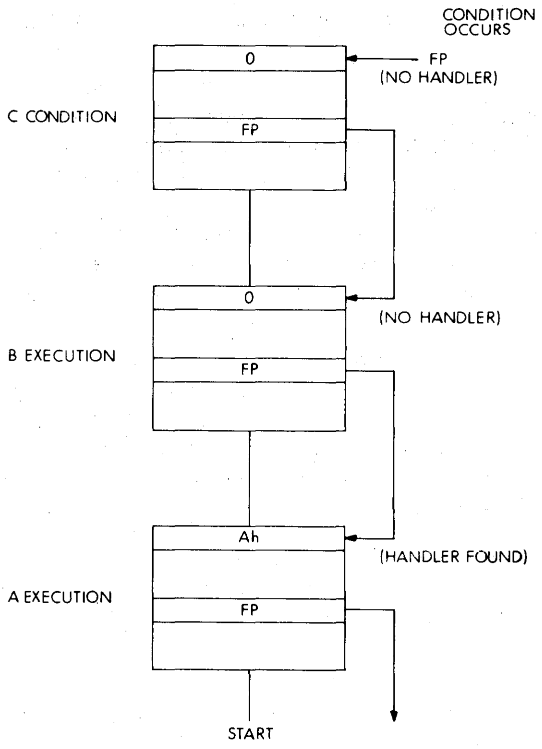


Figure 10-1a Stack Search of Multiple Conditions

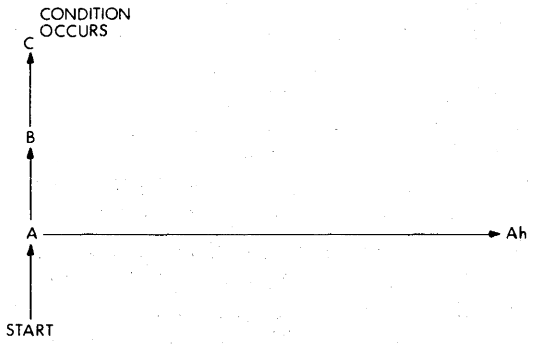


Figure 10-1b Conceptual Flow Diagram of Stack

Condition handlers are called using the standard procedure call conventions. They execute at the access mode at which the exception occurred.

Condition Handler Actions

Once entered, a condition handler has three alternatives:

1. Fix the problem and return a status value indicating that execution is to be continued at the point of the exception
2. Determine that it does not handle the exception and return a status value indicating that the exception is to be resignaled
3. Call the Unwind Call Stack system service to unwind the call stack to a specific frame

EXIT HANDLERS

Exit handlers are procedures that are called whenever an image requests an Exit system service from user, supervisor, or executive mode. Exit handlers allow a procedure that is not on the call stack to gain control and clean up procedure-specific databases.

Exit handlers are specified using the Declare Exit Handler system service. This service accepts as an argument the address of a termination handler control block. The termination handler control block minimally contains: a longword used to link termination handler control blocks together, the entry point address of an exit handler, the number of exit arguments, and one argument that is the address of a longword to receive the exit status value. Typically, additional arguments are specified to contain pointers and values that enable the exit handler to clean up a database. Figure 16-2 illustrates the format of an termination handler control block.

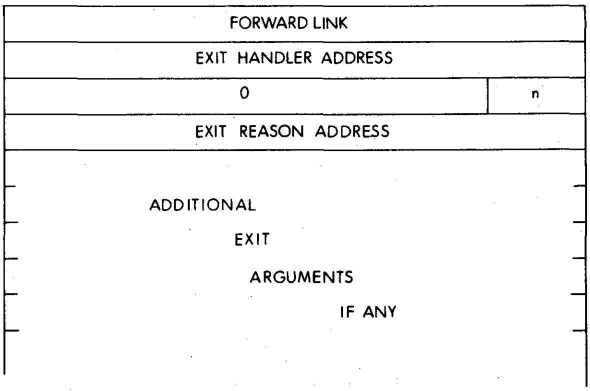


Figure 10-2 Termination Handler Control Block

The VAX/VMS operating system maintains a separate list of termination handler control blocks for each access mode. Each list is in last-in/first-out order. As each exit handler is specified, its termination handler control block is added to the front of the list for that access mode. The execution of an exit handler is a one-shot occurrence. That is, once executed, it must be respecified before it is executed again.

Exit Dispatching

The execution of exit handlers is triggered by a call to the Exit system service from user, supervisor, or executive mode. If the call is made from kernel mode, the process is immediately deleted after running down I/O and performing other cleanup operations. Otherwise, the appropriate lists of termination handler control blocks are examined to determine if any exit handlers were specified.

The exit handler dispatcher scans the list of termination handler control block one entry at a time. The respective exit handler is called for each one. The argument list specified in the call to the exit handler is that specified in the termination handler control block itself; the reason for the exit is filled into the longword whose address is specified by the first argument. If the entire list is scanned and control returns to the exit handler dispatcher (i.e., if none of the exit handlers resets and changes the flow of control), another Exit system service is executed.

ASYNCHRONOUS SYSTEM TRAPS

Certain system services allow a process to request an interrupt for notification of an event that occurs out of sequence with the execution of the process. The system enables a trap for the event and, when it occurs, the system delivers an interrupt to the process. Control is then passed to the user-specified routine that handles the interrupt.

Since the interrupt occurs asynchronously (out of sequence) with respect to the process's execution, the interrupt mechanism is called an asynchronous system trap (AST). That is, the process does not have direct control over the exact moment of AST delivery. The system services that use the AST mechanism accept as an argument the address of the AST service routine that should be given control when the interrupt is delivered and a longword argument.

The AST service routine executed as a result of specifying an AST entry point in a system service is a procedure. It is entered using a CALLG instruction and must exit using a RET instruction. The AST service routine executes at the access mode in effect when it was declared. The result is a call frame on the stack for the access mode of the AST receiver, as illustrated in Figure 16-3.

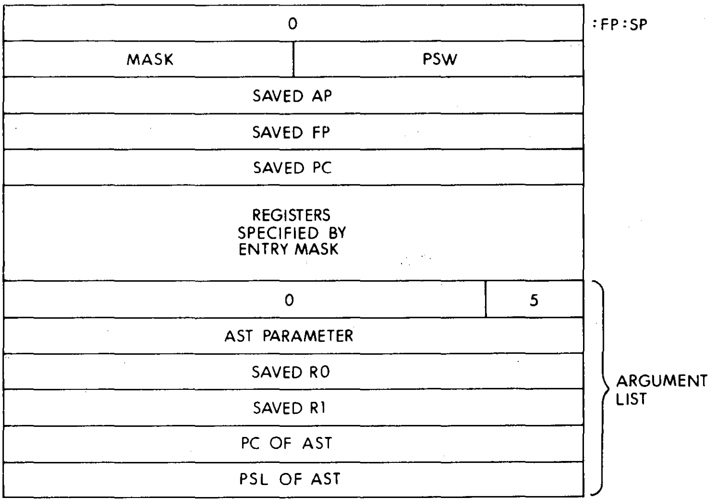


Figure 10-3 AST Receiver Stack Content

The argument list supplied to the AST routine is contained on the stack for the access mode receiving the AST. The registers PC and PSL in the argument list are those saved at the point at which AST delivery interrupted the process.

When an AST is requested for a process, the following three events occur:

1. The system queues the AST in an AST queue linked to that process's software process control block (PCB)
2. When appropriate enabling conditions exist, the AST is delivered to the process
3. The process's AST handling routine receives the AST

If conditions permit, the AST can be delivered directly to the process rather than being enqueued.

AST Enqueuing

The asynchronous system trap (AST) queue for a process is maintained in order of access mode. The highest privileged (lowest numbered) access mode is at the head of the queue. The queue is first-in/first-out within an access mode.

When an AST is specified for a process, it is either delivered directly to the process or queued to the PCB, depending on the setting of the AST control bits in the PCB and the state of the process. If the AST is deliverable based on a check of the AST enabled and active bits in the PCB, and if the process is currently executing, the AST is delivered to the process. The system computes the new value of the AST Level (ASTLVL) and stores it in the hardware PCB contained in the process header.

If the AST is deliverable and the process is the current process, the ASTLVL register is also updated. If the process is not the current process, an AST enqueuing event is reported for the process.

If the AST is deliverable but the process is nonresident, the AST is enqueued rather than delivered and an AST enqueuing event is reported. The swapper computes the proper ASTLVL value when the process is made resident.

If the AST is not deliverable based on the state of the AST enabled and active bits, the AST control block is placed in the proper position in the AST queue. An AST enqueuing event is reported.

I/O Status Posting AST

The posting of I/O status upon I/O completion is a special case of AST enqueuing. Using the AST mechanism, the posting of I/O status is performed in the context of the process that initiated the I/O operation.

The I/O status posting AST is executed in kernel mode at Interrupt Priority Level (IPL) 2. It moves the final I/O status to the specified I/O status block and moves the data for a buffered read from the system buffer to the process buffer. Then, it releases the system buffer.

A normal AST control block is queued for the process as a result of the handling of the I/O status posting request if a completion AST address is specified in the I/O request packet.

AST Delivery

The actual delivery of a pending asynchronous system trap (AST) is initiated by the AST delivery interrupt at interrupt priority level (IPL) 2. The interrupt is triggered as a result of an return from exception or interrupt (REI) instruction and is processed entirely on the kernel stack. When the interrupt occurs, the system first checks for the deliverability of the AST control block at the head of the queue. The AST is deliverable if all of the following condition are met:

1. ASTs are enabled for that access mode
2. No AST is active for that access mode
3. The process is not executing at a more privileged access mode

An immediate return is taken if the AST is not deliverable. This apparently redundant check is necessary to deal with an AST delivery interrupt triggered as a result of executing the previous process which is now inappropriate for the new current process.

If the AST control block is deliverable, the following steps are taken:

1. The AST control block is removed from the AST queue
2. The active bit for the proper access mode is set
3. A new value for ASTLVL is computed and placed into the ASTLVL field of the hardware PCB and the ASTLVL processor register

Normal AST control blocks (those other than I/O status, SUSPEND process, DELETE process, GETJPI, and power recovery ASTs) are processed by building an AST stack frame on the stack for the access mode of the receiver and removing the interrupt PC and PSL from the kernel stack. A new PC and PSL for the proper mode are constructed according to the AST control block. Both previous mode and current mode are set to the mode for which delivery is being made. The storage for the AST control block just serviced is released. Then, the AST handling routine specified in the control block is entered using the return from exception or interrupt (REI) instruction.

I/O status requests are processed in a highly streamlined fashion without building an AST stack frame. The I/O packet is either released or turned into a normal AST control block and requeued for the access mode originally making the I/O request.

Control of AST Delivery

Three methods exist for the control of AST delivery to a process:

1. Set AST Enable system service
2. Automatic disabling of ASTs for an access mode if an AST is active for that mode
3. Setting the IPL higher than the AST delivery interrupt to inhibit AST delivery (kernel mode only)

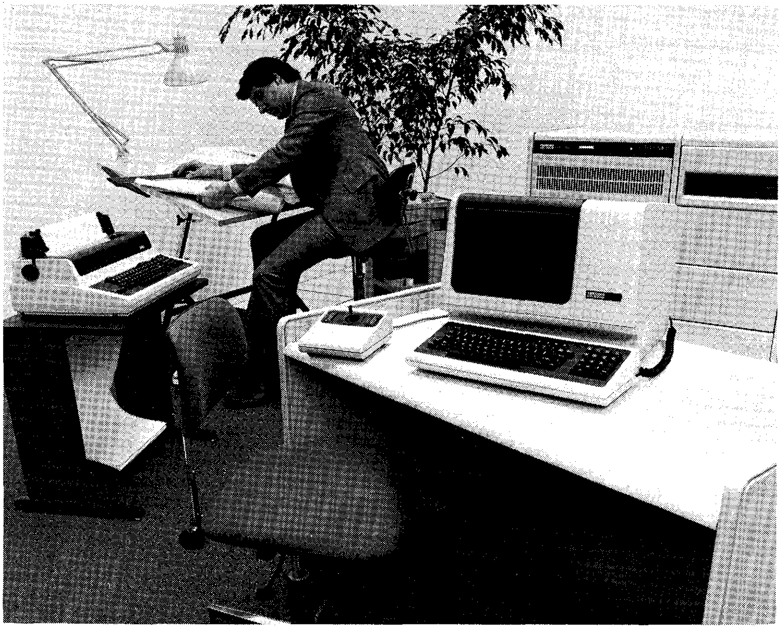
The AST Control system service allows a process to set or clear the AST enable bits for each of the four access modes (only at the mode of the caller). This method of AST control permits non-kernel mode routines to synchronize with their ASTs.

AST delivery is implicitly disabled for an access mode when an AST is currently active for that mode. The disable is removed when the AST procedure returns.

Within kernel mode routines, both AST delivery to kernel mode and interrupts can be disabled by raising the IPL.

Exception During AST Delivery

The AST delivery routine uses the exception mechanism to signal a software-detected condition if there is insufficient stack space to deliver the AST. The AST control block for the AST detecting the stack problem is released and the AST active state for the affected mode is cleared. When this occurs, the AST is lost; however, the information in the AST signal parameters is not. An AST fault condition is a serious error and is intended to provide information, but not to permit continuation.



CHAPTER OVERVIEW

A wide range of system services is incorporated into the VAX/VMS operating system in order to assure the smooth and efficient execution of user processes. The system services control input and output procedures, maintain logical and symbolic tables, handle exception conditions, provide system traps, and keep track of time and time conversion. In this chapter the calling standards for system services are listed with some call examples. Also, the algorithms which the system services operate are given for several cases.

Topics include:

- Event-related Services
- Asynchronous System Traps
- Logical Name Services
- I/O Services
- Timer and Time Services
- Exception Condition Services
- Process Control Services
- Memory Management Services
- Change Mode Services
- Lock Management Services

CHAPTER 11
SYSTEM SERVICES

INTRODUCTION

System services are procedures incorporated into and used by the operating system to control resources available to processes, to provide for communication among processes, and to perform basic operating system functions, such as the coordination of input/output operations.

The VAX/VMS system services can be called both from the VAX-11 MACRO assembly language and from the VAX high-level languages. The examples in this chapter are all MACRO calls; however, examples for other languages can be found in the language user's guides, and complete system services details can be found in *The VAX/VMS System Services Reference Manual.*

Although most system services are employed primarily by the operating system itself on behalf of logged-on users, many are generally available and provide techniques that can be used in application programs. For example, when a user logs onto the system, the Create Process system service is called to create a user process. The user, in turn, may call the Create Process service to create a subprocesss.

While many system services are available and suitable for application programming, the general use of certain services must be restricted to privileged users in order to protect the performance of the system and the integrity of user processes.

Information about a user's privileges is maintained by the system manager in the user authorization file (UAF). In addition to containing user profile information, the authorization file also contains a list of specific user privileges and resource quotas. When the user logs onto the system, the list of privileges and quotas assigned by the system manager to the user is associated with the process created on the user's behalf.

When the image issues a call to a system service that is protected by privilege, the privilege list is checked. If the image has been granted the specific system service privilege it requires, then the image is permitted to execute that system service; otherwise, a status code indicating an error is returned.

When a system service that uses a resource controlled by a quota is called, the process's quota for that resource is checked. If the process has exceeded its quota, or if it has no quota allotment, an error status code may be returned. In some cases, the process may be placed in a wait state until the resource becomes available.

Some system services provide techniques for coordinating and synchronizing the execution of different processes. These services enable users to control their subprocesses, allow users with group privilege to affect processes in their group, and give users with world privilege the ability to control any process.

A process can execute at any one of four access modes: user, supervisor, executive, or kernel. The access modes determine a process's ability to access pages of virtual memory. Each page has a protection code associated with it, specifying the type of access—read, write, or no access—allowed for each mode.

In some system service calls, the access mode of the caller is checked to see whether the caller may execute a particular function.

The system services are organized in the following functional categories:

- Event Flag Services
- Asynchronous System Trap (AST) Services
- Logical Name Services
- Input/Output Services
- Process Control Services
- Timer and Time Conversion Services
- Condition Handling Services
- Memory Management Services
- Change Mode Services
- Lock Management Services

The following sections describe each of the system services.

EVENT FLAG SERVICES

Event flag services are those services that allow a process or a group of cooperating processes to read, wait for, and manipulate event flags. A process can use event flags to synchronize sequences of operations in a program.

Event flags are status posting bits maintained by the VAX/VMS operating system for general programming use. Programs can use event flags to perform a variety of signaling functions:

- Setting or clearing specific flags
- Testing the current status of flags
- Placing the process in a wait state pending the setting of a specific flag or a group of flags

Moreover, event flags can be used in common by more than one process as long as the cooperating processes are in the same group.

Event flags may be set in shared memory as well as in local memory. Flags set in a multiport memory such as the MA780 multiport memory can be used to coordinate processes on different processors.

Some system services can set an event flag to indicate the completion or the occurrence of an event, and the calling program can test the flag. For example, the user can specify that the Queue I/O Request ($QIO) system service set an event flag when the requested input or output operation completes.

Each event flag is identified by a unique decimal number referred to by event flag arguments in system service calls. For example, if event flag 1 is specified in a call to the $QIO system service, then event flag number 1 is set when the I/O operation completes.

To allow manipulation of event flag groups, the event flags are ordered in clusters. Each cluster contains 32 event flags, numbered from right to left, corresponding to bits 0 through 31 in a longword. The system defines two types of clusters:

- A local event flag cluster can only be used internally by a single process. Local clusters are automatically available to each process
- A common event flag cluster can be shared by cooperating processes in the same group. Before a process can refer to a common event flag cluster, it must explicity "associate" with the cluster by using the Associate Common Event Flag Cluster ($ASCEFC) system service

The range of event flag numbers and the clusters to which they belong are summarized in Table 11-1.

Table 11-1 Summary of Event Flag and Cluster Numbers

Cluster Number	**Event Flag Numbers**	**Description**	**Restriction**
0 1	0-31 32-63	Process-local event flag clusters for general use	Event flags 24 through 31 are reserved for system use.
2 3	64-95 96-127	Assignable common event flag cluster	Must be associated before use

Listed below are the event flag system services.

Associate Common Event Flag Cluster—$ASCEFC

When a common event flag cluster is created, it must be identified by a 1- to 15-character name string. All processes that associate with the cluster must use the same name to refer to the cluster; the $ASCEFC system service establishes the correspondence between the cluster name and the actual cluster.

Before any processes can use event flags in a common event flag cluster, the cluster must be created: the Associate Common Event Flag Cluster ($ASCEFC) system service creates a common event flag cluster. If the cluster has already been created, other processes in the same group can call $ASCEFC to establish their association with the cluster and use its flags. The protection to be applied to the cluster and a permanent or nonpermanent status are assigned to the event flag cluster when it is created.

The following example shows how a process might create a common event flag cluster named COMMON-CLUSTER.

```
CLUSTER      .ASCID/COMMON-CLUSTER/;CLUSTER NAME
             .
             .
             $ASCFEC-S EFN=#65, NAME=CLUSTER ;CREATE
             ;CLUSTER
```

Disassociate Common Event Flag Cluster—$DACEFC

The Disassociate Common Flag Cluster system service disassociates the requesting process from the common event flag cluster that contains the specified event flag. If the common event cluster is temporary, it is deleted when the number of processes associated with it is zero. An implicit disassociate is performed for all clusters to which an image has associated, when the image exits.

The following example illustrates the disassociation of the user's process from the common event flag cluster containing event flag number 64.

```
CNAME:       .CLUSTER/;CLUSTER NAME
             .
             .
             $DACEFC-S EFN=#64;DISASSOCIATE CLUSTER
```

Delete Common Event Flag Cluster—$DLCEFC
The Delete Common Event Flag Cluster system service causes a permanent common event flag cluster to become nonpermanent. The cluster is actually deleted when no processes are associated with it. A process must have the privilege to create a permanent event flag cluster (PRMCEB) in order to delete one.

Set Event Flag—$SETEF
The Set Event Flag system service causes the specified event flag to be set and causes any processes waiting for the event to be made computable.

The following example associates the user process with common event flag cluster 3 and sets the third flag within the cluster. Note that event flag number 96 is equivalent to bit zero of the longword (cluster 3), and therefore event flag number 99 is equivalent to bit 3 in cluster 3.

```
SHARE:      .ASCID/COMMON-CLUSTER/;CLUSTER NAME
            .
            .
            .
            $ASCFEC-S EFN=#96, NAME=SHARE ;ASSOCIATE WITH
            ;CLUSTER
            $SETEF-S EFN=#99 ;SET 3RD FLAG IN COMMON-CLUSTER
```

Clear Event Flag—$CLREF
The Clear Event Flag system service sets an event flag in a local or common event flag cluster to 0.

The following example illustrates a system service call that clears event flag 32.

```
$CLREF_S EFN=#32
```

Read Event Flags—$READEF
The Read Event Flags system service returns the current status of all 32 event flags in a local or common event flag cluster.

Wait For Single Event Flag—$WAITFR
The Wait For Single Event Flag system service tests the specified event flag and returns immediately if the event flag is set. Otherwise, the process is placed in a wait state until the event flag is set.

The user's process can be placed in a wait state for a pre-determined period of time by specifying an event flag argument to the $SETIMR service and then using the Wait For Single Event Flag system service as follows:

```
TIME:          .BLKQ              1                   ;WILL CONTAIN TIME INTERVAL TO WAIT
               .
               .
               .
               $SETIMR-S          EFN=#33, DAYTIM=TIME        ;SET THE TIMER
               $WAITFR-S          EFN=#33,            ;WAIT UNTIL TIMER EXPIRES
```

Wait For Logical OR of Event Flags—$WFLOR

The Wait for Logical OR of Event Flags system service tests the event flags specified by a mask within a specified cluster and returns immediately if any of the specified flags are set. Otherwise, the process is placed in a wait state until at least one of the selected event flags is set.

Wait for Logical AND of Event Flags—$WFLAND

The Wait for Logical AND of Event Flags system service allows a process to specify a mask of event flags for which it wishes to wait. All of the indicated event flags within a specified event cluster must be set; the process is placed in a wait state until they are all set.

The following example illustrates a program that issues two $QIO system service calls, and uses the $WFLAND system service to wait until both I/O operations complete before it continues execution.

The MASK argument specifies which flags in the cluster are to be waited for: the first and second. The EFN argument specifies any flag number in the cluster containing flags for which you are waiting.

```
$QIO-S       EFN=#1,...             ;ISSUE FIRST QUEUE I/O REQUEST
BSBW         ERROR                  ;CHECK FOR ERROR
$QIO-S       EFN=#2,...             ;ISSUE SECOND I/O REQUEST
BSBW         ERROR                  ;CHECK FOR ERROR
$WFLAND-S    EFN=#1, MASK=#↑B0110   ;WAIT UNTIL BOTH COMPLETE
BSBW         ERROR                  ;CHECK FOR ERROR
.
.
.                                   ;CONTINUE EXECUTION
```

ASYNCHRONOUS SYSTEM TRAP (AST) SERVICES

Various system services allow a process to request that it be interrupted when a particular event (such as I/O completion) occurs. Since the interrupt occurs asynchronously with respect to the process's execution, the interrupt mechanism is called an asynchronous system trap (AST). The trap provides a transfer of control to a user-specified routine that handles the event.

The system services that use the AST mechanism accept as an optional argument the address of an AST service routine, that is, a routine to be given control when the event occurs.

These service routines are:

- Queue I/O Request ($QIO)
- Set Timer ($SETIMR)
- Set Power Recovery AST ($SETPRA)
- Update Section File on Disk ($UPDSEC)
- Get Job/Process Information ($GETJPI)

For example, when the user calls the Set Timer ($SETIMR) system service, the user can specify the address of a routine to be executed at a particular time of day or when a time interval expires.

The service sets the timer and returns; the program image continues executing. When the requested timer event occurs, the system "delivers" an AST by interrupting the process and calling a specified routine, unless AST delivery is temporarily blocked. (Conditions that can prevent AST delivery are explained later on in this section).

Each request for an AST is qualified by the access mode from which the AST is requested. Thus, if an image executing in user mode requests notification of an event by means of an AST, the AST service routine executes in user mode.

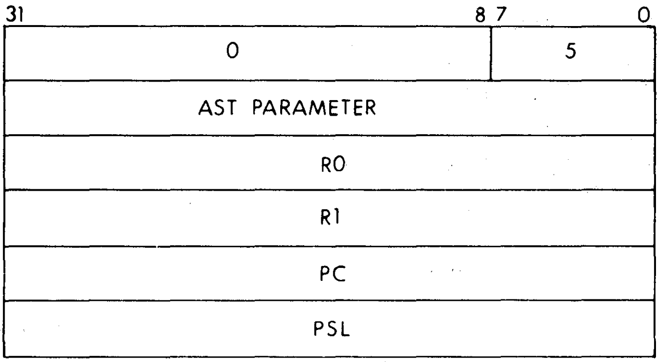
A process that is in certain wait states can be interrupted for the delivery of an AST and the execution of an AST service routine. When the AST service routine completes execution, the process is returned to the wait state, if the condition that caused the wait is still in effect.

The following wait states may be interrupted:

- Event flag waits
- Hibernation
- Resource waits and page fault waits

An AST routine must be a separate routine. The system calls the AST with a CALLG instruction; the routine must return using a RET instruction. If the service routine modifies any registers other than R0 or R1, it must set the appropriate bits in the entry mask so that the contents of those registers are saved.

On entry to the AST service routine, the Argument Pointer (AP) register points to an argument list that has the format:



The registers R0 and R1, the PC, and PSL in this list are those that were saved when the process was interrupted by delivery of the AST.

The AST parameter is an argument passed to the AST service routine so that it can identify the event that caused the AST.

When an AST occurs, the system may not be able to deliver the interrupt to the service routine immediately. An AST cannot be delivered if any of the following conditions exist:

1. An AST service routine is currently executing at the same or at a more privileged access mode
2. AST delivery is explicitly disabled for the access mode of the AST being delivered
3. The process is executing at an access mode more privileged than that for which the AST is declared

If an AST cannot be delivered when the interrupt occurs, the AST is queued until the conditions disabling delivery are removed. Queued ASTs are ordered by the access mode from which they were declared, with those declared from more privileged access modes at the front of the queue. If more than one AST is queued for an access mode, the ASTs are delivered in the order in which they are queued.

The following example illustrates a program that calls the $SETIMR system service with a request for an AST when a timer event occurs.

```
NOON:       .BLKQ       1                          ;WILL CONTAIN 12:00 SYSTEM TIME
LIBRA:      WORD        0                          ;ENTRY MASK FOR LIBRA
            .
            .
            .
            $SETMIR-S   DAYTIM=NOON.ASTADIR=TIMEAST          :SET TIMER
            BSBW        ERROR                      ;CHECK FOR ERROR
            .                                      Timer
            .                                      Interrupt
            .
TIMEAST:
            .WORD       0                          ;ENTRY MASK FOR AST ROUTINE
            .                                      ;HANDLE TIMER REQUEST
            .
            RET                                    ;DONE
            .END        LIBRA
```

- The call to the $SETIMR system service requests an AST at 12:00 noon
- The DAYTIM argument refers to the quadword NOON, which must contain the time in system time format. For details on how this is done, see "Timer and Time Conversion Services." The ASTADR argument refers to TIMEAST, the address of the AST service routine
- When the call to the system service completes, the process continues execution
- The timer expires at 12:00 and notifies the system. The system interrupts execution of the process and gives control to the AST service routine
- The user routine TIMEAST handles the interrupt. When the AST routine completes, it issues a RET instruction to return control to the program. The program resumes execution at the point at which it was interrupted

Listed below are the services that enable or disable AST delivery or that require an AST service routine as an argument. (Other services accept an AST service routine as an optional argument.)

Set AST Enable—$SETAST
The Set AST Enable system service enables or disables the delivery of ASTs for the access mode from which the service call was issued.

Declare AST—$DCLAST
The Declare AST system service queues an AST for calling or for a less privileged access mode. For example, a routine executing in supervisor mode can declare an AST for either supervisor or user mode.

Set Power Recovery AST—$SETPRA
The Set Power Recovery AST system service establishes a routine to receive control using the AST mechanism after a power recovery is detected.

LOGICAL NAME SERVICES

The VAX/VMS logical name services provide a technique for manipulating and substituting character string names. Before discussing the logical name services, the nature and use of logical names themselves and of the software structures known as logical name tables will be examined.

Logical names are commonly used to specify devices or for input/output operations. The user can code programs with logical or symbolic names to refer to physical devices or files, and then establish an equivalence or real name by issuing the ASSIGN command from the command stream prior to program execution. When the program executes, a reference to the logical name results in the substitution of the equivalence name.

Logical and equivalence name pairs are maintained in three logical name tables. Each table is associated with a unique number identifier, as follows:

Table	Number
Process	2
Group	1
System	0

A process logical name table contains names used exclusively by the process. A process logical name table exists for each process in the system. Some entries in the process logical name table are made by system programs executing at more privileged access modes; these entries are qualified by the access mode from which the entry was made. Table 11-2 illustrates a user process logical name table.

This process logical name table equates the logical name TERMINAL to the specific terminal TTA2:. INFILE and OUTFILE are equated to disk file specifications: these logical names were created from supervisor mode.

Table 11-2 Sample Process Logical Name Table (Group=200)

Logical Name	Equivalence Name	Access Mode
TERMINAL	TTA2:	User
INFILE	DM1:[HIGGINS]TEST.DAT	Supervisor
OUTFILE	DM1:[HIGGINS]TEST.OUT	Supervisor

The group logical name table contains names that cooperating processes in the same group can use. The user must have special privilege to place a name in the group logical name table. Table 11-3 illustrates a sample group logical name table.

Table 11-3 Sample Group Logical Name Table

Logical Name	Equivalence Name	Group Number
TERMINAL	TTA1:	100
MAILBOX	MB3:	200
DISPLAY	TERMINAL	200
TERMINAL	TTA3:	300

The group logical name table shows entries qualified by group numbers; only processes that have the indicated group number can access these entries.

In Group 100, the logical name TERMINAL is equated to the terminal TTA1:. Individual processes in Group 100 that want to refer to the logical name TERMINAL do not individually have to assign it an equivalence name.

Group 200 has entries for logical names MAILBOX and DISPLAY. Other processes in Group 200 can use these logical names for input and output operations.

In Group 300, the logical name TERMINAL is equated to the physical device name TTA3:. Note that there are two entries for TERMINAL in the group logical name table. These are discrete entries, since they are qualified by the number of the group to which they belong. Other processes in Group 300 can refer to this logical name TERMINAL without individually having to assign it an equivalence name.

The system logical name table contains names that all processes in the system can access. This table includes the default names for all system-assigned logical names. Only users with special privilege may place a name in the system logical table. Table 11-4 illustrates a system logical name table.

Table 11-4 Sample System Logical Name Table

Logical Name	Equivalence Name
SYS$LIBRARY	DBA0:[SYSLIB]
SYS$SYSTEM	DBA0:[SYSEXE]

The system logical name table contains the default physical device names for all processes in the system. SYS$LIBRARY and SYS$SYSTEM provide logical names for all users to refer to the directories containing system files.

The VAX/VMS operating system logical name services are listed below.

Create Logical Name—$CRELOG

The Create Logical Name system service inserts a logical name and its equivalence name into the process, group, or system logical name table. If the logical name already exists in the respective table, the new definition supersedes the old.

In the following example, the user can perform an assignment within a program by providing character string descriptors for the name strings and use the $CRELOG system service. The logical name TERMINAL is equated to the physical device name TTA2:.

```
TERMINAL:     DESCRIPTOR                            ;DESCRIPTOR FOR LOGICAL NAME
TTNAME:       DESCRIPTOR                            ;DESCRIPTOR FOR EQUIVALENCE
              .
              .
              $CRELOG-S            TBLFLG=#2.LOGNAM=TERMINAL.EQLNAM= TTNAME
```

The TBLFLG argument indicates the logical name table number, in this case, the process logical name table.

Delete Logical Name—$DELLOG

The Delete Logical Name system service deletes a logical name and its equivalence name from the process, group, or system logical name table.

For example, the following call deletes all names from the process logical name table that were entered in the table from user mode:

```
$DELLOG-S TBLFLG=#2
```

Translate Logical Name—$TRNLOG

The Translate Logical Name system service searches the logical name tables for a specified logical name and returns an equivalence name string. The process, group, and system logical name tables are searched, in that order.

INPUT/OUTPUT SERVICES

The VAX/VMS operating system provides the user with two methods to perform input/output operations:

- Indirectly, through VAX-11 Record Management Services (RMS)
- Directly, through input/output system services

VAX-11 RMS provides a set of macros for general purpose, device-independent functions, such as data storage, retrieval, and modification.

The I/O system services permit the user to utilize the I/O resources of the operating system directly in a device-dependent manner. The I/O system services can perform the following functions:

- Assign and deassign channels
- Queue I/O requests
- Synchronize I/O completion
- Allocate and deallocate devices
- Create mailboxes
- Perform network operations

Listed below are the input/output system services.

Assign I/O Channel—$ASSIGN

The Assign I/O Channel system service 1) provides a path between a device and an I/O channel so that input/output operations can be performed on the device, or 2) establishes a logical link with a remote node on a network.

When coding a call to the $ASSIGN service, the following arguments must be passed:

- Name of device (physical or logical device name)
- Address of word to receive channel number

The service returns a channel number which must be used when coding an input or output request. In the following example, an I/O channel is assigned to device TTA2. The channel number is returned in the word whose address is TTCHAN.

```
TTNAME:         .ASCID              /TTA2/          ;TERMINAL DESCRIPTOR
TTCHAN:         .BLKW               1               ;TERMINAL CHANNEL NUMBER
                .
                .
                $ASSIGN S DEVNAM=TTNAME,CHAN=TTCHAN
```

Deassign I/O Channel—$DASSGN
The Deassign I/O Channel system service releases an I/O channel acquired for input/output operations with the Assign I/O channel ($ASSIGN) system service.

In the following example, the user releases the terminal channel assignment acquired in the previous $ASSIGN example.

```
        $DASSGN_S CHAN=TTCHAN
```

Queue I/O Request—$QIO
The Queue I/O Request system service initiates an input or output operation by queuing a request to a device associated with a specific channel. Control returns immediately to the issuing process, which can synchronize I/O completion in one of three ways:

1. Specify the address of an AST routine that is to execute when the I/O completes
2. Wait for a specified event flag to be set
3. Poll the specified I/O status block for a completion status

The event flag and I/O status block, if specified, are cleared before the I/O request is queued.

In the following example, the user synchronizes I/O completion by coding an event flag as an argument to $QIO.

```
$QIO_S EFN=#1,...                ;ISSUE 1ST I/O REQUEST
BSBW ERROR                       ;QUEUED SUCCESSFULLY?
$QIO_S EFN=#2,...                ;ISSUE 2ND I/O REQUEST
BSBW ERROR                       ;QUEUED SUCCESSFULLY?
$WFLAND_S EFN=#0,MASK=#↑B0110
;WAIT TIL BOTH DONE
```

- When an event flag number is coded as an argument, $QIO clears the event flag when it queues the I/O request. When the I/O completes, the flag is set
- In this example, the program issues two I/O requests. A different event flag is specified for each request
- The Wait for Logical AND of Event Flags ($WFLAND) system service places the process in a wait state until both I/O operations are complete. The EFN argument can specify any flag in the cluster containing the flags for which the user is waiting. The MASK argument indicates the specific flags for which the user is waiting

Queue I/O Request and Wait For Event Flag—$QIOW
The Queue I/O Request and Wait for Event Flag system service combines the $QIO and $WAITFR (Wait for Single Event Flag) system services. It can be used when a program must wait for I/O completion.

Queue Input Request and Wait For Event Flag—$INPUT
The $INPUT macro is a simplified form of the Queue I/O Request and Wait for Event Flag ($QIOW) system service. This macro queues a virtual input operation using the IO$_READVBLK function code and waits for I/O completion.

Queue Output Request and Wait for Event Flag—$OUTPUT
The $OUTPUT macro is a simplified form of the Queue I/O Request and Wait for Event Flag ($QIOW) system service. This macro performs a virtual output operation using the IO$_WRITEVBLK function code and waits for I/O completion.

Formatted ASCII Output—$FAO
The Formatted ASCII Output system service converts binary values into ASCII characters and returns the converted characters in an output string. It can be used to:

- Insert variable character string data into an output string
- Convert binary values into the ASCII representations of their decimal, hexadecimal, or octal equivalents and substitute the result into an output string

Input to the $FAO service consists of:

1. A control string that contains the fixed text portion of the output and formatting directives. The directives indicate the position within the string where substitutions are to be made, and describe the data type and length of the input values that are to be substituted or converted
2. An output buffer to contain the string after conversions and substitutions have been made
3. An optional argument indicating a word to receive the final length of the formatted output string
4. Parameters that provide arguments for the directive

Formatted ASCII Output with List Parameter—$FAOL
The Formatted ASCII Output with List Parameter macro provides an alternative way to specify input parameters for a call to the $FAO system service.

Allocate Device—$ALLOC
The Allocate Device system service reserves a device for exclusive use by a process and its subprocesses. No other process can allocate the device or assign channels to it until the image that called $ALLOC exits or explicitly deallocates the device with the Deallocate Device ($DALLOC) system service.

In coding the $ALLOC system service, a device name must be provided. The device name specified can be:

- A physical device name, for example, the tape drive MTB3:
- A logical name, for example, TAPE
- A generic device name, for example, MT:

If the user specifies a physical device name, $ALLOC attempts to allocate the specified device. If the user specifies a logical name, $ALLOC translates the logical name and attempts to allocate the physical device name equated to the logical name. If the user specifies a generic device name, but not a specific controller and/or unit number, $ALLOC attempts to allocate any available device of the specified type.

The following example illustrates the allocation of a tape device specified by the logical name TAPE.

```
LOGDEV:     .ASCID              /TAPE/          ;LOGICAL NAME FOR TAPE
DEVDESC:                                        ;DESCRIPTOR FOR PHYSICAL NAME
            .LONG               64              ;LENGTH OF BUFFER
            .LONG               DEVDESC8        ;ADDRESS OF BUFFER
            .BLKB               64              ;GET PHYSICAL NAME RETURNED
TAPECHAN:
            .BLKW               1               ;CHANNEL FOR I/O TAPE
            .
            .
            .
            $ALLOC-S DEVNAM=LOGDEV, PHYLEN=DEVDESC=DEVDESC,-
            PHYBUF=DEVDESC
            .
            .
            BSBW                ERROR
            $ASSIGN-S           DEVNAM=DEVDESC,CHAN=TAPECHAN          ;ASSIGN
CHANNEL
            BSBW                ERROR
                                                ;CONTINUE WITH I/O
            .
            $DASSGN-S           CHAN=TAPECHAN   ;DEASSIGN CHANNEL
            BSBW                ERROR
            $DALLOC-S           DEVNAM=DEVDESC        ;DEALLOCATE TAPE
```

- The $ALLOC system service call requests allocation of a device corresponding to the logical name TAPE, defined by the character string descriptor LOGDEV. The PHYBUF argument refers to the buffer provided to receive the physical device name of the device actually allocated, and its length. $ALLOC translates the logical name TAPE and returns the equivalence name string into the buffer at DEVDESC. It writes the length of the string in the first word of DEVDESC
- The $ASSIGN command uses the character string returned by the $ALLOC system service as the input device name argument, and requests that the channel number be written into TAPECHAN

- When I/O operations are completed, the $DASSGN system service deassigns the channel and the $DALLOC system service deallocates the device. The channel must be deassigned before the device can be deallocated

Deallocate Device—$DALLOC
The Deallocate Device system service deallocates a previously allocated device. Exclusive use by the issuing process is relinquished and other processes can assign or allocate the device.

The following example illustrates device deallocation.

```
        $DALLOC_S DEVNAM=DEVDESC
```

The system automatically deallocates at image exit any devices that were allocated from within the image.

Mount Volume—$MOUNT
The Mount Volume system service allows a process to mount a single volume, or a volume set. A device name, a volume name, and a logical name must be specified.

Dismount Volume—$DISMOU
The Dismount Volume system service allows a process to dismount a volume set. A call to $DISMOU must specify a device name. If the volume mounted on the device is part of a full mounted volume set, and no flags are specified, the whole volume set is dismounted.

Get I/O Channel Information—$GETCHN
The Get I/O Channel Information system service returns information about a device to which an I/O channel has been assigned. Two sets of information are optionally returned:

- Primary device characteristics
- Secondary device characteristics

In most cases, the two sets of characteristic information are identical. However, the two sets provide different information in the following cases:

- If the device has an associated mailbox, the primary characteristics are those of the assigned device and the secondary characteristics are those of the associated mailbox
- If the device is a spooled device, the primary characteristics are those of the intermediate device and the secondary characteristics are those of the spooled device
- If the device represents a logical link on the network, the secondary characteristics contain information about the link

Get I/O Device Information—$GETDEV
The Get I/O Device Information system service returns information about an I/O device. This service allows a process to obtain information about a device to which the process has not assigned a channel.

Get Device/Volume Information—$GETDVI
The $GETDVI system service returns information about an I/O device. As with the $GETDEV system service, the process does not need to have an I/O channel assigned to the device.

Cancel I/O on Channel—$CANCEL
The Cancel I/O on Channel system service cancels all pending I/O requests on a specific channel. This may include the request currently in progress, as well as all I/O requests queued.

For example, the $CANCEL system service can be called as follows:

```
        $CANCEL_S CHAN=TTCHAN
```

In this example, the $CANCEL system service initiates the cancellation of all pending I/O requests on the channel whose number is located at TTCHAN.

Create Mailbox and Assign Channel—$CREMBX
Mailboxes are virtual devices that can be used for communication between processes. Actual data transfer is accomplished by using RMS or I/O services. When a mailbox is created, a channel is assigned to it for use by the creating process. Other processes can then assign channels to the mailbox using the $CREMBX or $ASSIGN system service.

The Create Mailbox and Assign Channel ($CREMBX) system service creates the mailbox or, if the specified mailbox exists, assigns a channel to it. When the $CREMBX service creates a mailbox, it identifies the mailbox by a user-specified logical name and assigns it an equivalence name. The equivalence name is a physical device name in the format MBAn:, where n is a unit number.

When another process assigns a channel to the mailbox with the $ASSIGN system service, it can identify the mailbox by its logical name. $ASSIGN automatically translates the logical name. The process can obtain the MBAn: name by translating the logical name (with the $TRNLOG system service), or it can call the Get I/O Channel Information ($GETCHN) system service to obtain the unit number and the physical device name.

Mailboxes are either temporary or permanent; user privileges are required to create either type. $CREMBX enters the logical name and equivalence name for a temporary mailbox in the group logical name table of the process that created it. The system deletes a temporary mailbox when no more channels are assigned to it.

The $CREMBX system service enters the logical name and equivalence name for a permanent mailbox in the system logical name table. Permanent mailboxes continue to exist until they are specifically marked for deletion with the Delete Mailbox ($DELMBX) system service.

Delete Mailbox—$DELMBX

The Delete Mailbox system service marks a permanent mailbox for deletion. The actual deletion of the mailbox and of its associated logical name assignment occurs when no more I/O channels are assigned to the mailbox.

Broadcast—$BRDCST

The Broadcast system service writes a message to one or more terminals.

Send Message to Accounting Manager—$SNDACC

The Send Message to Accounting Manager system service controls accounting log activity and allows a process to write an arbitrary data message into the accounting log file.

By default, the system writes a record into the accounting log file whenever a job terminates. Termination records are written for interactive users, batch jobs, non-interactive processes, log-in failures, and print jobs. The $SNDACC system service allows users to write additional data into the accounting log and allows privileged users to disable or enable all accounting or accounting for particular types of jobs.

Send Message to Symbiont Manager—$SNDSMB

The Send Message to Symbiont Manager system service is used by the operating system to queue users' print files to a system printer or to queue command procedure files for detached job execution.

Symbiont manager requests:

- Create and delete queues
- Add or delete files from a queue
- Change the attributes of files in a queue
- Start and restart dequeuing

Send Message to Operator—$SNDOPR
The Send Message to Operator system service allows a process to send a message to one or more terminals designated as operators' terminals and optionally receive a reply.

This service is used by the system to implement the REQUEST and REPLY commands, which provide communication between users and operators. An operator establishes a terminal as an operator's console by issuing the REPLY/ENABLE command, specifying the types of message that will be handled. Users can then send messages to the operator with the REQUEST command, optionally requesting replies.

Send Message to Error Logger—$SNDERR
The Send Message to Error Logger system service writes an arbitrary message to the system error log file. The user-specified message is preceded by the date and time.

Get Message—$GETMSG
The Get Message system service locates and returns message text associated with a given message identification code into the caller's buffer. The message can be from the system message file or can be a user-defined message.

This service is used by the operating system to retrieve messages based on unique message identifications and to prepare to output them.

Put Message—$PUTMSG
The Put Message system service is a generalized message formatting and output routine used by the operating system to write informational and error messages to user processes.

$PUTMSG retrieves a message from the system message file by calling the Get Message ($GETMSG) system service and formats the message by calling the Formatted ASCII Output ($FAO) system service, if necessary. If the caller specifies an action routine to receive control, the action routine is called before $PUTMSG writes each formatted message to the process's current output device. If the process's error device is different than the output device, $PUTMSG writes the message to the error device as well.

The action routine can access the message text, scan it, write it to a user-specified file or device, modify it, and so on.

PROCESS CONTROL SERVICES
A process is the basic executable entity scheduled by the system software. It provides the context in which an image executes. When

the user logs onto the system, the system creates a process for the execution of program images.

A process is either a subprocess or a detached process. A subprocess receives a portion of its creator's resource quotas, and must terminate before the creator. A detached process is fully independent. An example of a detached process is the process created by the system for the user during login.

Process control services allow the user to create, delete, and control the execution of processes.

Create Process—$CREPRC
The Create Process system service allows a process to create another process. The created process can be either a subprocess or a detached process.

When coding the $CREPRC system service, the IMAGE argument must be provided. This argument provides the process with the name of the program image to execute. The specification of the UIC argument controls whether the created process is a subprocess or a detached process. In the following example, a subprocess is created to execute the program image in the file named LIBRA.EXE.

```
PROGRAM:     .ASCID          /LIBRA/              ;IMAGE TO EXECUTE
             .
             .
             $CREPRC-S       IMAGE=PROGRAM,...    ;CREATE  PROCESS  TO  EXE-
CUTE LIBRA
```

In this example, only a file name is specified; the service uses current disk and directory defaults, performs logical name translation, uses the default file type of EXE, and locates the most recent version of the image file. When the subprocess completes execution of the image, the subprocess is deleted.

Delete Process—$DELPRC
The Delete Process system service allows a process to delete itself or another process.

Process deletion completely removes a process from the system. Deletion occurs as a result of any of the following conditions:

- The command stream contains a LOGOUT command or an end-of-file
- An image specified by $CREPRC exits
- A process issues the Delete Process ($DELPRC) system service

User privileges are required to delete:

- Other processes in the same group (GROUP privilege)
- Any process in the system (WORLD privilege)

For example, if a process has created a subprocess named ACE, it can delete the subprocess as shown below:

```
PROCESS:        .ASCID/ACE/
                .
                .
                $DELPRC-S PRCNAM=PROCESS
```

Hibernate—$HIBER

There are two ways to halt the execution of a process temporarily: hibernation, performed by the Hibernate ($HIBER) system service, and suspension, performed by the Suspend Process ($SUSPND) system service. However, hibernation and suspension differ in the following ways:

Process Hibernation and Suspension

Hibernation	*Suspension*
Can only hibernate self	Can suspend self or another process, depending on privilege
Reversed by $WAKE system service	Reversed by $RESUME system service
Interruptible; can receive ASTs	Noninterruptible; cannot receive ASTs
Can wake self	Cannot cause self to resume
Can schedule wakeup at an absolute time or at a fixed time interval	Cannot schedule resumption
Hibernate/wake complete quickly; require little system overhead	Requires system dynamic memory

The Hibernate ($HIBER) system service allows a process to make itself inactive but to remain known to the system so that it can be interrupted, for example, to receive ASTs. A hibernate request is a wait-for-wake-event request. When a wake is issued for a hibernating process with the $WAKE system service or as a result of a Schedule Wakeup ($SCHDWK) system service, the process continues execution at the instruction following the Hibernate call.

Wake—$WAKE

The Wake system service activates a process that has placed itself in a state of hibernation with the Hibernate ($HIBER) system service.

In the following example, the $WAKE system service is issued to wake (activate) the process ORION.

```
ORIONDESC:      .ASCID                /ORION/          ;DESCRIPTOR FOR PROCESS NAME
                .
                .
                .
                $WAKE-S PRCNAM=ORIONDESC                                   ;WAKE
ORION
                BSBW                  ERROR
                .
                .
                .
```

Schedule Wakeup—$SCHDWK

The Schedule Wakeup system service schedules the awakening of a process that has placed itself in a state of hibernation with the Hibernate ($HIBER) system service. A wakeup can be scheduled for a specified absolute time or for a delta time. Optionally, the request can specify that the wakeup is to be repeated at fixed intervals.

For an example of schedule wakeup, refer to "Timer and Time Conversion Services."

Suspend Process—$SUSPND

The Suspend Process system service allows a process to suspend itself or another process. A suspended process cannot receive ASTs or otherwise be executed until another process resumes or deletes it.

User privileges are required to suspend:

- Other processes in the group (GROUP privilege)
- Any other process in the system (WORLD privilege)

Resume Process—$RESUME

The Resume Process system service causes a process previously suspended by the Suspend Process ($SUSPND) system service to resume execution, or cancels the effect of a subsequent suspend request.

User privileges are required to resume execution of:

- Other processes in the same group (GROUP privilege)
- Any other process in the system (WORLD privilege)

Cancel Wakeup—$CANWAK

The Cancel Wakeup system service removes all scheduled wakeup requests for a process from the timer queue, including those made by the caller or by other processes. Scheduled wakeup requests are made with the Schedule Wakeup ($SCHDWK) system service.

User privileges are required to cancel scheduled wakeup requests for:

- Other processes in the same group (GROUP privilege)
- Any other process in the system (WORLD privilege)

Exit—$EXIT

The Exit system service is used by the operating system to initiate image rundown when the current image in a process completes execution. Control normally returns to the command interpreter.

Force Exit—$FORCEX

The Force Exit system service causes an Exit ($EXIT) system service call to be issued on behalf of a specified process.

User privileges are required to force an exit for:

- Other processes in the same group (GROUP privilege)
- Any other process in the system (WORLD privilege)

In the following example, a call to $FORCEX causes the image executing in the process named SMITH to exit.

```
PROGNAME:      /SMITH/                                ;DESCRIPTOR FOR PROCESS NAME
               .
               .
               $FORCEX-S PRCNAM=PROGNAME
```

Declare Exit Handler—$DCLEXH

The Declare Exit Handler system service describes an exit handling routine to receive control when an image exits. Image exit normally occurs when the image currently executing in a process returns control to the operating system. Image exit may also occur when the Exit ($EXIT) or Force Exit ($FORCEX) system service is called.

The following example illustrates the use of the Declare Exit Handler system service.

```
EXITBLOCK:                                                   ;EXIT CONTROL BLOCK
              .LONG                0                         ;SYSTEM USES THIS FOR POINTER
              .LONG                EXITRTN                   ;ADDRESS OF EXIT HANDLER
              .LONG                1                         ;NUMBER OF ARGS FOR HANDLER
              .LONG                STATUS                    ;ADDRESS TO RECEIVE STATUS CODE
STATUS:       BLKL                 1                         ;STATUS CODE FROM$EXIT
              .
              .
PEGASUS:      WORD                 ↑M                        ;ENTRY MASK FOR PEGASUS
              $DCLEXH-S            DESBLK=EXITBLOCK          ;DECLARE EXIT HANDLER
              .
              .
              RET                                            ;END OF MAIN ROUTINE
EXITRTN:                                                     ;EXIT HANDLER
              WORD                 ↑M                        ;ENTRY MASK
              CMPL                 STATUS,#SS$-NORMAL        ;NORMAL EXIT?
              BEQL                 10$                       ;YES,FINISH
              .                                              ;NO,CLEAN UP
              .
10$:          RET                                            ;FINISHED
```

- EXITBLOCK is the exit control block for the exit handler EXITRTN. The third longword indicates the number of arguments to be passed. In this example only one argument is passed; this is the address of a longword for the system to store the return status code. This argument must be provided in an exit control block
- The $DCLEXH system service call designates the address of the exit control block, thus declaring EXITRTN as an exit handler
- EXITRTN checks the status code. If this is a normal exit, EXITRTN returns control. Otherwise, it handles the error condition

Cancel Exit Handler—$CANEXH

The Cancel Exit Handler system service deletes an exit control block from the list of control blocks for the calling access mode. Exit control blocks are declared by the Declare Exit Handler ($DCLEXH) system service, and are queued according to access mode in a last-in, first-out order.

Set Process Name—$SETPRN

The Set Process Name system service allows a process to establish or to change its own process name.

A process can set or change its own name with the Set Process Name ($SETPRN) system service. For example, a process can set its name to DIPSY as follows:

```
DIPSY:        DESCRIPTOR                                     ;NAME DESCRIPTOR
              .
              .
              .
              $SETPRN-S PRCNAM=DIPSY
```

Set Priority—$SETPRI

The Set Priority system service changes a process's base and current priority. The system scheduler uses the current priority to determine the order in which executable processes are to run.

User privileges are required to:

- Change the priority for other processes in the same group (GROUP privilege)
- Change the priority for any other process in the system (WORLD privilege)
- Set any process's priority to a value greater than one's own initial base priority (SETPRI privilege)

Set Resource Wait Mode—$SETRWM

The Set Resource Wait Mode system service allows a process to indicate what action a system service should take when it lacks a system resource required for its execution:

- When resource wait mode is enabled (the default mode), the service waits until a resource is available and then resumes execution
- When resource wait mode is disabled, the service returns control to the caller immediately with a status code indicating that a resource is unavailable

Get Job/Process Information—$GETJPI

The Get Job/Process Information system service provides accounting, status, and identification information about a specified process.

User privileges are required to obtain information about:

- Other processes in the same group (GROUP privilege)
- Any other process in the system (WORLD privilege)

Set Privileges—$SETPRV

The Set Privileges system service allows a process to enable or disable specified user privileges.

TIMER AND TIME CONVERSION SERVICES

Many applications require the scheduling of program activities based on clock time. In VAX/VMS, an image can schedule events for a specific time of day, or after a specified time interval. Timer services can:

- Schedule setting an event flag or queuing an asynchronous system trap (AST) for the current process, or cancel a pending request that has not yet been honored
- Schedule a wakeup request for a hibernating process, and cancel a pending wakeup request that has not yet been honored
- Set the system time

VAX/VMS maintains the current date and time (using a 24-hour clock) in 64-bit format. The time value is a binary number in 100-nanosecond units offset from the system base date and time, which is 00:00 o'clock, November 17, 1858. This is the Smithsonian base date and time for the astronomical calendar.

All the time values passed to system services must also be in 64-bit format. A time value can be expressed as:

- An absolute time, which is specific date and time of day. Absolute times are always positive values
- A delta time, which is a future offset (number of hours, minutes, seconds, and so on) from the current time. Delta times are always expressed as negative values

Time conversion services:

- Obtain the current date and time in an ASCII string or in system format
- Convert an ASCII string into the system time format
- Convert a system time value into an ASCII string
- Convert the time from system format to integer values

Listed below are the Timer and Time Conversion System Services.

Get Time—$GETTIM

The Get Time system service furnishes the current system time in 64-bit format. The time is maintained in 100-nanosecond units from the system base time.

The current time can be obtained in system format with the Get Time ($GETTIM) system service, which places the time in a quadword buffer. For example:

```
TIME:           .BLKQ            1               ;BUFFER FOR TIME
                .
                .
                $GETTIME-S       TIMADR=TIME     ;GET TIME
```

This call to $GETTIM returns the current date and time system format in the quadword buffer TIME.

Convert Binary Time to Numeric Time—$NUMTIM

The Convert Binary Time to Numeric Time system service converts an absolute or delta time from 64-bit system time format to binary integer date and time values.

Convert Binary Time to ASCII String—$ASCTIM
The Convert Binary Time to ASCII String ($ASCTIM) system service converts a time in system format to an ASCII string and returns the string in a 23-byte buffer. To obtain the current time in ASCII, code the $ASCTIM system service as follows:

```
ATIMENOW:                                            ;DESCRIPTOR FOR ASCII TIME
                .LONG              23                ;LENGTH OF BUFFER
                .LONG              ATIMENOW8         ;ADDRESS OF BUFFER
                                   .BLKB             ;23 BYTES TO HOLD TIME

                $ASCTIM-S          TIMBUF=ATIMENOW        ;GET CURRENT TIME
```

The string returned by the service in the buffer ATIMENOW has the format:

dd-mmm-yyyy hh:mm:ss.cc

dd is the day of the month, mmm is the month (a 3-character alphabetic abbreviation), yyyy is the year, and hh:mm:ss.cc is the time in hours, minutes, seconds, and hundredths of seconds.

Convert ASCII String to Binary Time—$BINTIM
The converse of the $ASCTIM system service is the Convert ASCII String to Binary Time ($BINTIM) system service. The user provides the service with the time in ASCII format, and the service converts the string to a time value in 64-bit format suitable for input to the Set Timer ($SETIMR) or Schedule Wakeup ($SCHDWK) system services.

When the user omits any of the fields in the ASCII string buffer, the service uses the current date or time value for the field. Thus, to code a date-independent timer request, the input buffer for the $BINTIM system service would appear as illustrated in the example below. The two hyphens and at least a single blank space must precede the time field.

```
ANOON:          .ASCID             /--12:00.00/      ;ASCII 12 NOON
BNOON:          BLKQ               1                 ;BUFFER FOR BINARY 12
                .
                .
                .
                $BINTIM-S          TIMBUF=ANOON,TIMEADR=BNOON        ;CONVERT TIME
```

When the $BINTIM service completes, a 64-bit time value representing "noon today" is returned in the quadword at BNOON.

The $BINTIM system service also converts ASCII strings to delta time values to be used as input to timer services. The buffer for delta time ASCII strings has the format:

ddd hh:mm:ss.cc

The first field, indicating the number of days, must be specified as 0 if coding a "today" delta time.

The following example shows how to use the $BINTIM service to obtain a delta time in system format.

```
ATENMIN:     DESCRIPTOR       <0 00:10:00.00>     ;ASCII TEN MINUTES
BTENMIN:
             .BLKQ            1                   ;BUFFER FOR BINARY TEN
                                                  ;MINUTES
             .
             .
             .
             $BINTIM-S        TIMBUF=ATENMIN,TIMADR=BTENMIN       ;CONVERT
TIME
```

Set Timer—$SETIMR
The Set Timer system service allows a process to schedule setting an event flag and/or queuing an AST at some future time. The time for the event can be specified as an absolute time or as a delta time.

Cancel Timer Request—$CANTIM
The Cancel Timer Request system service cancels all or a selected subset of the Set Timer requests previously issued by the current image executing in a process. Cancellation is based on the request identification specified in the Set Timer ($SETIMR) system service. If more than one timer request was given with the same request identification, they are all canceled.

Schedule Wakeup—$SCHDWK
The Schedule Wakeup system service schedules the awakening of a process that has placed itself in a state of hibernation with the Hibernate ($HIBER) system service. A wakeup can be scheduled for a specified absolute time or for a delta time. Optionally, the request can specify that the wakeup is to be repeated at fixed intervals.

Cancel Wakeup—$CANWAK
The Cancel Wakeup system service removes all scheduled wakeup requests for a process from the timer queue, including those made by the caller or by other processes. Scheduled wakeup requests are made with the Schedule Wakeup ($SCHDWK) system service.

Set System Time—$SETIME
The Set System Time service allows users with operator (OPER) and logical I/O (LOGIO) privileges to set the current system time. The user can specify a new time or can recalibrate the current system time using the hardware time-of-year clock. This service might be used, for example, to synchronize two processors or to adjust to or from daylight savings time.

CONDITION HANDLING SERVICES

A condition handler is a procedure that is given control when an exception occurs. An exception is an event that is detected by the hardware or software and that interrupts the execution of an image. Examples of exceptions include arithmetic overflow or underflow and reserved opcode or operand faults.

If the user determines that a program needs to be informed of particular exceptions so that it can take corrective action, the user can code and specify a condition handler. This condition handler, which will receive control when any exception occurs, can test for specific exceptions.

If an exception occurs and a condition handler has not been specified, the default condition handler established by the command interpreter is given control. If the exception is a fatal error, the default condition handler issues a descriptive message and performs an exit on behalf of the image that incurred the exception.

Listed below are the Condition Handling Services.

Set Exception Vector—$SETEXV

The Set Exception Vector system service assigns a condition handler address to an exception vector or cancels an address previously assigned to a vector.

Set System Service Failure Exception Mode—$SETSFM

This system service controls whether a software exception is generated when an error or severe error status code is returned from a system service call. Initially, system service failure exceptions are disabled; the caller should explicitly test for successful completion following a system service call.

Unwind Call Stack—$UNWIND

The Unwind Call Stack system service allows a condition handling routine to unwind the procedure call stack to a specified depth. Optionally, a new return address can be specified to alter the flow of execution when the topmost call frame has been unwound.

Declare Change Mode or Compatibility Mode Handler—$DCLCMH

Declare Change Mode or Compatibility Mode Hander ($DCLCMH) system service establishes the address of a routine to receive control when a Change Mode to User or Change Mode to Supervisor instruction trap occurs, or a compatibility mode fault occurs.

MEMORY MANAGEMENT SERVICES

The VAX/VMS memory management routines map and control the relationship between physical memory and a process's virtual address space. These activities are, for the most part, transparent to the user and user programs. However, in some cases the user may make the program more efficient by explicitly controlling its virtual memory usage. Memory Management services allow the user to:

- Increase or decrease the virtual address space available in a process's program or control region
- Control the process's working set size and the swapping of pages between physical memory and the paging device
- Define disk files containing data or shareable images and map the file into the process's virtual address space

Listed below are the Memory Management Services.

Expand Program/Control Region—$EXPREG

The Expand Program/Control Region system service adds a specified number of new virtual pages to a process's program region or control region for the execution of the current image. Expansion occurs at the current end of that region's virtual address space.

For example, if the user desires to add four pages to a process's program region, the call to the $EXPREG system service is coded as follows:

```
SPACE:
            .BLKL               2       ;.RETURN START AND END OF NEW PAGES
            .
            .
            $EXPREG-S           PAGCNT=#4,RETADR=SPACE,REGION=#0     ;GET    4
PAGES
```

- PAGCNT is the argument denoting the number of pages to be added
- RETADR is the argument receiving the starting and ending virtual addresses of added pages
- REGION is the argument denoting which region is to be expanded. A value of 0 indicates program region (P0) and a value of 1 indicates control region (P1)

Therefore, to add the same number of pages to the control region, the user would specify REGION = #1.

Contract Program/Control Region—$CNTREG

The Contract Program/Control Region system service deletes a specified number of pages from the current end of the program or control region of a process's virtual address space. The deleted pages become inaccessible; any references to them cause access violations.

The following example shows four pages being deleted from the program (P0) region:

```
$CNTREG_S PAGCNT=#4,REGION=#0
```

- PAGCNT is the argument denoting the number of pages to be deleted
- REGION is the argument specifying from which region the pages are to be deleted

Create Virtual Address Space—$CRETVA
The Create Virtual Address Space system service adds a range of pages to a process's virtual address space for the execution of the current image or until a $DELTVA is issued for the pages.

Delete Virtual Address Space—$DELTVA
The Delete Virtual Address Space system service deletes a range of addresses from a process's virtual address space. Upon successful completion of the service, the deleted pages are inaccessible; any references to them cause access violations.

Create and Map Section—$CRMPSC
The Create and Map Section system service creates and/or maps a section. A section can be a disk file section or a page frame section. A disk file section is data or code from a disk file that can be brought into memory and made available, either only to the process that creates it (private section) or to all processes that map to it (global section). A page frame section consists of one or more physical page frames in memory or I/O space.

Creating a disk file secton involves defining all or part of a disk file as a section. Mapping a disk file section involves making a correspondence between virtual blocks in the file and pages in the caller's virtual address space. If the $CRMPSC service specifies a global section that already exists, the service maps it.

Map Global Section—$MGBLSC
The Map Global Section system service provides a process with access to an existing global section. Mapping a global section establishes the correspondence between pages in the process's virtual address space and the physical pages occupied by the global section.

Update Section File on Disk—$UPDSEC
The Update Section File on Disk system service writes all modified pages in an active private or global section back into the section file on

disk. One or more I/O requests are queued, based on the number of pages that have been modified.

Delete Global Section—$DGBLSC
The Delete Global Section system service marks an existing permanent global section for deletion. The actual deletion of the global section takes place when all processes that have mapped the global section have deleted the mapped pages.

Lock Pages in Working Set—$LKWSET
The Lock Pages in Working Set system service allows a process to specify that a group of pages that are heavily used should never be replaced in the working set. The specified pages are brought into the working set if they are not already there and are locked so that they do not become candidates for replacement.

Unlock Pages From Working Set—$ULWSET
The Unlock Pages from Working Set system service allows a process to specify that a group of pages that were previously locked in the working set are to be unlocked and become candidates for page replacement like other working set pages.

Purge Working Set—$PURGWS
The Purge Working Set system service enables a process to remove pages from its current working set to reduce the amount of physical memory occupied by the current image.

Lock Pages in Memory—$LCKPAG
The Lock Pages in Memory system service locks a page or range of pages in memory. The specified virtual pages are forced into the working set and then locked in memory. A locked page is not swapped with its working set. These pages are not candidates for page replacement and in this sense are locked in the working set as well.

Unlock Page From Memory—$UNLPAG
The Unlock Pages from Memory system service releases the page lock on a page or range of pages previously locked in memory by the Lock Pages in Memory ($LCKPAG) system service.

Adjust Working Set Limit—$ADJWSL
The Adjust Working Set Limit system service changes the current limit of a process's working set size by a specified number of pages. This service allows a process to control the number of pages resident in physical memory for the execution of the current image.

Set Protection on Pages—$SETPRT
The Set Protection on Pages system service allows an image running in a process to change the protection on a page or range of pages.

Set Process Swap Mode—$SETSWM
The Set Process Swap Mode system service allows a process to control whether it can be swapped out of the balance set. Once a process is locked in the balance set, it cannot be swapped out of memory until it is explicitly unlocked.

CHANGE MODE SERVICES
The Change Mode system services allow a process to change to either executive mode or kernel mode to execute a specified routine. Use of these services requires privilege.

Change To Executive—$CMEXEC
The Change to Executive Mode system service allows a process to change its access mode to executive, execute a specified routine, and then return to the access mode in effect before the call was issued.

Change to Kernel Mode—$CMKRNL
The Change to Kernel Mode system service allows a process to change its access mode to kernel, execute a specified routine, and then return to the access mode in effect before the call was issued.

Adjust Outer Mode Stack Pointer—$ADJSTK
The Adjust Outer Mode Stack Pointer system service modifies the stack pointer for a less privileged access mode. This service is used by the operating system to modify a stack pointer for a less privileged access mode after placing arguments on the stack.

LOCK MANAGEMENT SERVICES

The VMS Lock Management Services are a tool to help users develop complex resource-sharing applications; for example, database systems. it provides a resource nametable for defining a resource, a variety of lock modes for controlling access to it, and the means for processes to queue lock requests.

The resource nametable is tree structured and allows the user to define their resource to practically any granularity or hierarchical depth.

There are six lock modes available.

- Null Lock (LCK$K-NLMODE)
- Concurrent Read (LCK$K-CRMODE)
- Concurrent Write (LCK$K-CWMODE)

- Protected Read (LCK$K-PRMOD)
- Protected Write (LCK$K-PWMODE)
- Exclusive (LCK$K-EXMODE)

If a lock request is made on a resource, and another process already has an incompatible lock on that resource, the lock request is queued until the resource is unlocked or the lock has been changed to a compatible one.

A process may have more than one lock at one time. The limit on the number of locks depends on the quota assigned to the process.

For more information about the Lock Management Services, particularly about lock compatibility and how this service can be applied, refer to chapter 14.

Enqueue Lock Request - $ENQ
The Enqueue Lock Request system service allows users to queue requests to access a resource or to convert the current lock request mode to another lock request mode.

An Enqueue Lock Request must specify the type of lock mode and, if it is a new lock request (not a convert lock request), the resource name. The options available to a procedure for synchronizing with the Lock Management Service are the same as with the QIO system service; that is:

- wait for a specified event flag to be set
- specify the address of an AST routine to be executed when the request is granted
- poll the lock status block for a lock-granted status

Enqueue Lock Request and Wait for Event Flag - $ENQW
The $ENQW system service combines the Enqueue Lock Request ($ENQ) and Wait for Single Event Flag ($WAITFR) system services. It may be used when a program must wait until the requested lock has been granted.

Dequeue Lock Request - $DEQ
The $DEQ system service is used to dequeue locks that the calling process had previously queued. All locks can be dequeued, whether granted or waiting, new or conversion.

CHAPTER OVERVIEW

Input and output services require a complex management system; otherwise the user is left with the task of producing detailed I/O control for each process. Under the VAX/VMS operating system, complete I/O services are provided for handling, controlling, and queueing I/O needs or requests. VAX-11 RMS (Record Management Services) gives users a wide range of file management techniques while remaining transparent. This chapter investigates the I/O services of the VAX software.

Topics include:

- Programming Interfaces
- Ancillary Control Processes (ACP)
- I/O Request Processing
- Queue I/O (QIO)
- I/O Completion
- Record Management Services (RMS)

CHAPTER 12
INPUT/OUTPUT SERVICES

INTRODUCTION

The VAX/VMS operating system supports a wide variety of input and output devices, including disks, magnetic tapes, lineprinters, and card readers. Input/output operations are extremely flexible and as device- and function-independent as possible.

Processes issue I/O requests to channels which have been previously associated with particular physical device units. A **channel** is a logical path through the system, connecting the user process with a predetermined physical I/O device unit. Each process is able to establish its own communication between physical devices and channels. I/O requests are queued by priority, first-in/first-out within priority, and then processed strictly in queue order.

RMS and QIO

I/O requests can be handled indirectly through the use of an established set of procedures, such as VAX-11 RMS (Record Management Services), or they can be interfaced directly to an I/O driver by means of a QIO request. The principal feature of the VAX-11 RMS software is its ease of use and device independence. Generally it is used for I/O requests to mass storage devices, while the more direct—and complicated—QIO is for specialized use of terminals, special devices (e.g., graphics and special communications equipment), and highly specialized formatting.

Figure 12-1 represents an overview of the major I/O processing system components and user relationships.

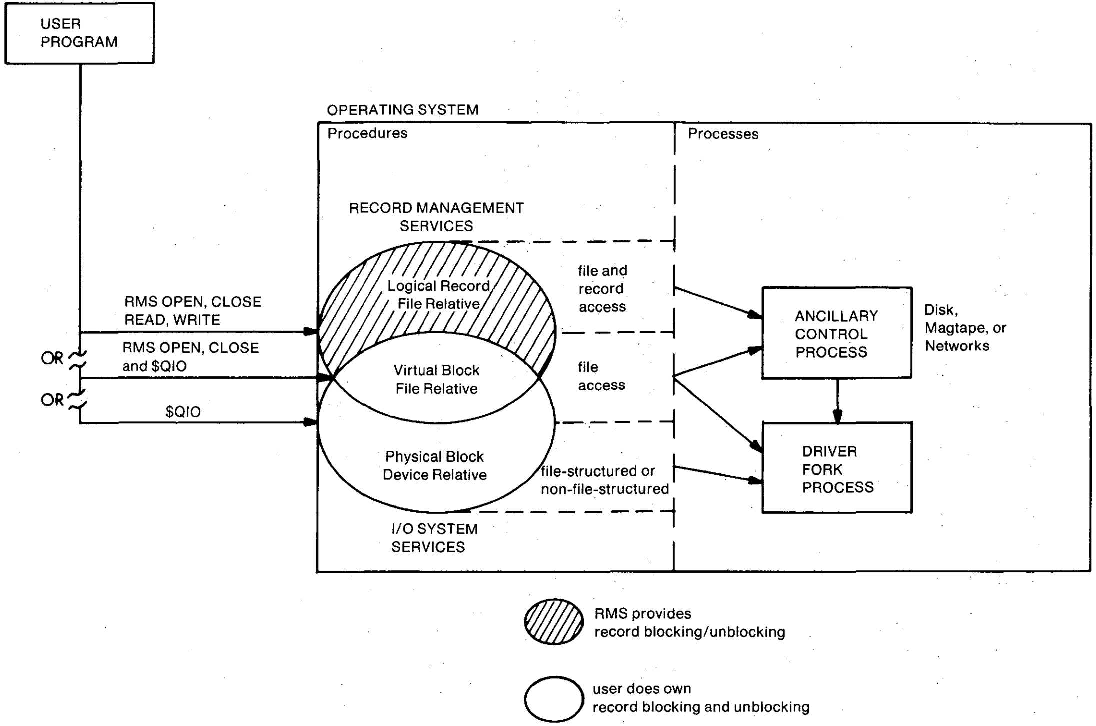

USER PROGRAM
OPERATING SYSTEM
Procedures
Processes
RECORD MANAGEMENT SERVICES
Logical Record File Relative
Virtual Block File Relative
Physical Block Device Relative
I/O SYSTEM SERVICES
RMS OPEN, CLOSE READ, WRITE
OR
RMS OPEN, CLOSE and $QIO
OR
$QIO
file and record access
file access
file-structured or non-file-structured
ANCILLARY CONTROL PROCESS
Disk, Magtape, or Networks
DRIVER FORK PROCESS
RMS provides record blocking/unblocking
user does own record blocking and unblocking

Figure 12-1 User Interfaces to I/O Services

PROGRAMMING INTERFACES

The I/O programming tools are: the Record Management services (RMS)—for general purpose file and record processing—and the I/O system services—for direct I/O processing. Table 12-1 summarizes the programming interfaces.

Table 12-1 I/O Programming Interfaces

Method	Program Interface	I/O Components	Purpose
Record I/O	RMS requests	RMS, ACP and Driver	Use Files-11 disk or ANSI magtape file structure, device-independent I/O, use RMS record access methods
File I/O	RMS OPEN and $QIO requests	RMS for OPEN, ACP and Driver	Use Files-11 disk or ANSI magtape file structure, implement own record access methods
Device I/O	$QIO requests	Driver	Fast dumps to disk or magnetic tape, foreign file structure

RMS procedures provide device-independent, file-structured access to all types of I/O peripherals. The most general purpose type of access enables programs to process logical records, where RMS software automatically provides record blocking and unblocking.

RMS users can also choose to perform their own record blocking on file-structured volumes such as disk and magnetic tape, either to control buffer allocation or to optimize special record processing.

The I/O system services provide both device-independent and device-dependent programming. Users can perform their own record blocking on file-structured and non-file-structured devices. In addition, users with sufficient privilege can perform I/O operations using either logical or physical I/O requests, for example, to define their own file structures and accessing methods on disk and magnetic tape volumes.

ANCILLARY CONTROL PROCESSES

I/O control processes, called ancillary control processes (ACPs), process file-structured I/O requests. An ACP provides file structuring and volume access control for a particular type of device. There are three types of ACPs provided in the system: Files-11 disk, ANS (American National Standard) magnetic tape, and DECnet (network) communications link.

The RMS and I/O system services programming interfaces are the same regardless of the ACP involved. However, since ACPs are particular for a device type, they do not have to be present in the system if the device is not present. There is one network ACP process for all DECnet network communications links in the system, and none if the system is not in a network. For either disk or magnetic tape devices, the system manager can install one ACP per volume for throughput, or one ACP for all volumes, to save space.

DEVICE DRIVERS

Once the ACP sets up the information for file-structured I/O requests, a request can be passed to a device driver. All non-file-structured I/O requests are passed directly to a device driver. Drivers also perform all the hardware retry and recovery operations.

To incur the least overhead, driver processes are created dynamically when a user makes an I/O request for a device or a device generates an unsolicited interrupt. They have minimal context, execute to completion when created, and are memory-resident throughout execution. One driver process is created for each device unit in the system. All driver processes for the same device type share the code they execute.

I/O REQUEST PROCESSING

All I/O requests pass through a Queue I/O (QIO) Request system service. If a program requests RMS procedures, RMS issues the Queue I/O Request system service on the program's behalf. Queue I/O Request processing is extremely rapid because the system can keep each device unit as busy as possible by minimizing the code that must be executed to initiate requests and post request completion.

The processor's many interrupt priority levels improve interrupt response because they enable the software to have the minimum amount of code executing at high priority levels by using low priority levels for code handling request verification and completion notification. In addition, device drivers take advantage of the processor's ability to overlap execution with I/O by enabling processes to execute between the initiation of a request and its completion. User processes can queue requests to a driver at any time, and the driver immediately initiates the next request in its queue upon receiving an I/O completion interrupt.

All access validation and checking takes place before an I/O request is actually queued. For file-structured I/O requests, the Queue I/O Request system service obtains all the block mapping and volume access checking information from the ancillary control process (ACP). For example, on I/O requests for multivolume files, the system service obtains mapping information from the ACP. This enables it to queue requests to different drivers when the user's I/O request involves a transfer that spans volumes. The Queue I/O Request system service also checks the validity of the function requested (read, write, rewind, etc.) for the particular device. Because all access validation and function checking is performed before the request is queued, the driver has little to do to initiate a request.

Once the system service has verified the I/O request, it raises the interrupt priority level to that of the driver. The only activity it has to perform at this level is a test to see if the driver is busy. If the driver is not busy, it calls the driver. Otherwise, it queues the request according to the priority of the requesting process and immediately returns to the user process. When the driver is called, it initiates the request and returns to the user process.

At the time the device subsequently generates its interrupt at the hardware interrupt priority level, the interrupt dispatcher calls the appropriate interrupt service routine. An interrupt service routine simply saves the device control/status registers, requests a software interrupt at the driver's interrupt priority level, and returns to the interrupt dispatcher, which is then free to scan for unit attentions. Because a disk controller cannot generate interrupts on any unit performing a seek until the current transfer completes, the interrupt dispatcher will also dispatch seek completion when dispatching a disk I/O transfer completion interrupt.

When the driver receives the completion interrupt, it prepares the I/O completion status for the requester, and requests a software interrupt. The driver is then free to process another request in its queue and, if the queue is not empty, the driver begins again. All I/O completion notification takes place outside the driver, minimizing the inter-request idle time. The I/O post routine notifies the process of I/O completion and releases or unlocks the buffer.

QUEUE I/O

Queue I/O is the interface by which the user interacts directly with the I/O driver.

Assigning Channels

A channel is a communication path that is associated with a physical device unit during VAX/VMS I/O operations. It is used by a process in the transfer of information to and from the device. Before any I/O operations can be requested for a device, the device must be assigned to an I/O channel by the Assign I/O Channel ($ASSIGN) system service.

In coding a call to the $ASSIGN service, the name of the device (real device name or logical name) and the address of the longword to receive the channel number must be supplied. The channel number, which is returned by the service, is then referred to when coding an I/O request.

Physical, Logical, and Virtual I/O

I/O transfers can take place in three possible modes of operation: physical, logical, and virtual I/O functions.

Physical I/O concerns reading and writing data in the actual physical units accepted by the hardware, for example, sectors on a disk. This function mode allows access to all device level I/O operations.

Logical I/O concerns reading and writing data in blocks that usually could map directly into physical blocks. For block-structured devices—disks, for example—logical blocks are numbered starting at zero (0).

Virtual I/O consists of file-oriented operations—creating files and reading and writing files, for example. In this case, the VAX/VMS operating system maps virtual block numbers into logical block numbers. For file-structured devices—disks, for example—virtual blocks are the same size as logical blocks. They are numbered starting at one (1) and are relative to the file rather than to the device. On non-file-structured devices, virtual I/O is equivalent to logical I/O; mapping from virtual block number to logical block number is direct.

Issuing I/O Requests

VAX/VMS I/O function requests are issued via the Queue I/O Request ($QIO) system service. Prior to issuing such a request, the I/O channel must be assigned to the selected device through the use of the Assign I/O Channel ($ASSIGN) system service. To effect I/O operations on the device, subsequent calls to the Queue I/O Request system service must specify the channel number returned by the Assign I/O Channel system service.

The Queue I/O Request system service can be performed only on assigned I/O channels and only from access modes that are equal to or more privileged than the access mode from which the original channel assignment was made.

Certain requirements must be met before a request is queued. For example, a valid channel number must be included in the request; the request must not exceed certain process quotas; and there must be sufficient dynamic memory available to complete the operation.

After an I/O request has been queued, the system does not require the issuing process to wait for the operation to complete. If at any time the user process which issued the QIO request cannot proceed until the I/O operation is completed, an event flag can be used to synchronize I/O completion. The process should specify an event flag in the QIO request and should issue a $WAITFR (Wait for Single Event Flag) system service request at the point where synchronization is required.

I/O COMPLETION

The successful or unsuccessful completion of an I/O request can be denoted by one or more return conditions. Selection of return conditions depends on the arguments included in the QIO macro call.

There are three primary returns:

- Event flag
- I/O status block
- Asynchronous system trap

Event Flags

Event flags are status-posting bits used by some I/O system services to indicate the completion or the occurrence of an event. The QIO system service sets an event flag when it completes an input or output operation. Event flag services provide the techniques that allow the user to set or clear specific flags, test the current status of flags, or place a program in a wait state pending the setting of a particular flag or group of flags.

I/O Status Block

The completion status of the I/O request is returned in the I/O status block (IOSB).

The IOSB indicates whether or not the operation was successfully completed, the number of bytes transferred, and additional device-dependent return information.

Asynchronous System Traps
An asynchronous system trap (AST) routine can optionally be specified in the QIO request if the user wants to interrupt a process to execute special code on completion of the request. When the I/O operation completes, control branches to the AST service routine. The AST service routine is then executed at the access mode from which the QIO service was requested. Using an AST to signal I/O completion allows the process to be occupied with other functions during the I/O operation. The process does not have to wait until some event occurs before proceeding to another operation.

RECORD MANAGEMENT SERVICES

A powerful, transparent collection of routines, Record Management Services (RMS) provides extensive capabilities for data storage, retrieval, and modification. Complex file manipulation is easily achieved through RMS facilities. Users may select from several file organizations and file access techniques—each of which is suited to particular applications—from the simplest sequential search of a sequentially organized file to a sophisticated keyed access of an indexed file based on several alternate key fields.

The three file organizations supported by Record Management Services—sequential, relative, and indexed—are variously available to three different access modes—sequential, keyed, and Record's File Address. In most cases, RMS software supports dynamic access, a useful feature that allows access mode switching within a process.

NOTE

> Most RMS functionality is also available to users of DECnet commmunications software, DIGITAL's networking architecture. For details, see Chapter 7 of this Handbook.

RMS FILE ORGANIZATIONS

A file is a collection of related information. For example, a file might contain a company's personnel information (employee names, addresses, job titles). Within this file, the information is divided into records. All the information on a single employee could constitute a single record.

Each record in the personnel file would itself be divided into discrete pieces of information known as fields. The user defines the number, locations within the record, and logical interpretations of these fields. The name of an employee would be a field in his personnel record, as would a wage class or a social security number.

The user can completely control the grouping of fields into records and records into files. Programs either build records and pass them to RMS for storage in a file, or issue requests for records while RMS performs the necessary operations to retrieve the records from a file.

Table 12-2 File Organizations—Advantages and Disadvantages

Sequential	**Advantages**—Uses disk and memory efficiently: minimum disk overhead and block-boundary crossing. Provides optimal usage if the application accesses all records sequentially on each run. Provides the most flexible record format. Allows data to be stored on many different types of media, in a device-independent manner. Allows easy file extension
	Disadvantages—Some high-level languages allow sequential access only. Allows records to be added only to end of file. Allows write access by multiple, concurrent users, but only in very restricted cases
Relative	**Advantages**—Allows both sequential and random access for all languages. Provides random record deletion and insertion. Allows records to be read- and write-shared
	Disadvantages—Allows data to be stored on disk only. Requires that files contain a record cell for each relative record number allocated; that is, files may not be densely populated. Requires that record cells be the same size

Indexed	**Advantages**—Allows sequential and random access by key value for all languages. Allows random record deletion and insertion. Allows records to be read- and write-shared. Allows variable-length records to change length on update. Allows easy file extension **Disadvantages**—Allows data to be stored on disk only. Requires more disk space. Uses more of the central processing unit to process records. Generally requires multiple disk accesses to randomly process a record

Sequential File Organization
In sequential file organization, records appear in consecutive sequence. The order in which records appear is always the order in which the records were originally written to the file by an application program. Figure 12-2 illustrates sequential file organization.

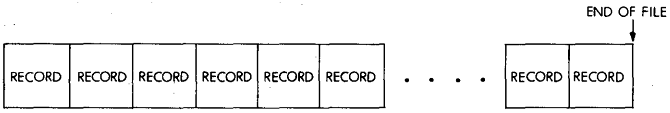


Figure 12-2 Sequential File Organization

Relative File Organization
When relative organization is selected, Record Management Services structures a file as a series of fixed-size record cells. Cell size is based on the size specified as the maximum permitted length for a record in the file. These cells are numbered from 1 (the first) to n (the last). A cell's number represents its location relative to the beginning of the file.

Each cell in a relative file can contain a single record. There is no requirement, however, that every cell contain a record. Empty cells can be interspersed among cells containing records. Figure 12-3 illustrates a relative file organization.

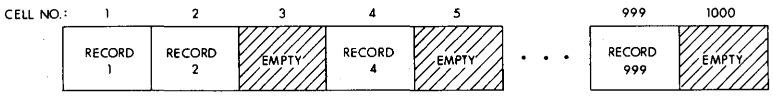


Figure 12-3 Relative File Organization

Because cell numbers in a relative file are unique, they can be used to identify both a cell and the record (if any) occupying that cell. Thus, record number 1 occupies the first cell in the file, record number 17 occupies the seventeenth cell, and so on. When a cell number is used to identify a record, it is also known as a relative record number.

Indexed File Organization

The location of records in indexed file organization is transparent to the program. Record Management Services completely controls the placement of records in an indexed file. The presence of keys in the records of the file governs this placement.

A key is a byte string present in every record of an indexed file. Any of the six RMS keyfield data types may be used as a key: 1) character string; 2) signed 15-bit integer; 3) unsigned 16-bit binary; 4) signed 31-bit integer; 5) unsigned 32-bit binary; 6) packed decimal. Unique among file organizations, indexed files can be accessed by data in the files, rather than by addresses. The location and length of this key are identical in all records. When creating an indexed file, the user decides which byte string in the file's records is to be a key. Selecting such a byte string indicates to RMS that the contents (i.e., key value) of that string in any particular record written to the file can be used by a program to identify that record for subsequent retrieval. Frequently, the byte string chosen as the key is one of the fields already defined in the record. Non-numeric entries (eg., names, job descriptions) are coded internally in a manner that is equivalent to alphabetization.

At least one key, the primary key, must be defined for an indexed file. Optionally, additional keys or alternate keys can be defined. An alternate key value can also be used as a means of identifying a record for retrieval.

As processes write records into an indexed file, Record Management Services (RMS) builds a tree-structured table known as an index. An index consists of a series of entries containing a key value copied from a record that a program wrote into the file. Along with each key value is a pointer to the location in the file of the record from which the value was copied. RMS builds and maintains a separate index for each key defined for the file. Each index is stored in the file. Thus, every indexed file contains at least one index, the primary key index. Figure 12-4 illustrates an indexed file organization with a primary key. When alternate keys are defined, RMS builds and stores an additional index for each alternate key.

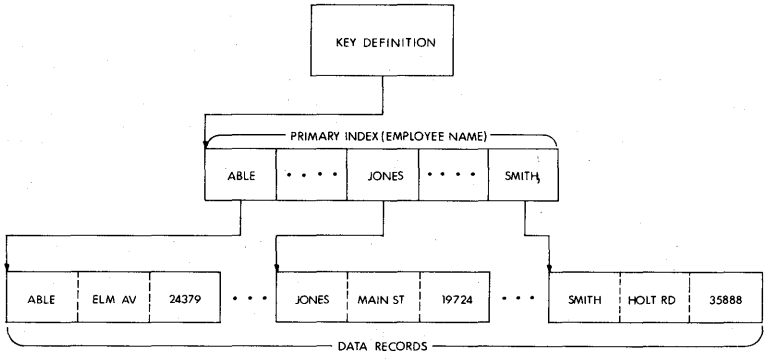


Figure 12-4 Indexed File Organization

RMS RECORD ACCESS MODES

The methods of retrieving and storing records in a file are called record access modes. A different record access mode can be used to process records within the file each time it is opened. A program can also change access mode during the processing of a file.

Sequential Record Access Mode

Sequential record access means that records are retrieved or written in the sequence established by the organization of the file. Sequential record access mode can be used to access all RMS files and all record-oriented devices, including mailboxes.

Sequential Record Access to Sequential Files In a sequentially organized file, physical arrangement establishes the order in which records are retrieved when using sequential access mode. To read a particular record in a file, say the fifteenth record, a program must open the file and access the first fourteen records before accessing the desired record. Thus each record in a sequential file can be retrieved only by first accessing all records that physically precede it.

When writing new records to a sequential file in sequential access mode, a program must first request that RMS position the file immediately following the last record. Then each sequential write operation the program issues causes a record to be written following the previous record.

Sequential Record Access to Relative Files During the sequential access of records in the relative file organization, the contents of the record cells in the file establish the order in which a program processes records. RMS recognizes whether successively numbered record cells are empty or contain records.

When a program issues read requests in sequential access mode for a relative file, RMS ignores empty record cells and searches successive cells for the first one containing a record. When a program adds new records in sequential access mode to a relative file, RMS places a record in the cell whose relative number is one higher than the relative number of the previous request, as long as that cell does not already contain a record. RMS allows a program to write new records only into empty cells in the file.

Sequential Record Access to Indexed Files A program can use the sequential record access mode to retrieve records from an indexed file in the order represented by any index. The entries in an index are arranged in ascending order by key values. If more than one key is defined for the file, each separate index associated with a key represents a different logical ordering of the records in the file.

When reading records in sequential record access mode from an indexed file, a program initially specifies a key (primary key, first alternate key, second alternate key, etc.) to RMS. Thereafter, RMS uses the index associated with that specified key to retrieve records in the sequence represented by the entries in the index. Each successive record RMS returns in response to a read request contains a value in the specified key field that is equal to or greater than that of the previous record returned.

When writing records to an indexed file, RMS uses the definition of the primary key field to place the record in the file.

Random Record Access Mode

In random access mode, the program establishes the order in which records are processed. Each program request for access to a record operates independently of the previous record accessed. Each request in random mode identifies the particular record of interest. Successive requests in random mode can identify and access records anywhere in the file.

Random Record Access to Sequential Files Native programs can access sequential files on disk using relative record number to randomly locate a record, provided that the records are in fixed-length record format.

Random Record Access to Relative Files Programs can read or write records in a relative file by specifying the relative record number. RMS interprets each number as the corresponding cell in the file. A program can read records at random by successively requesting, for example, record number 47, record number 11, record number 31, and so forth. If no record exists in a specified cell, RMS returns a nonexistence indicator to the requesting program. Similarly, a program can store records in a relative file by identifying the cell in the file that a record is to occupy. If a program attempts to write a new record in a cell already containing a record, RMS returns a record-already-exists indicator to the program.

Random Record Access to Indexed Files For indexed files, a key value rather than relative record number identifies the record. Each program read request in random access mode specifies a key value and the index (primary index, first alternate index, second alternate index, etc.) that RMS must search. When RMS finds the key value in the specified index, it reads the record that the index entry points to and passes the record to the user program.

Program requests to write records randomly in an indexed file do not require the separate specification of a key value. All key values (primary and, if any, alternate key values) are in the record itself. When an indexed file is opened, RMS retrieves all definitions stored in the file. RMS knows the location and length of each key field in a record. Before writing a record into the file, RMS examines the values contained in the key fields and creates new entries in the indices. In this way RMS ensures that the record can be retrieved by any of its key values. The process by which RMS adds new records to the file is precisely the process it uses to construct the original index or indices.

Record's File Address (RFA) Access Mode

Record's File Address (RFA) access mode can be used to retrieve records in any file organization as long as the file resides on a disk volume. RFA access allows a specific record to be identified for retrieval, using the record's unique address. The actual format of this address depends on the organization of the file. In all instances, however, only RMS can interpret this format.

After every successful read or write operation, RMS returns the RFA of the subject record to the program. The program can then save this

RFA to use again to retrieve the same record. This is an optimizing feature that can greatly speed up record access in RFA mode. It is not required that this RFA be used only during the current execution of the program. RFAs can be saved and used at any subsequent time.

Dynamic Access

Dynamic access is not strictly an access mode. It is the ability to switch from one access mode to another while processing a file. For example, a program can access a record randomly, then switch to sequential access mode for processing subsequent records. There is no limitation on the number of times such switching can occur. The only limitation is that the file organization must support the access mode selected.

FILE AND RECORD ATTRIBUTES

When an RMS file is created, its physical characteristics or attributes must be defined. These characteristics are defined by source language statements in an application program or by an RMS utility. The program or user assigns the file a name, the owner's user identification code, and a protection code, and selects the file organization. Other attributes are also selected, including:

- Device
- File size
- File location
- Record format and size
- Keys (for indexed files only)

Device selection is related to the organization of the file. Sequential files can be created on Files-11 disk volumes or ANSI magnetic tape volumes. Sequential files can also be read from mailboxes, terminals, and card readers, and written to mailboxes, terminals, and lineprinters. Relative and indexed files can be created on Files-11 disk volumes.

The logical limit on file size is $2^{31}-1$ blocks, with a more realistic limit being the volume set on which a file can reside. When creating an RMS file on a disk volume, the user can specify an initial allocation size. If no file size is given, Record Management Services (RMS) allocates the minimum amount of storage needed to contain the defined attributes of the file. The initial size can be extended dynamically. The user can let RMS locate the file, or the user can allocate the file at a specific location on the disk to optimize disk access time. The file's starting location can be specified optionally using a volume-relative block number or physical address (track and sector number with or without a given cylinder specification).

When creating a file on a magnetic tape volume, a user or program does not specify an initial allocation size. The blocks are simply written one after another down the tape, beginning after the last file, if any, already written on the tape. Once a tape file has been created, another file can replace it or be appended to it, but all subsequent files on the tape, if any, are lost.

Record Formats

The user provides the specifications for the records the file will contain. The specified format establishes how each record appears in the file. There are four avaiable record formats:

- fixed length
- variable length
- variable with fixed-length control (not for indexed files)
- stream (for sequential files only)

Fixed length record format refers to records of a file that are all equal in size. Each record occupies an identical amount of space in the file. All file organizations support fixed length record format.

Variable-length record format records can be either equal or unequal in length. All file organizations support variable-length record format. RMS prefixes a count field to each variable-length record it writes. The count field describes the length (in bytes) of the record. RMS removes this count field before it passes a record to the program. RMS produces two types of count fields, depending on the storage medium on which the file resides.

Variable-length records in files on Files-11 disk volumes have a 2-byte binary count field preceding the data field portion of each record. The specified size excludes the count field.

Variable-length records on ANSI magnetic tapes have 4-character decimal count fields preceding the data portion of each record. The specified size includes the count field. In the context of ANSI tapes, this record format is known as D format.

Variable with fixed-length control records consist of two distinct parts, the fixed-length control area and a variable-length data record. Although stored together, the two parts are returned to the program separately when the record is read. The size of the fixed-length control area is identical for all records of the file. The contents of the fixed-length control area are completely under the control of the program and can be used for any purpose. For example, fixed-length control areas can be used to store the identifier (relative record number or

RFA) of related records. Indexed file organizations do not support this record format.

Stream record format records in a file are variable-length records delimited by the occurrence of special character sequences called terminators. Terminators are part of the record they delimit. No count fields or control information is stored in the file. This is supported for sequential disk files only.

Key Definitions for Indexed Files

To define a key for an indexed file, the user specifies the position and length of the key field in the records. At least one key, the primary key, must be defined for an indexed file. Additionally, up to 254 alternate keys can be defined. In general, most files have two or three keys. Because indices require storage space and Record Management Services (RMS) updates indices as records are added or modified, no more than six to eight keys should be defined where storage space or access time is important.

Each primary and alternate key represents from 1 to 255 bytes in each record of the file. RMS permits six keyfield data types:

- String (1 to 255 bytes of character data)
- Signed 15-bit integer
- Unsigned 16-bit binary
- Signed 31-bit integer
- Unsigned 32-bit binary
- Packed decimal (1 to 31 nibbles)

The string keyfield can be composed of simple or segmented keys. A simple key is a single, contiguous string of characters in the record, i.e., a single field. A segmented key, however, can consist of from two to eight fields within records. These fields need not be contiguous. When processing records that contain segmented keys, RMS treats the separate fields (segments) as a logically contiguous character string. The integer, binary, and packed decimal data types can be simple keys only.

When defining keys at file-creation time, two characteristics for each key can be specified:

- Duplicate key values are or are not allowed
- Key value can or cannot change

When duplicate key values are allowed, more than one record can have the same value in a given key. For example, the creator of a personnel file could define the department name field as an alternate key. As programs wrote records into the file, the alternate index for the department name key field would contain multiple entries for each key value (e.g., PAYROLL, SALES, ADMINISTRATION) since departments are composed of more than one employee. When such duplication occurs, RMS stores the records so that they can be retrieved in first-in/first-out (FIFO) order.

An application could be written to list the names of employees in any particular department. A single execution of the application could list the names of all employees working, for example, in the department called SALES. By randomly accessing the file by alternate key (with the key value SALES), the application would obtain the first record written into the file containing this value. Then, the application could switch to sequential record access and successively obtain records with the same value, SALES, in the alternate key field. Part of the logic of the application would be to determine the point at which a sequentially accessed record no longer contained the value SALES in the alternate key field. The program could then switch back to random record access mode and access the first record containing a different value (e.g., PAYROLL) in the department name key field.

If key values can change, records can be read and then written back into the file with a modified key value. For example, this specification would allow a program to access a record in the personnel file and change the contents of a department name field to reflect the transfer of an employee from one department to another. This characteristic can be specified only for alternate keys.

Program Operations on RMS Files

After Record Management Services (RMS) has created a file, a program can access the file and store and retrieve data.

When a program accesses the file as a logical structure (i.e., a sequential, relative, or indexed file), it uses record I/O operations such as add, update, and delete record. The organization of the file determines the types of record operations permitted.

If the record accessing capabilities of RMS are not used, programs can access the file as an array of virtual blocks. To process a file at this level, programs use a type of access known as block I/O.

File Processing

At the file level, before beginning record processing, a program can:

- Create a file
- Open an existing file
- Modify file attributes
- Extend a file

- Close a file
- Delete a file

Once a program has opened a file for the first time, it has access to the unique internal ID for the file. If the program intends to open the file subsequently, it can use that internal ID to open the file and avoid any directory search.

File Organization and Sharing — With the exception of magnetic tape files, which cannot be shared, every RMS file can be shared by any number of programs that are reading, but not writing, the file. Sequential files on disk can be accessed by a single writer or shared by multiple readers. Relative and indexed files, however, can be shared by multiple readers and multiple writers. A program can read or write records in a relative or indexed file while other programs are similarly reading or writing records in the file. Thus, the information in such files can be changing while programs are accessing them.

NOTE

RMS file sharing support is available for certain sequential files. Specifically, sequential files with 512 byte fixed-length records may be shared in the same ways as relative and indexed files.

Program Sharing — A file's organization establishes whether it can be shared for reading with a single writer or for multiple readers and writers. A program specifies whether such sharing actually occurs at runtime. The user controls the sharing of a file through information the program provides Record Management Services (RMS) when it opens the file. First, a program must declare what operations (e.g., read, write, delete, update) it intends to perform on the file. Second, a program must specify whether other programs can read the file or both read and write the file concurrently with the first program.

The combination of these two types of information allows RMS to determine if multiple user programs can access a file at the same time. Whenever a program's sharing information is compatible with the corresponding information another program provides, both programs can access the file concurrently.

Record Locking — RMS can lock records to control operations to a relative or indexed file that more than on record steam within a process, or more than one process, can access simultaneously. The purpose of this facility is to ensure that a program can add, delete, or modify a record in a file without another program simultaneously accessing the same record.

When a program opens an indexed or relative file with the declared intention of writing or updating records, RMS locks any record accessed by the program. This locking prevents another program from accessing that record until the program releases it. The lock remains in effect until the program accesses another record. RMS then unlocks the first record and locks the second. The first record is then available for access by another concurrently executing program.

A program may also select a "manual" unlocking mode, in which all records accessed by the program remain locked until they are explicitly unlocked by calls to RMS.

Record I/O Processing

The organization of a file, defined when the file is created, determines the types of operations that the program can perform on records. Depending on file organization, Record Management Services permits a program to perform the following record operations:

- Get a record—RMS returns an existing record within the file to the program
- Put a record—RMS adds a new record that the program constructs to the file. The new record cannot replace an already existing record
- Find a record—RMS locates an existing record in the file. It does not return the record to the program, but establishes a new current position in the file
- Delete a record—RMS removes an existing record from the file. The delete record operation is not valid for sequential file organizations
- Update a record—The program modifies the contents of a record read from the file. RMS writes the modified record into the file, replacing the old record. The update record operation is not valid for sequential file organizations, except for sequentially organized disk files

Sequential File Record I/O

In a sequential file organization, a program can read existing records from the file using sequential or record's file address (RFA) access modes. New records can be added only to the end of the file and only through the use of sequential access mode, except that in the case where the sequential file has records of fixed length, records can be added using keyed access.

Relative File Record I/O

Relative file organization permits programs greater flexibility in performing record operations than does sequential organization. A program can read existing records from the file using sequential, random, or RFA record access mode.

New records can be sequentially or randomly written as long as the intended record cell does not already contain a record. Similarly, any record access mode can be used to perform a find operation. After a record has been found or read, RMS permits the delete operation. Once a record has been deleted, the record cell is available for a new record. A program can also update records in the file. If the format of the records is variable, update operations can modify record length up to the maximum size specified when the file was created.

Indexed File Record I/O

Indexed file organization provides the greatest flexibility in performing record operations. A program can read existing records from the file in sequential, record's file address (RFA), or random record access mode. When reading records in random record access mode, the program can choose one of four types of matches that RMS performs using the program-provided key value. The four types of matches are:

- Exact key match
- Approximate key match
- Generic key match
- Approximate and generic key match

Exact key match requires that the contents of the key in the record retrieved precisely match the key value specified in the program read operation.

The approximate match facility allows the program to select either of the following relationships between the key of the record retrieved and the key value specified by the program:

- Equal to or greater than
- Greater than

Generic key match means that the program need specify only an initial portion of the key value. Record Management Services (RMS) returns to the program the first occurrence of a record whose key contains a value beginning with those characters. This allows the program to retrieve a class of records, for example, all employee records in the personnel file with a name field beginning with M.

The final type of key match combines both generic and approximate facilities. The program specifies only an initial portion of the key value, as with generic match. Additionally, a program specifies that the key data field of the record retrieved must be either:

- equal to or greater than the program-supplied value
- greater than the program-supplied value

The find operation, similar to the read operation, can be performed in sequential, RFA, or random access mode. When finding records in random access mode, the program can specify any one of the four types of key matches provided for read operations.

In addition to read, write, and find operations, the program can delete any record in an indexed file and update any record.

Block I/O Processing

Block I/O allows a program to bypass the record processing capabilities of RMS entirely. Rather than performing record operations through the use of supported access modes, a program can process a file as a structure consisting solely of virtual blocks.

Using block I/O, a program reads or writes multiple virtual blocks by identifying a starting virtual block number in the file. Regardless of the organization of the file, RMS accesses the identified block or blocks on behalf of the program.

The presence of the block I/O facility permits user-created record formats on a Files-11 disk volume or ANSI magnetic tape volume.

RMS Runtime Environment

The environment within which a program processes RMS files at run-time consists of two levels, the file processing level and the record processing level.

At the file processing level, RMS and the operating system provide an environment that permits concurrently executing programs to share access to the same file. RMS ascertains the amount of sharing permissible from information provided by the programs themselves. Additionally, at the file processing level, RMS provides facilities that allow programs to minimize buffer space requirements for file processing.

At the record processing level, RMS allows programs to access records in a file through one or more record access streams (except for sequential files, which may only connect a single stream). Each record access stream represents an independent and simultaneously active series of record operations directed toward the file. Within each stream, programs can perform record operations synchronously or asynchronously. That is, RMS allows programs to choose between receiving control only after a record operation request has been satisfied (synchronous operation) or receiving control before the request has been satisfied (asynchronous operation).

For both synchronous and asynchronous record operations, RMS provides two record transfer modes, move mode and locate mode. Move mode causes RMS to copy a record from an I/O buffer into a program-provided location. Locate mode allows programs to read records directly in an I/O buffer.

RMS Utilities

The RMS procedures are complimented by a File Definition Language (FDL) and a number of utilities designed especially for RMS file creation and maintenance. They are called directly through DCL, and include:

- CONVERT
- CONVERT/RECLAIM
- EDIT/FDL
- CREATE/FDL
- ANALYZE/RMS_FILE

The File Definition Language is a special-purpose language used to write file specifications for data files. These specifications are then used by the RMS utilities and library routines to create data files and other data structures.

The CONVERT utility copies records from a source data file to a second data file with a different FDL specification and frequently with a different file organization. CONVERT can also be used to create a data file from an FDL specification.

EDIT/FDL is a utility used for creating or modifying an FDL specification file. Since FDL files are text files, they can created using a VAX/VMS editor, but programmers may find EDIT/FDL easier to use. It is designed for the task; for example, EDIT/FDL will provide default values for undesignated specifications.

CREATE/FDL is used to create an empty data file from an FDL specification. With this capability, a user can designed FDL files with EDIT/FDL, defining commonly needed data files, and then create such data files later, whenever they are required.

The ANALYZE/RMS_FILE utility is used to check the structure of a file for errors or potential for improvement. It can generate a statistical report on a files structure and use, or be employed interactively to "explore" the files structure. Conversely to CREATE/FDL, ANALYZE/RMS_FILE can generate an FDL file from a data file.

CONVERT/RECLAIM is used for reclaiming empty buckets in a prolog 2 indexed file, so that new records can be written into them. If all the records in a bucket have been deleted, that bucket is locked until the CONVERT/RECLAIM utility makes it available.

In toto, the RMS Utilities provide a total system for designing and tuning files in a users applications. (See figure 9 - 5.)

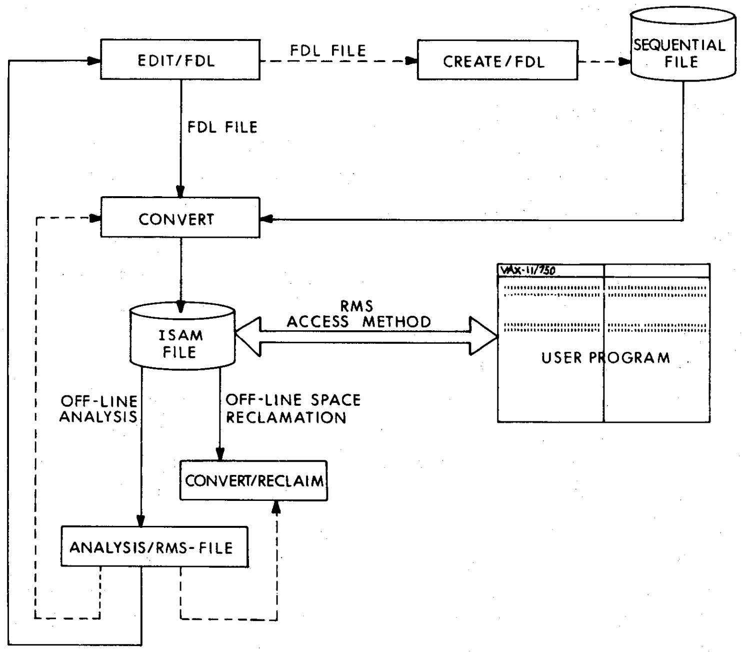


Figure 12-5 File Design and Tuning

USING VAX-11 RMS

VAX-11 RMS (Record Management Services) is a powerful tool for handling input/output tasks. Whether the user simply needs to have a program read input lines from a terminal, or needs full write sharing capability with record locking—allowing multiple processes to access and update records in the same files simultaneously—RMS can simplify and handle the task. Of course, more complex operations may require a number of parameters and (optionally) allow specification of many more; nevertheless, all of the basic RMS services use one of two control structures as input for their operation. The File Access Block (FAB) contains only fields relevant to file operations, such as the creation of a new file or opening an existing one. The Record Access Block (RAB) contains parameters necessary to perform record operations, such as record retrieval and update, on records within a file. The following table illustrates this division:

Table 12-3

Category	**Macro Name**	**Service**
FILE PROCESSING	$CREATE	Creates and opens a new file
FAB=address	$OPEN	Opens an existing file and initiates file processing
	$CLOSE	Terminates file processing and closes the file
RECORD PROCESSING RAB = address	$CONNECT	Associates and connects an RAB to the file
	$GET	Retrieves a record from a file
	$PUT	Writes a new record to a file
	$UPDATE	Rewrites an existing record in a file

The following brief program, with comments, demonstrates the ease and simplicity of using VAX-11 RMS (Record Management Services) to achieve an I/O operation. Several different runs of the program follow. It reads a sequential file containing ASCII text and uses a Run Time Library routine to print the text on the user's terminal. Then the file is copied to a remote network node and the program accesses it on that node. Relative and indexed files could be handled as easily as this sequential file, and with no rewriting of the program.

```
Buffer: .blkb   100                         ; allocate a 100 byte buffer
Buff_desc:                                  ; descriptor for buffer
        .long   0                           ; length will be set at run time
        .long   Buffer                      ; address of buffer

My_fab: $FAB    FNM=<INFILE>                ; File Access Block

My_rab: $RAB    FAB=My_fab,-                ; Record Access Block
                UBF=Buffer,-
                USZ=100

Start:  .word   0

        $OPEN   FAB=My_fab                  ; open the file
        BLBC    R0,Exit                     ; exit on error

        $CONNECT RAB=My_rab                 ; connect for record operations
        BLBC    R0,Exit                     ; exit on error

Get_record:
        $GET    RAB=My_rab                  ; get the first record
        BLBC    R0,Exit                     ; exit on error

        MOVW    My_rab + rab$w_rsz,Buff_desc  ; store length of record in desc
        PUSHAB  Buff_desc                   ; push descriptor address for output
        CALLS   #1,LIB$PUT_OUTPUT           ; print the record
        BRB     Get_record                  ; go back and get the next record

        $CLOSE  FAB=My_fab                  ; and close the file

Exit:   RET

        .end    Start
```

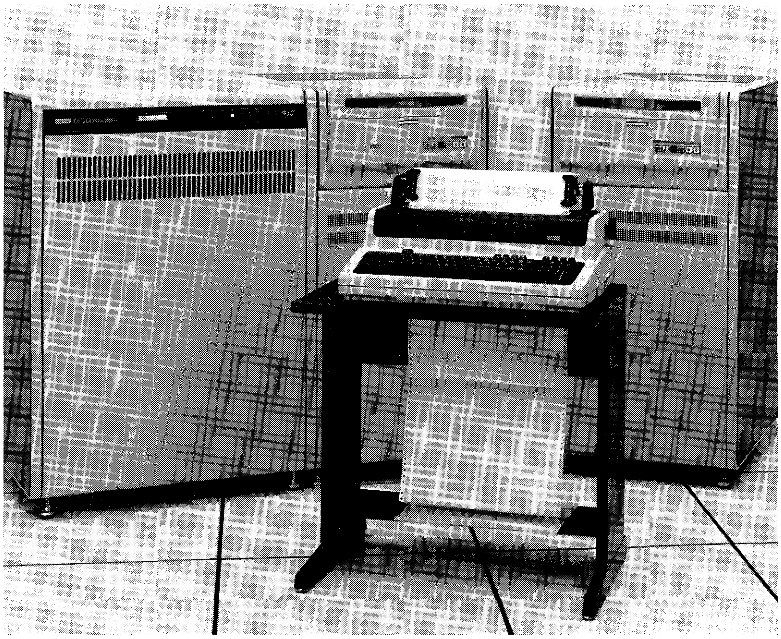
Figure 12-6   A Sample RMS User Program

Notice in the running of the sample program that merely by use of the ASSIGN command the programmer was able to apply INFILE to several different files without reworking the program. The program could have accessed any of a variety of sequential, relative, or indexed formats without modification. Also, it is not necessary to close a file explicitly because all files will be closed ("run down") by RMS when the image exits. In fact, VAX-11 RMS handles all internal buffer and control structure allocation and management for the user.

The variety of file organizations, record formats, and access modes, plus network support, that RMS provides makes it one of the most useful features of the VAX/VMS operating system.

```
$ @LOGIN
$ set noverify
$ ! demonstrate how easy RMS is to use
$ set noonerror
$ !
$ ! use TYPE to type out the sample text file used in
$ ! the following tests.
$ !
$ TYPE DEMO.TXT
this is a test program
containing ascii text records
this will be printed out by
the test program SIMPLE
$ !
$ ! now assign the text file DEMO.TXT
$ ! to the logical name INFILE which the
$ ! test program SIMPLE will open for input
$ !
$ ASSIGN DEMO.TXT INFILE
$ run simple
this is a test program
containing ascii text records
this will be printed out by
the test program SIMPLE
%RMS-E-EOF, end of file
$ !
$ ! copy the text file to another vax node on the network
$ !
$ COPY/LOG DEMO.TXT GALAXY::DEMO.TXT
%COPY-S-COPIED, DBA0:[SAETHER]DEMO.TXT;3 copied to GALAXY::DEMO.TXT; (4 records)
%COPY-S-NEWFILES, 1 file created
$ !
$ ! now use the same test program SIMPLE to access
$ ! the file across the network
$ !
$ ASSIGN GALAXY::DEMO.TXT INFILE
  PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
$ RUN SIMPLE
this is a test program
containing ascii text records
this will be printed out by
the test program SIMPLE
%RMS-E-EOF, end of file
```

Figure 12-7 Running the Sample Program



CHAPTER OVERVIEW

Physical devices need software control if they are to run properly (or at all). This chapter explains the VAX/VMS device driver elements, defines a fork process, and gives a complete sample lineprinter I/O driver source program listing.

Topics include:

- Elements of a Device Driver
- General Device Activity
- A Lineprinter I/O Request and Program Listing

CHAPTER 13
I/O DRIVERS

INTRODUCTION

A VAX/VMS device driver is a set of tables and routines that control I/O operations on a peripheral device attached to a VAX system. A driver performs the following functions:

- Defines the peripheral device for the rest of the VAX/VMS operating system
- Defines the driver for the system procedure that maps and loads the driver and its device database into system virtual memory
- Initializes the device (and/or its controller) at system startup time and after a power failure
- Translates software requests for I/O operations into device-specific commands
- Activates the device
- Responds to hardware interrupts generated by the device
- Reports device errors
- Returns data and status from the device to user software

The VAX/VMS operating system performs all I/O processing that is independent of the particular device. Such processing is known as device-independent processing. When details of an I/O operation need to be translated into terms recognizable by a specific device, the operating system transfers control to a device driver. Such processing is called device-dependent processing.

Since different types of peripheral devices expect different commands and setups, each type of device on a VAX system needs its own supporting driver. The driver, consisting of code and static tables, performs all device-dependent processing.

The operating system and device drivers cooperate in processing an I/O operation by sharing a common I/O database. The database describes, in terms familiar to the VAX/VMS operating system, the specifications and functions of each device.

DEVICE DRIVER ELEMENTS

A device driver contains a set of routines that the operating system calls to perform device-dependent processing on an I/O request. The routines of a VAX/VMS driver perform the following functions:

Initialization	At the time that the driver is loaded, or after a power failure, initializes a device or controller by setting hardware registers and initializing fields in the I/O database
I/O setup	Prepares an I/O request for a device by formatting data, allocating system buffers, locking pages in memory, validating the request, etc.
Start I/O	Sets up device registers and the I/O database to start a device; completes I/O request
Interrupt handling	Responds to hardware interrupts; read and resets device registers; returns status
Error recovery	Sets up device registers for retry of an I/O operation; applies Error Correcting Code (ECC) corrections to disk data; returns error status
Error logging	Writes the contents of device registers and other data into an error buffer
Cancel I/O	Sets up device registers to terminate I/O activity

Drivers need not contain all of the routine types listed above, but every driver must at least include subroutines to handle I/O startup and interrupts. Figure 12-1 illustrates operating system interaction with I/O driver subroutines.

The other parts of a device driver are static tables that describe the device and the driver. Each driver must contain the following three tables.

Driver prologue table	Describes the driver and the device type to the system driver loading procedure
Driver dispatch table	Lists the entry point addresses of standard driver routines. Also records the size of diagnostic and error logging buffers for the device type

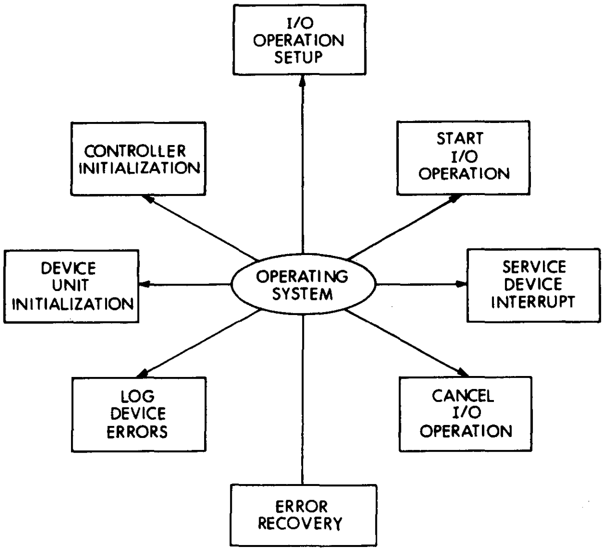


Figure 13-1 Operating System Calls to Driver Subroutines

Function decision table	Lists all valid function codes and buffered function codes for the device. Also lists the addresses of function setup routines. Function Decision Table (FDT) routines may be internal to the operating system, to the device driver itself, or to both. (Buffered I/O is I/O that is buffered through the system data buffer i.e., a READ or WRITE to a user terminal. Direct I/O is I/O executed directly out of the user data buffer, i.e., a READ or WRITE to a disk)

Drivers do not decide when to act or what function to perform. Instead, the operating system interprets all demands for service from users and devices. By consulting the I/O database and the tables in the drivers, the operating system determines which device-dependent processing is available in the driver, and the entry point of the routine that can provide the service. Figure 12-2 illustrates I/O driver organization conceptually.

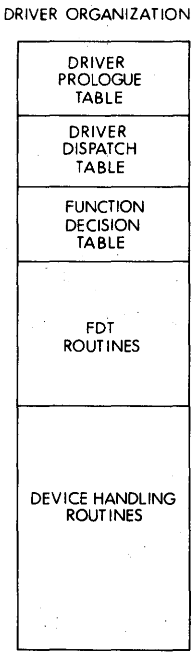


Figure 13-2 Driver Organization

The I/O Database

The operating system and the device drivers refer to an I/O database that consists of three main parts:

- Driver tables that allow the operating system to load drivers, validate device functions, and call drivers at their entry points
- System data structures that describe every bus adapter, every device unit, and every logical pathway to a device or group of devices
- Dynamically allocated packets that define individual requests for I/O activity; these packets are known as I/O request packets (IRPs)

The three driver tables are listed in the previous section. The control blocks that describe and permit access to peripheral hardware are created either at system initialization or at the time that a driver is loaded. Drivers reference some or all of the control blocks described below.

Device Data Block	Records the generic device name and driver name for a set of devices attached to a single controller
Unit Control Block	Defines the characteristics and current state of an individual device unit
Channel Request Block	Defines the current activity of a single controller
Interrupt Dispatch Block	Records the characteristics of a single controller and points to the devices it controls
Adapter Control Block	Defines the characteristics and current state of a UNIBUS or MASSBUS adapter

The third part of the I/O database is a group of I/O request packets. When a user program requests device activity, the operating system constructs a packet of data—called an I/O request packet—that describes the I/O request in a standard form.

The I/O request packet (IRP) is sent to device driver routines as a source of detailed instructions. The packet includes buffer addresses, a pointer to the target device, I/O function codes, and further pointers to the I/O database. In addition, the packet contains fields into which the driver subroutine can write, such as status fields.

The I/O database and I/O request packets are the communicating links between operating system and driver handling of I/O processing.

FORK PROCESSES

Device driver routines that complete an I/O operation after a device interrupt execute for relatively short periods of time. The routines may be suspended to wait for shared resources. To facilitate fast dispatching of driver routines, the operating system does not schedule the routines as full processes.

Instead, the VAX/VMS operating system implements I/O drivers as fork processes. Fork processes are created dynamically and contain minimal context. They execute at software interrupt level and entirely within the system address space. They cannot incur exceptions, and thus all code they execute must be resident. Of the 15 software interrupt levels provided by VAX hardware, levels 8 through 11 are used to schedule fork processes. Figure 12-3 illustrates the interrupt priority level scheme.

Fork processes are scheduled by constructing a specialized control block called a fork block, inserting the fork block in a fork queue, and then requesting a software interrupt. Each software interrupt level

PRIORITY		HARDWARE EVENT	
Hex	Decimal		
1F	31	Machine Check, Kernel Stack Not Valid	
1E	30	Power Fail	
1D	29		Processor,
1C	28		
1B	27		Memory, or
1A	26		
19	25		Bus Error
18	24	Clock	
17	23	UNIBUS BR7	
16	22	UNIBUS BR6	
15	21	UNIBUS BR6	
14	20	UNIBUS BR4	Device Interrupt
13	19		
12	18		
11	17		
10	16		
PRIORITY		SOFTWARE EVENT	
0F	15		
0E	14		Reserved for
0D	13		DIGITAL
0C	12		
0B	11		
0A	10		Device
09	09		Drivers
08	08		
07	07	Timer Process	
06	06	Queue Asynchronous System Trap (AST)	
05	05	Reserved for DIGITAL	
04	04	I/O Post	
03	03	Process Scheduler	
02	02	AST Delivery	
01	01	Reserved for DIGITAL	
00	00	User Process Level	

Figure 13-3 Interrupt Priority Level

contains a fork queue which is a first-in/first-out list of fork blocks for fork processes that are waiting to be dispatched.

The fork block contains the initial context for a fork process. Fork processes are dispatched by the fork dispatcher, which is executed in response to a software interrupt on levels 6 and 8 through 11. A fork block is removed from the front of the appropriate queue. The fork process is then dispatched using a Jump to Subroutine (JSB) instruction to its entry point. The fork process can freely use registers R0 through R5, but must explicitly save and restore any other registers.

A fork process terminates its execution via an RSB instruction, which causes a return to the fork dispatcher. The fork dispatcher repeats the dispatching sequence for another fork block until the fork queue for the corresponding level is empty. It then restores registers R0 through R5 and executes a Return from Exception or Interrupt (REI) instruction which causes execution to resume at a lower interrupt priority level (IPL).

GENERAL DEVICE ACTIVITY

The VAX/VMS operating system and the device driver cooperate to execute a user request for an I/O operation.

All input and output operations under the VAX/VMS operating system result from software requests for I/O processing. Presented here is a general discussion of a user process terminal READ request.

Before I/O requests can be made to a device, however, the user must assign a channel to establish a link between the user process and the device. The process uses this channel to transfer information to and from the device. The Assign I/O Channel ($ASSIGN) system service is used to assign a channel to a device. The $ASSIGN system service returns the channel number. The process can then request an I/O operation by calling the Queue I/O ($QIO) system service and specifying, as one argument, the channel number returned by the $ASSIGN system service.

The VAX/VMS mechanism for requesting an I/O operation is the Queue I/O ($QIO) system service call. Any native mode program can issue a $QIO system service directly. For example, an assembly language program can issue a $QIO directly with the following instructions:

```
$QIO_S          CHAN = R2,-
                FUNC = #IO$_READVBLK
                P1 = buffer_address,-
                P2 = #buffer_size
```

An example of the $ASSIGN system service is as follows:

```
CHANNEL:          .WORD 0
DEVICE:           .LONG 20$-10$
                  .LONG 10$
10$:              .ASCII \TTA0:\
20$:
                  $ASSIGN_S DEVNAM=DEVICE,-
                  CHAN=CHANNEL
```

An assembly language program can also issue a $QIO indirectly to a record-oriented device with a Record Management Services (RMS) function call. In the following example, the $GET is eventually translated by RMS into a $QIO system service call.

```
$GET                RAB = file_rab
```

The RMS function call assumes that all the details of the I/O request have been set up previously in the RAB and FAB data structures. An executive privileged piece of software in system space called RMS translates the function call into a $QIO system service call.

The RMS $GET command is equivalent to a READ function. The $GET command translates into the basic $QIO system service request for I/O processing. The $GET command sets up a $QIO with the information necessary to perform the I/O request. Typically, $GET will provide $QIO with the following attributes:

- Channel number
- Buffer address
- Buffer length

Record Management Services will issue the $QIO resulting from the $GET command.

However, as in the previous case, where $QIO was requested directly, the $ASSIGN request for a channel number must also precede the RMS $GET command. The $OPEN command must precede the $GET command. The $OPEN command translates into a number of functions, including the $ASSIGN system service. The $ASSIGN system service once again provides the process with a channel to the specified device.

FORTRAN programmers request I/O operations with the FORTRAN language statements READ and WRITE. An example follows:

```
READ(2,format,ERR=exit,END=exit2)
```

The FORTRAN compiler translates the READ statement into a call to the FORTRAN runtime system. The runtime system further translates the statement into a $GET which translates into a $QIO system service call. This process occurs according to the following steps:

1. Program executes "READ" statement
2. READ statement causes a call to the runtime system
3. Runtime system checks to see if this is the first I/O request on that file
4. If it is, the runtime system performs an "OPEN." (When the first I/O request is queued, a function of the runtime system is to perform an $OPEN on the file, establishing a device channel)
5. Performs $GET command
6. $GET translates into $QIO

Optionally, the FORTRAN programmer may directly specify the $QIO system service. This procedure is as follows:

```
DIMENSION      IOSB(2),BUF(20)
STATUS = SYS$ASSIGN('TTA5:',CHAN,,)
IF(.NOT. STATUS)GO TO 980
      .
      .
      .
STATUS=SYS$QIO(,%VAL(CHAN),%VAL('31'X),IOSB,,,BUF,%
VAL(80),,,,)
IF (.NOT. STATUS) GO TO 900
IF (.NOT. IOSB (1)) GO TO 950
      .
      .
      .
900 TYPE 910, STATUS
910 FORMAT (' QIO NOT ACCEPTED, STATUS =', Z8)
      .
      .
      .
950 TYPE 960, IOSB (1)
960 FORMAT (' I/O FAILURE, IOSB =', Z8)
      .
      .
      .
980 TYPE 990, STATUS
990 FORMAT (' I/O ASSIGN FAILED, STATUS = ', Z8)
      .
      .
      .
END
```

In this case, the programmer must perform an $ASSIGN to establish a device channel.

Therefore, by requesting an I/O operation directly, via Record Management Services, or from a high-level language, the user process specifies a logical path to the device, a READ function code, and the address of a user buffer to hold the data.

The $QIO service routine in the operating system allocates an I/O request packet, validates the request, and locates database descriptions of the device and its driver (i.e., channel control block, unit control block, device data block, etc.). The operating system locates device-independent information in this database and stores it in the I/O request packet (IRP).

Upon completion of device-independent I/O preprocessing, the $QIO service routine calls a READ function routine in the driver to allocate a system buffer into which the device can write data. The READ function routine is pointed to by the function decision table. The system buffer is that buffer used to contain typed information from the terminal to be transferred to user process space via the kernel mode AST. The READ function routine places device-dependent information into the IRP. Figure 12-4 illustrates the function decision table.

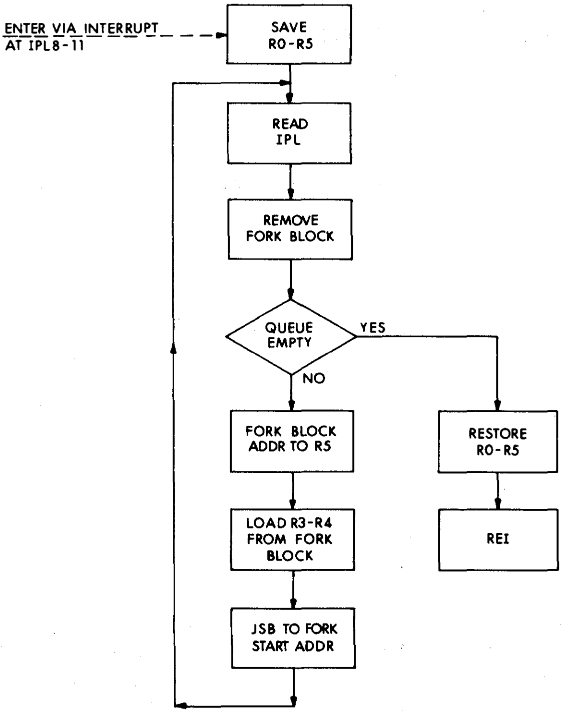


Figure 13-4 Function Decision Table

The I/O request packet (IRP) will be the only data explicitly passed to the I/O driver. The READ function routine returns the completed IRP to the operating system for queuing to the driver. Up to this point, all of the execution has been contained within the context of the user process. However, the queuing of the I/O request to the driver by the operating system is executed in system space, running in kernel mode.

If the device is free, the operating system calls the driver Start I/O routine immediately. The Start I/O routine, using the IRP as its database, activates the device. If the device is busy, however, the operating system inserts the IRP into a device wait queue. The IRP remains in the queue until the device is free and the IRP is first in the queue. Then the operating system dequeues the IRP and calls the driver Start I/O routine.

After a driver starts a device, the driver transfers control to an operating system routine that suspends the driver until a device interrupt or timeout occurs. The operating system suspends the driver so that a computable process can utilize the CPU while the driver waits for a device interrupt.

At this point of the READ operation, the user could enter information from the keyboard. Typing a character causes an interrupt to occur at device hardware interrupt priority level (IPL). When the interrupt occurs, control is passed to the device driver interrupt service routine (ISR). In the case of a READ operation, the ISR removes data from the device data register and places it in the system buffer. In the case of the WRITE operation, the ISR removes data from the system buffer and places it in the device data register. When the data transfer is complete, the driver's service routine restores the saved state of the driver process.

The driver process obtains status information about the transfer by reading device registers. The driver returns the status of the I/O to the operating system for later delivery to the issuing process. The operating system copies the newly read data into the user buffer via the kernel mode asynchronous system trap (AST). It then returns to the user process with the final status of the I/O operation via the kernel AST.

The next section covers in greater detail an actual write I/O operation to the lineprinter. Accompanying the text is a commented copy of the DIGITAL-supported VAX/VMS lineprinter device driver. Correspondence between text and I/O driver code is noted.

A SAMPLE LINEPRINTER QIO REQUEST

The following section describes a typical write function to the lineprinter. At the conclusion of the text, a commented source program listing of the lineprinter I/O device driver is included. The device driver listing is separated into sections which correspond to the following text.

The LP11-R is a highspeed buffered lineprinter. A process can perform three functions with respect to the LP11. They are:

1. Write data to the lineprinter

2. Read lineprinter device characteristics:
   — Carriage width in characters
   — Check for mechanical form-feed capability
   — Check for lowercase print capability
3. Write lineprinter device characteristics:
   — User can set carriage width
   — User can set lowercase print capability
   — User can set mechanical form-feed capability

Point 2 corresponds to the operating system's sense mode routine, EXE$SENSEMODE, which is called by the function decision table (FDT) dispatcher when the routine is entered in the Function Decision Table.

This section will illustrate a user process request to print a line of data to the lineprinter. The driver routines used to accomplish a write data function are:

- FDT subroutines that reformat the user-supplied data
- A driver Start I/O routine that writes data to the device data buffer until the printer enters a busy state, at which time the driver will wait for the printer to interrupt, indicating the device data buffer has been printed
- A driver interrupt handler that returns control to the Start I/O routine after a hardware interrupt from the lineprinter
- Initialization routines called at system startup and after power failure to initialize the unit and/or the controller

A process can print a line to this device by issuing a $QIO call that specifies the WRITE VIRTUAL BLOCK function. This procedure is illustrated below:

```
$QIO_S          CHAN = channel_number,-
                FUNC = #IO$_WRITEVBLK,-
                P1 = (user) buffer_address,-
                P2 = (user) #buffer_size,-
                P4 = #↑X30 (carriage control—performs a double
                space before line is printed and a carriage return
                after)
```

Figure 12-5 illustrates the flow of execution through the operating system and the lineprinter driver to satisfy this I/O request.

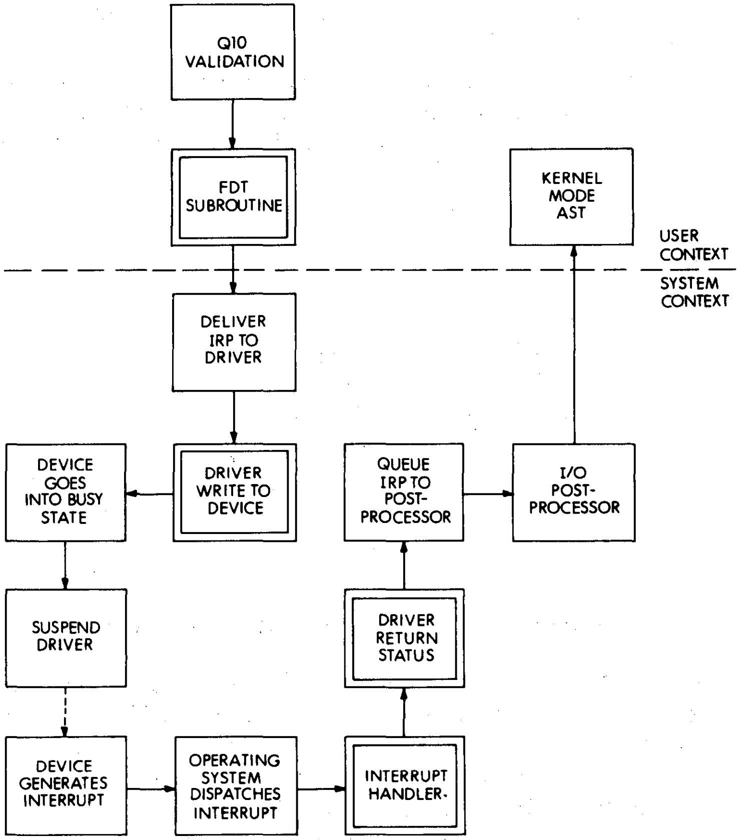


Figure 13-5 Lineprinter Write Function

The double-sided boxes in the figure indicate processing performed by driver subroutines. All processing above the dotted line occurs in the context of the user process. Processing below the dotted line occurs in system or interrupt context.

I/O Preprocessing by the Operating System (Device-Independent)
The first step in processing an I/O request is to validate that the request is correctly specified. This function is performed by the $QIO service routine and consists of the following tasks:

1. The $QIO service routine validates that the location channel_number contains an index into the process I/O channel list,

and that the process has previously assigned the lineprinter device to the specified process channel.

Also, during this sequence, the $QIO service routine obtains the address of the lineprinter driver's function decision table (FDT). Figure 12-6 illustrates the sequence of pointers from the channel index number to the FDT address.

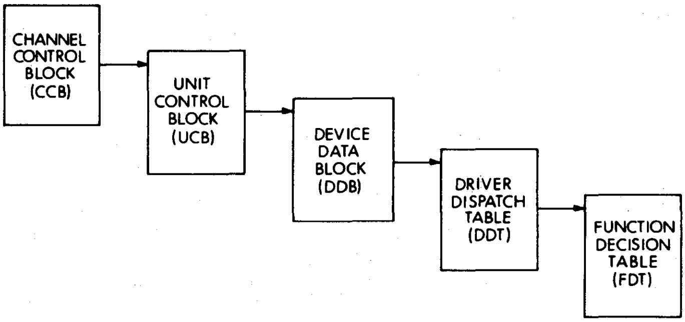

CHANNEL CONTROL BLOCK (CCB)
UNIT CONTROL BLOCK (UCB)
DEVICE DATA BLOCK (DDB)
DRIVER DISPATCH TABLE (DDT)
FUNCTION DECISION TABLE (FDT)

Figure 13-6 Locating Function Decision Table

2. The $QIO service routine validates (via the FDT) that the lineprinter function decision table lists IO$_WRITEVBLK as a valid function for the device
3. The $QIO service routine validates that the process quotas permit this buffered I/O request
4. An FDT routine validates that the user has read access to the buffer starting at buffer_address

If all the operating system preprocessing checks succeed, the $QIO service routine creates an I/O request packet (IRP) in non-paged system address space. The $QIO routine writes all device-independent details about the I/O request into the IRP.

I/O Preprocessing by the Driver (Device-Dependent)
This section corresponds to the write function decision table (FDT) routine coding of the lineprinter driver.

The $QIO service routine scans the function decision table. It then checks the FDT for an entry that associates the IO$_WRITEVBLK function code with an FDT subroutine. The $QIO service routine then calls the subroutine—a device-specific subroutine located in the lineprinter device driver.

The FDT subroutine copies data from the user process space buffer into a system space buffer. The subroutine performs this function by performing the following tasks:

- The subroutine calculates the length of the required system space buffer
- If the process byte count quota is not exceeded, the subroutine allocates a buffer from system address space and stores the address of the buffer in the I/O request packet (IRP). It charges against the process byte count quota
- The FDT subroutine accesses the lineprinter's unit control block (UCB)
- The subroutine reads the description of the lineprinter's current line and page position from the driver's UCB
- The subroutine then reformats the data contained in the process buffer and places it into the system buffer. It adds carriage control characters (specified in the $QIO argument "P4") before and after the data
- The subroutine rewrites the updated line and page positions into the device's unit control block
- The subroutine then transfers control to an operating system routine that queues the I/O packet to the device driver

Both operating system and driver I/O preprocessing occur in the context of the user process. The user address space is mapped, and the interrupt priority level is low enough to permit context switching or process scheduling of the process and paging. Subsequent queuing of the I/O request to the driver and all resulting driver processing occur at higher interrupt priority levels (IPLs) to synchronize driver handling of the device.

Queuing the I/O Packet to the Driver

To queue the I/O request packet to the proper driver, the operating system queuing routine first raises the IPL to the device fork level stored in the unit control block (UCB). Raising priority to fork level synchronizes driver access to the I/O database.

If the device is not busy—indicated by the "Busy" bit's being clear in the status word of the UCB—then the operating system can transfer control to the driver at the start I/O entry point. To find the proper entry point, the initiation routine chains through the I/O database to the driver dispatch table (DDT), which contains the start I/O entry point. This process is illustrated in Figure 12-7.

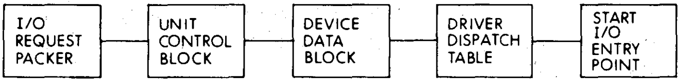


Figure 13-7 Entry Point Location

If the device is busy with another I/O transfer, the operating system inserts the I/O request packet (IRP) in a device wait queue according to the software schedulable priority of the process.

Start I/O

This section also corresponds to the Start I/O routine of the lineprinter device driver. The Start I/O routine contains the code to modify device registers, fork to device fork level, and complete the I/O request.

The lineprinter driver routine writes to the lineprinter data buffer according to the following algorithm.

1. Locates the LP11 device registers via a chain of pointers starting at the device's unit control block (UCB). This process is illustrated in Figure 12-8.

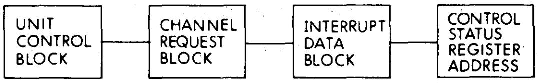


Figure 13-8 LP11 Device Register Location

The control/status register (CSR) indicates the status of the lineprinter controller. The next successive address is the data buffer. In contrast to many other devices, such as disks, the LP11 lineprinter does not share a controller with other devices. Therefore, no arbitration ownership of the controller is required. The CSR address is always the address of the lineprinter control/status register, and all other device registers are at fixed offsets from this address.

2. The lineprinter driver routine writes data in the device's data buffer, then raises the IPL to block out all interrupts and sets the interrupt enable bit in the device's control/status register. It then calls an operating system routine to suspend driver processing until the lineprinter generates an interrupt.

The operating system routine suspends the driver by:

- Saving driver context—R3, R4, and the address of the next instruction in the driver—in the device's unit control block (UCB)

- Calculating the time at which the device will timeout
- Setting bits in the device's UCB to indicate that the driver expects a device interrupt within a specified time period

The operating system then drops the IPL back to driver fork level and returns control to the caller of the driver Start I/O routine.

The driver remains in a suspended state until one of two events occurs:

- The lineprinter generates a hardware interrupt
- The operating system reports a device timeout because the lineprinter did not generate a hardware interrupt within a specified period of time

Normally, the LP11 prints the contents of its data buffer and generates the interrupt. If the printer is turned off during an operation or if it runs out of paper, the operating system reports a device timeout.

Interrupt Handling

This section corresponds to the interrupt service routine code of the lineprinter device driver.

When the LP11 lineprinter generates a hardware interrupt, an operating system interrupt handling routine gains control, determines which device is requesting an interrupt, and passes the interrupt to the LP11 driver interrupt handling routine.

The driver's interrupt handling routine restores control to the driver as follows:

1. Confirms that the interrupt was expected by examining bits in the device's unit control block (UCB)
2. Restores the saved registers, R3 and R4, from the device's UCB
3. Transfers control to the driver PC address which was stored in the device's UCB

Rather than execute in the interrupt context, the reactivated driver routine calls the operating system to create a driver fork process. The operating system again suspends driver processing by:

- Saving driver context (i.e., R3, R4, and the driver PC address in the UCB)
- Inserting the UCB in the appropiate fork queue

The driver suspension allows the operating system to reschedule driver processing at a lower IPL. The fork dispatcher can reactive the driver when IPL drops to driver fork level.

After creating the fork process, the operating system returns control to the driver's interrupt handling routine. The handling routine:

- Restores registers saved at the time of the device interrupt
- Dismisses the interrupt

I/O Completion Processing
When the driver Start I/O routine finishes the write transfer, the routine stores in R0:

- A success status code
- The number of bytes transferred

Then the routine transfers control to the operating system to complete the I/O request.

The operating system inserts the I/O request packet (IRP) into an I/O postprocessing queue. If another IRP is in the device wait queue, the operating system dequeues that IRP, and calls the driver Start I/O routine to process the IRP.

When the interrupt priority level (IPL) drops to IPL$_IOPOST, an I/O postprocessing dispatcher dequeues the IRP for the lineprinter I/O request and performs the following steps:

1. Adds one (1) to the process's buffered I/O quota
2. Deallocates the system buffer used for the reformatted user data
3. Sets an event flag to indicate that the I/O operation is complete
4. Queues a kernel mode asynchronous system trap (AST) routine that deallocates the IRP and optionally loads I/O status into a user-specified I/O Status Block

The user process determines that the I/O operation is complete by examining the event flag.

LINE PRINTER I/O DRIVER SOURCE PROGRAM LISTING

```
        .TITLE  LPDRIVER - LP11/LS11/LV11 LINE PRINTER DRIVER
        .IDENT  /X05/

;
; COPYRIGHT (C) 1977
; DIGITAL EQUIPMENT CORPORATION, MAYNARD, MASSACHUSETTS 01754
;
; THIS SOFTWARE IS FURNISHED  UNDER A LICENSE FOR USE ONLY ON A SINGLE
; COMPUTER  SYSTEM AND  MAY BE  COPIED ONLY WITH  THE INCLUSION OF THE
; ABOVE COPYRIGHT NOTICE.  THIS SOFTWARE, OR ANY OTHER COPIES THEREOF,
; MAY NOT BE PROVIDED OR  OTHERWISE MADE AVAILABLE TO ANY OTHER PERSON
; EXCEPT FOR USE ON SUCH SYSTEM AND TO ONE WHO AGREES TO THESE LICENSE
; TERMS.  TITLE TO AND  OWNERSHIP OF THE  SOFTWARE  SHALL AT ALL TIMES
; REMAIN IN DEC.
;
; THE INFORMATION IN THIS SOFTWARE IS SUBJECT TO CHANGE WITHOUT NOTICE
; AND SHOULD  NOT BE CONSTRUED  AS A COMMITMENT  BY DIGITAL  EQUIPMENT
; CORPORATION.
;
; DEC ASSUMES  NO  RESPONSIBILITY  FOR  THE USE OR  RELIABILITY OF ITS
; SOFTWARE ON EQUIPMENT WHICH IS NOT SUPPLIED BY DEC.
```

```
;
;
;
;
;
;
;
; LP11/LS11/LV11 LINE PRINTER DRIVER
;
; MACRO LIBRARY CALLS
;

        $CRBDEF                                 ;DEFINE CRB OFFSETS
        $DDBDEF                                 ;DEFINE DDB OFFSETS
        $DPTDEF                                 ;DEFINE DPT OFFSETS
        $IDBDEF                                 ;DEFINE IDB OFFSETS
        $IODEF                                  ;DEFINE I/O FUNCTION CODES
        $IRPDEF                                 ;DEFINE IRP OFFSETS
        $LPDEF                                  ;DEFINE LINE PRINTER CHARACTERISTICS
        $MSGDEF                                 ;DEFINE SYSTEM MESSAGE TYPES
        $PCBDEF                                 ;DEFINE PCB OFFSETS
        $UCBDEF                                 ;DEFINE UCB OFFSETS
        $VECDEF                                 ;DEFINE VEC OFFSETS

;
; LOCAL SYMBOLS
;
; ARGUMENT LIST OFFSET DEFINITIONS
;

P1=0                                            ;FIRST FUNCTION DEPENDENT PARAMETER
P2=4                                            ;SECOND FUNCTION DEPENDENT PARAMETER
P3=8                                            ;THIRD FUNCTION DEPENDENT PARAMETER
P4=12                                           ;FOURTH FUNCTION DEPENDEND PARAMETER
P5=16                                           ;FIFTH FUNCTION DEPENDENT PARAMETER
P6=20                                           ;SIXTH FUNCTION DEPENDENT PARAMETER

;
; CHARACTER CODE DEFINITIONS
;

C_CR=13                                         ;CARRIAGE RETURN
C_FF=12                                         ;FORM FEED
C_VT=11                                         ;VERTICLE TAB
C_LF=10                                         ;LINE FEED
C_TAB=9                                         ;TABULATION

;
; FLAG REGISTER BIT DEFINITIONS
;

M_CRPEND=1                                      ;CARRIAGE RETURN PENDING
V_CRPEND=0                                      ;

;
; LP11/LS11/LV11 DEVICE REGISTER OFFSET DEFINITIONS
;

        $DEFINI LP

$DEF    LP_CSR          .BLKW   1               ;CONTROL STATUS REGISTER
$DEF    LP_DBR          .BLKW   1               ;DATA BUFFER REGISTER

        $DEFEND LP

;
; DEFINE DEVICE DEPENDENT UNIT CONTROL BLOCK OFFSETS
;

        $DEFINI UCB

.=UCB$K_LENGTH                                  ;

$DEF    UCB$L_LP_MUTEX  .BLKL   1               ;LINE PRINTER UCB MUTEX
$DEF    UCB$B_LP_CURSOR .BLKB   1               ;CURRENT HORIZONAL POSITION
$DEF    UCB$B_LP_LINCNT .BLKB   1               ;CURRENT LINE COUNT ON PAGE
$DEF    UCB$B_LP_OFLCNT .BLKB   1               ;OFFLINE TIME COUNTER
                        .BLKB   1               ;SPARE UNUSED BYTE

        $DEFEND UCB
```

```
;
; LOCAL DATA
;
; DRIVER PROLOGUE TABLE
;

        DPTAB   -                               ;DEFINE DRIVER PROLOGUE TABLE
                END=LP_END,-                    ;END OF DRIVER
                ADAPTER=UBA,-                   ;ADAPTER TYPE
                UCBSIZE=124,-                   ;UCB SIZE
                NAME=LPDRIVER                   ;DRIVER NAME
        DPT_STORE INIT                          ;CONTROL BLOCK INIT VALUES
        DPT_STORE UCB,UCB$B_FIPL,B,8            ;FORK IPL
        DPT_STORE UCB,UCB$L_DEVCHAR,L,-         ;DEVICE CHARACTERISTICS
                <DEV$M_REC-                     ; RECORD ORIENTED
                !DEV$M_AVL-                     ; AVAILABLE
                !DEV$M_CCL-                     ; CARRIAGE CONTROL DEVICE
                !DEV$M_ODV>                     ; OUTPUT DEVICE
        DPT_STORE UCB,UCB$B_DEVCLASS,B,DC$_LP ;DEVICE CLASS
        DPT_STORE UCB,UCB$B_DEVTYPE,B,LP$_LP11 ;DEVICE TYPE




DPT_STORE UCB,UCB$W_DEVBUFSIZ,W,132 ;DEFAULT BUFFER SIZE
DPT_STORE UCB,UCB$L_DEVDEPEND,L,<64@24+LP$M_MECHFORM> ;PRINTER PARAMETERS
DPT_STORE UCB,UCB$B_DIPL,B,20   ;DEVICE IPL
DPT_STORE UCB,UCB$L_LP_MUTEX,W,-1 ;INITIALIZE MUTEX
DPT_STORE REINIT                ;CONTROL BLOCK RE-INIT VALUES
DPT_STORE CRB,CRB$L_INTD+4,D,LP$INT ;INTERRUPT SERVICE ROUTINE ADDRESS
DPT_STORE CRB,CRB$L_INTD+VEC$L_INITIAL,D,LP_LX11_CINIT ;CONTROLLER INIT
DPT_STORE CRB,CRB$L_INTD+VEC$L_UNITINIT,D,LP_LX11_INIT ;UNIT INIT
DPT_STORE DDB,DDB$L_DDT,D,LP$DDT ;DDT ADDRESS
DPT_STORE END                   ;

;
; DRIVER DISPATCH TABLE
;

        DDTAB   LP,-                            ;DRIVER DISPATCH TABLE
                STARTIO,-                       ;START I/O OPERATION
                0,-                             ;UNSOLICITED INTERRUPT
                FUNCTABLE,-                     ;FUNCTION TABLE
                +IOC$CANCELIO,-                 ;CANCEL I/O
                0,-                             ;REGISTER DUMP ROUTINE
                0,-                             ;SIZE OF DIAGNOSTIC BUFFER
                0                               ;SIZE OF ERROR LOG BUFFER
        .PAGE
        .SBTTL  LP11/LS11/LV11 FUNCTION DECISION TABLE
;
; LP11/LS11/LV11 FUNCTION DECISION TABLE
;

FUNCTABLE:                                      ;FUNCTION DECISION TABLE
        FUNCTAB ,-                              ;LEGAL FUNCTIONS
                <SENSECHAR,-                    ;SENSE CHARACTERISTICS
                 SETCHAR,-                      ;SET CHARACTERISTICS
                 SENSEMODE,-                    ;SENSE MODE
                 SETMODE,-                      ;SET MODE
                 WRITELBLK,-                    ;WRITE LOGICAL BLOCK
                 WRITEPBLK,-                    ;WRITE PHYSICAL BLOCK
                 WRITEVBLK>                     ;WRITE VIRTUAL BLOCK
        FUNCTAB ,-                              ;LEGAL FUNCTIONS
                <SENSECHAR,-                    ;SENSE CHARACTERISTICS
                 SETCHAR,-                      ;SET CHARACTERISTICS
                 SENSEMODE,-                    ;SENSE MODE
                 SETMODE,-                      ;SET MODE
                 WRITELBLK,-                    ;WRITE LOGICAL BLOCK
                 WRITEPBLK,-                    ;WRITE PHYSICAL BLOCK
                 WRITEVBLK>                     ;WRITE VIRTUAL BLOCK
        FUNCTAB LP_WRITE,<WRITELBLK,WRITEPBLK,WRITEVBLK> ;WRITE FUNCTIONS
        FUNCTAB LP_SETMODE,<SETCHAR,SETMODE> ;SET CHARACTERISTICS FUNCTIONS
        FUNCTAB +EXE$SENSEMODE,-                ;
                <SENSECHAR,-                    ;SENSE CHARACTERISTICS
                 SENSEMODE>                     ;SENSE MODE
```

FUNCTION DECISION TABLE (FDT) ROUTINE

Set Mode FDT Routine

```
        .PAGE
        .SBTTL  SET CHARACTERISTICS AND SET MODE FUNCTION PROCESSING
;+
; LP_SETMODE - SET CHARACTERISTICS AND SET MODE FUNCTION PROCESSING
;
; THIS ROUTINE IS CALLED FROM THE FUNCTION DECISION TABLE DISPATCHER TO PROCESS
; A SET MODE FUNCTION TO A LINE PRINTER.
;
; INPUTS:
;

;       R0 = SCRATCH.
;       R1 = SCRATCH.
;       R2 = SCRATCH.
;       R3 = ADDRESS OF I/O REQUEST PACKET.
;       R4 = CURRENT PROCESS PCB ADDRESS.
;       R5 = ASSIGNED DEVICE UCB ADDRESS.
;       R6 = ADDRESS OF CCB.
;       R7 = I/O FUNCTION CODE.
;       R8 = FUNCTION DECISION TABLE DISPATCH ADDRESS.
;       R9 = SCRATCH.
;       R10 = SCRATCH.
;       R11 = SCRATCH.
;       AP = ADDRESS OF FIRST FUNCTION DEPENDENT PARAMETER.
;
; OUTPUTS:
;
;       THE SPECIFIED CHARACTERISTICS ARE MOVED INTO THE DEVICE UCB AND THE
;       I/O IS COMPLETED.
;-

LP_SETMODE:                                     ;SET MODE FUNCTION PROCESSING
        MOVL    P1(AP),R1                       ;GET ADDRESS OF CHARACTERISTICS
        IFNORD  #8,(R1),20$                     ;CAN CHARACTERISTICS QUADWORD BE READ?
        PUSHL   R3                              ;SAVE PACKET ADDRESS
        MOVAB   UCB$L_LP_MUTEX(R5),R0           ;GET ADDRESS OF UCB MUTEX
        JSB     G^SCH$LOCKW                     ;LOCK UCB FOR WRITE ACCESS
        CMPL    #IO$_SETMODE,R7                 ;SET MODE FUNCTION?
        BEQL    10$                             ;IF EQL YES
        MOVW    (R1),UCB$B_DEVCLASS(R5)         ;SET DEVICE CLASS AND TYPE
10$:    MOVW    2(R1),UCB$W_DEVBUFSIZ(R5)       ;SET DEFAULT BUFFER SIZE
        MOVL    4(R1),UCB$L_DEVDEPEND(R5)       ;SET DEVICE CHARACTERISTICS
        JSB     G^SCH$UNLOCK                    ;UNLOCK UCB
        POPL    R3                              ;RESTORE PACKET
        MOVZWL  #SS$_NORMAL,R0                  ;SET NORMAL COMPLETION STATUS
        JMP     G^EXE$FINISHIOC                 ;
20$:    MOVZWL  #SS$_ACCVIO,R0                  ;SET ACCESS VIOLATION STATUS
        JMP     G^EXE$ABORTIO                   ;
```

Write FDT Routine

```
        .PAGE
        .SBTTL  WRITE FUNCTION PROCESSING
;+
; LP_WRITE - WRITE FUNCTION PROCESSING
;
; THIS ROUTINE IS CALLED FROM THE FUNCTION DECISION TABLE DISPATCHER TO PROCESS
; A WRITE PHYSICAL, WRITE LOGICAL, OR WRITE VIRTUAL FUNCTION TO A LINE PRINTER.
;
; INPUTS:
;
;       R0 = SCRATCH.
;       R1 = SCRATCH.
;       R2 = SCRATCH.
;       R3 = ADDRESS OF I/O REQUEST PACKET.
;       R4 = CURRENT PROCESS PCB ADDRESS.
;       R5 = ASSIGNED DEVICE UCB ADDRESS.
;       R6 = ADDRESS OF CCB.
;       R7 = I/O FUNCTION CODE.
;       R8 = FUNCTION DECISION TABLE DISPATCH ADDRESS.
;       R9 = SCRATCH.
;       R10 = SCRATCH.
;       R11 = SCRATCH.
;       AP = ADDRESS OF FIRST FUNCTION DEPENDENT PARAMETER.
;

; OUTPUTS:
;
;       THE FUNCTION PARAMETERS ARE CHECKED AND THE USER'S BUFFER IS FORMATTED
;       AND COPIED INTO A SYSTEM BUFFER FOR PROCESSING BY THE LINE PRINTER
;       DRIVER.
;-

LP_WRITE:                                       ;WRITE FUNCTION PROCESSING
        CLRL    R11                             ;CLEAR TOTAL NUMBER OF OVERHEAD BYTES
        CLRL    R10                             ;ASSUME WRITE PASS ALL FUNCTION
FORMAT: MOVL    FP,SP                           ;REMOVE ALL TEMPORARIES FROM STACK
        PUSHR   #^M<R3,R4,R5,R6,R7,AP>          ;SAVE REGISTERS
        MOVL    P1(AP),R8                       ;GET STARTING ADDRESS OF USER BUFFER
        MOVZWL  P2(AP),R9                       ;GET LENGTH OF USER BUFFER
        CMPL    #IO$_WRITEPBLK,R7               ;WRITE PHYSICAL BLOCK?
        BEQL    10$                             ;IF EQL YES
        MOVL    P4(AP),IRP$B_CARCON(R3)         ;INSERT CARRIAGE CONTROL INFORMATION
        JSB     G^TTS$CARRIAGE                  ;TRANSLATE CARRIAGE CONTROL INFORMATION
        MOVZBL  IRP$B_CARCON(R3),R0             ;GET NUMBER OF PREFIX CONTROL BYTES
        MOVZBL  IRP$B_CARCON+2(R3),R1           ;GET NUMBER OF SUFFIX CONTROL BYTES
        ADDL    R0,R1                           ;CALCULATE NUMBER OF CARRIAGE CONTROL BYTES
        MOVAB   32(R1)[R11],R10                 ;CALCULATE TOTAL NUMBER OF OVERHEAD BYTES
10$:    TSTL    R9                              ;ANY BUFFER SPECIFIED?
        BEQL    20$                             ;IF EQL NO
        MOVQ    R8,R0                           ;RETRIEVE BUFFER PARAMETERS
        JSB     G^EXE$WRITECHK                  ;CHECK ACCESSIBILITY OF USER BUFFER
20$:    MOVAB   12(R9)[R10],R1                  ;CALCULATE LENGTH OF BUFFER REQUIRED
        JSB     G^EXE$BUFFRQUOTA                ;CHECK IF PROCESS HAS SUFFICIENT QUOTA
        BLBC    R0,45$                          ;IF LBC QUOTA CHECK FAILURE
        JSB     G^EXE$ALLOCBUF                  ;ALLOCATE BUFFER FOR LINE PRINTER OUTPUT

        BLBC    R0,45$                          ;IF LBC ALLOCATION FAILURE
        MOVL    (SP),R3                         ;RETRIEVE ADDRESS OF I/O PACKET
        MOVL    R2,IRP$L_SVAPTE(R3)             ;SAVE ADDRESS OF BUFFERED I/O PACKET
        SUBW    R1,PCB$W_BYTCNT(R4)             ;ADJUST BUFFERED I/O QUOTA
        MOVW    R1,IRP$W_BOFF(R3)               ;SET NUMBER OF BYTES CHARGED TO QUOTA
        CLRL    IRP$L_MEDIA(R3)                 ;CLEAR LINE FEED COUNT IN PACKET
        MOVW    R9,IRP$W_BCNT(R3)               ;INSERT SIZE OF USER BUFFER
        MOVAB   12(R2),R2                       ;GET ADDRESS OF BUFFER DATA AREA
        MOVAB   UCB$L_LP_MUTEX(R5),R0           ;GET ADDRESS OF UCB MUTEX
        JSB     G^SCH$LOCKW                     ;LOCK UCB FOR WRITE ACCESS
        CMPL    #IO$_WRITEPBLK,R7               ;WRITE PASS ALL?
        BEQL    50$                             ;IF EQL YES
        SUBW    #12,R1                          ;CALCULATE ACTUAL LENGTH OF DATA AREA
        MOVZBL  UCB$B_LP_CURSOR(R5),R4          ;GET CURRENT HORIZONAL CARRIAGE POSITION
        MOVZWL  UCB$W_DEVSTS(R5),R6             ;GET CURRENT CARRIAGE RETURN PENDING FLAG
        MOVZBL  UCB$B_LP_LINCNT(R5),R7          ;GET CURRENT LINE ON PAGE
        MOVZBL  UCB$W_DEVBUFSIZ(R5),R10         ;GET WIDTH OF PRINTER CARRIAGE
        MOVL    #^X20,AP                        ;ASSUME PRINTER DOES NOT HAVE LOWER CASE
        BBC     #LP$V_LOWER,UCB$L_DEVDEPEND(R5),35$ ;IF CLR, NO LOWER CASE
        CLRL    AP                              ;SET FOR PRINTER WITH LOWER CASE
35$:    BSBB    70$                             ;INSERT PREFIX CARRIAGE CONTROL
30$:    DECL    R9                              ;ANY MORE BYTES TO TRANSFER TO SYSTEM BUFFER?
        BLSS    40$                             ;IF LSS NO
        MOVZBL  (R8)+,R0                        ;GET NEXT BYTE FROM USER BUFFER
        BSBB    WRITE_BYTE                      ;WRITE BYTE IN SYSTEM BUFFER
        BRB     30$                             ;
40$:    BSBB    80$                             ;INSERT SUFFIX CARRIAGE CONTROL IN BUFFER
        SUBL    IRP$L_SVAPTE(R3),R2             ;CALCULATE LENGTH OF OUTPUT PLUS HEADER
        SUBW3   #12,R2,IRP$L_MEDIA+2(R3)        ;CALCULATE ACTUAL LENGTH OF OUTPUT BUFFER
        MOVB    R4,UCB$B_LP_CURSOR(R5)          ;SAVE CURRENT HORIZONAL CARRIAGE POSITION
        INSV    R6,#V_CRPEND,#1,UCB$W_DEVSTS(R5) ;SAVE CARRIAGE RETURN PENDING

        MOVB    R7,UCB$B_LP_LINCNT(R5)          ;SAVE CURRENT LINE ON PAGE
        BRB     60$                             ;
45$:    POPR    #^M<R3,R4,R5,R6,R7,AP>          ;RESTORE REGISTERS
        JMP     G^EXE$ABORTIO                   ;
50$:    MOVW    R9,IRP$L_MEDIA+2(R3)            ;INSERT NUMBER OF BYTES TO PRINT
        MOVC    R9,(R8),(R2)                    ;MOVE CHARACTERS TO SYSTEM BUFFER
60$:    POPR    #^M<R3,R4,R5,R6,R7,AP>          ;RESTORE REGISTERS
        PUSHL   R3                              ;SAVE ADDRESS OF I/O PACKET
        MOVAB   UCB$L_LP_MUTEX(R5),R0           ;GET ADDRESS OF UCB MUTEX
        JSB     G^SCH$UNLOCK              ;UNLOCK UCB
        POPL    R3                        ;RESTORE ADDRESS OF I/O PACKET
        JMP     G^EXE$QIODRVPKT           ;QUEUE I/O PACKET TO DRIVER

;
; SUBROUTINE TO INSERT CARRIAGE CONTROL IN BUFFER
;

70$:    MOVZBL  IRP$B_CARCON(R3),-(SP)    ;GET NUMBER OF CHARACTERS TO OUTPUT
        BEQL    100$                      ;IF EQL NONE
        MOVZBL  IRP$B_CARCON+1(R3),R0     ;GET CHARACTER TO OUTPUT
        BRB     85$                       ;
80$:    MOVZBL  IRP$B_CARCON+2(R3),-(SP)  ;GET NUMBER OF CHARACTERS TO OUTPUT
        BEQL    100$                      ;IF EQL NONE
        MOVZBL  IRP$B_CARCON+3(R3),R0     ;GET CHARACTER TO OUTPUT
85$:    BNEQ    90$                       ;IF NEQ CHARACTER SPECIFIED
        MOVZBL  #C_CR,R0                  ;GET CARRIAGE RETURN
        BSBB    WRITE_BYTE                ;WRITE BYTE IN SYSTEM BUFFER
        MOVZBL  #C_LF,R0                  ;GET LINE FEED
90$:    BSBB    WRITE_BYTE                ;WRITE BYTE IN SYSTEM BUFFER
        SOBGTR  (SP),90$                  ;ANY MORE LEFT TO INSERT?
100$:   TSTL    (SP)+                     ;REMOVE COUNT FROM STACK
        RSB                               ;
        .PAGE
        .SBTTL  WRITE BYTE INTO SYSTEM BUFFER
;
; SUBROUTINE TO FORMAT AND FILL SYSTEM BUFFER WITH LINE PRINTER OUTPUT ONE BYTE
; AT A TIME.
;

WRITE_BYTE:                               ;WRITE BYTE INTO BUFFER
        CMPL    #^A/ /,R0                 ;CONTROL CHARACTER?
        BGTRU   40$                       ;IF GTRU YES
        BBSC    #V_CRPEND,R6,60$          ;IF SET, CARRIAGE RETURN PENDING
        CMPB    #^A/`/,R0                 ;POSSIBLY LOWER CASE CHARACTER?
        BGTRU   10$                       ;IF GTRU NO
        CMPB    #^X7F,R0                  ;DELETE CHARACTER?
        BEQL    30$                       ;IF EQL YES
        SUBL    AP,R0                     ;CONVERT CHARACTER TO UPPER CASE


10$:    CMPL    R4,R10                    ;STILL ROOM ON CURRENT LINE?
        BGTRU   30$                       ;IF GTRU NO
        INCL    R4                        ;INCREMENT HORIZONAL POSITION
20$:    DECL    R1                        ;ANY ROOM LEFT IN SYSTEM BUFFER?
        BLSS    150$                      ;IF LSS NO
        MOVB    R0,(R2)+                  ;INSERT CHARACTER IN SYSTEM BUFFER
30$:    RSB                               ;

;
; CONTROL CHARACTER ENCOUNTERED
;

40$:    CMPL    #C_CR,R0                  ;CARRIAGE RETURN?

        BLSSU   50$                       ;IF LSS NO
        BGTRU   70$                       ;IF GTRU NO
        BBS     #LP$V_CR,UCB$L_DEVDEPEND(R5),140$ ;IF SET, CARRIAGE RETURN REQUIRED
        BISL    #M_CRPEND,R6              ;SET CARRIAGE RETURN PENDING
        RSB                               ;
50$:    BBCC    #V_CRPEND,R6,20$          ;IF CLR, CARRIAGE RETURN NOT PENDING
60$:    PUSHL   R0                        ;SAVE CURRENT CHARACTER
        MOVZBL  #C_CR,R0                  ;GET CARRIAGE RETURN CHARACTER
        BSBB    140$                      ;INSERT CARRIAGE RETURN IN BUFFER
        POPL    R0                        ;RETRIEVE CURRENT CHARACTER
        BRB     WRITE_BYTE                ;

;
; CHARACTER IS A TAB, LINE FEED, VERTICLE TAB, OR FORM FEED
;

70$:    CMPL    #C_TAB,R0                 ;TABULATION CHARACTER?
        BGTRU   50$                       ;IF GTRU NO
        BLSSU   80$                       ;IF LSSU NO

;
; CHARACTER IS A TAB
;
```

```
        BBSC    #V_CRPEND,R6,60$            ;IF SET, CARRIAGE RETURN PENDING
        PUSHAB  8(R4)                       ;CALCULATE NEXT TAB POSITION
        BICL    #7,(SP)                     ;CLEAR EXCESS BITS
        SUBL    R4,(SP)                     ;CALCULATE BLANK COUNT
        MOVZBL  #^A/ /,R0                   ;SET SPACE CHARACTER
        BRB     100$                        ;

;
; CHARACTER IS A LINE FEED, VERTICLE TAB, OR FORM FEED
;

80$:    CMPL    #C_VT,R0                    ;VERTICLE TAB?
        BEQL    50$                         ;IF EQL YES
        BGTRU   110$                        ;IF GTRU LINE FEED

;
; CHARACTER IS A FORM FEED
;

        MOVZBL  UCB$L_DEVDEPEND+3(R5),R0 ;GET NUMBER OF LINES PER PAGE
        SUBL3   R4,R0,-(SP)                 ;CALCULATE NUMBER OF LINES TO END OF PAGE
        BBC     #LP$V_MECHFORM,UCB$L_DEVDEPEND(R5),90$ ;IF CLR, NO MECHANICAL FEED
        ADDL    (SP)+,IRP$L_MEDIA(R3)       ;UPDATE NUMBER OF LINES PRINTED
        MOVZBL  #C_FF,R0                    ;SET FORM FEED CHARACTER
        BRB     120$                        ;
90$:    MOVZBL  #C_LF,R0                    ;SET LINE FEED CHARACTER
100$:   BSBW    WRITE_BYTE                  ;INSERT BYTE IN SYSTEM BUFFER
        SOBGTR  (SP),100$                   ;ANY MORE BYTES TO INSERT?
        TSTL    (SP)+                       ;REMOVE LOOP COUNT FROM STACK
        RSB                                 ;

;
; CHARACTER IS A LINE FEED
;

110$:   INCL    R7                          ;INCREMENT LINE POSITION ON PAGE
        INCL    IRP$L_MEDIA(R3)             ;INCREMENT NUMBER OF LINES PRINTED

        CMPB    R7,UCB$L_DEVDEPEND+3(R5) ;END OF PAGE?
        BNEQ    130$                        ;IF NEQ NO
120$:   CLRL    R7                          ;CLEAR LINE POSITION ON PAGE
130$:   BICL    #M_CRPEND,R6                ;CLEAR CARRIAGE RETURN PENDING
140$:   CLRL    R4                          ;CLEAR HORIZONAL POSITION
        BRW     20$                         ;

;
; OUTPUT WILL NOT FIT IN ALLOCATED BUFFER
;

150$:   MOVL    IRP$L_SVAPTE(R3),R0         ;GET ADDRESS OF BUUFER TO DEALLOCATE
        CLRL    IRP$L_SVAPTE(R3)            ;INDICATE NO BUFFER ALLOCATED
        MOVZWL  IRP$W_SIZE(R0),R10          ;SAVE SIZE OF BUFFER
        JSB     G^EXE$DEANONPAGED           ;DEALLOCATE BUFFER
        MOVAB   -4*6(FP),SP                 ;REMOVE ALL TEMPORARIES FROM STACK
        POPR    #^M<R3,R4,R5,R6,R7,AP>      ;RESTORE REGISTERS
        ADDW    R10,PCB$W_BYTCNT(R4)        ;ADJUST BYTE COUNT QUOTA
        ADDL    #32,R11                     ;ADJUST COUNT OF OVERHEAD BYTES
        PUSHL   R3                          ;SAVE ADDRESS OF I/O PACKET
        MOVAB   UCB$L_LP_MUTEX(R5),R0       ;GET ADDRESS OF UCB MUTEX
        JSB     G^SCH$UNLOCK                ;UNLOCK UCB
        POPL    R3                          ;RESTORE ADDRESS OF I/O PACKET
        BRW     FORMAT                      ;TRY AGAIN
```

I/O Entry Routine Code

```
        .PAGE
        .SBTTL  LINE PRINTER DRIVER
;+
; STARTIO - START I/O OPERATION ON LINE PRINTERS
;
; THIS ROUTINE IS ENTERED WHEN THE ASSOCIATED UNIT IS IDLE AND A PACKET
; IS AVAILABLE.
;
; INPUTS:
;
;       R3 = ADDRESS OF I/O REQUEST PACKET.
;       R5 = UCB ADDRESS FOR IDLE UNIT.
```

```
;
; OUTPUTS:
;
;       NO EXPLICIT OUTPUTS - THE UNIT IS IN WAITING FOR INTERRPUT STATE
;                             OR THE I/O IS COMPLETE.
;-

STARTIO:
        MOVL    UCB$L_IRP(R5),R3            ;RETRIEVE ADDRESS OF I/O PACKET
        MOVW    IRP$L_MEDIA+2(R3),UCB$W_BOFF(R5) ;SET NUMBER OF CHARACTERS TO PRINT
        MOVL    UCB$L_SVAPTE(R5),R3         ;GET ADDRESS OF SYSTEM BUFFER
        MOVAB   12(R3),R3                   ;GET ADDRESS OF DATA AREA
        MOVL    UCB$L_CRB(R5),R4            ;GET ADDRESS OF CRB
        MOVL    @CRB$L_INTD+VEC$L_IDB(R4),R4 ;GET DEVICE CSR ADDRESS

;
; START NEXT OUTPUT SEQUENCE
;

10$:    ADDL3   #LP_DBR,R4,R0               ;CALCULATE ADDRESS OF DATA BUFFER REGISTER
        MOVZWL  UCB$W_BOFF(R5),R1           ;GET NUMBER OF CHARACTERS REMAINING
        MOVW    #^X8080,R2                  ;GET CONTROL REGISTER TEST MASK
        BRB     25$                         ;
20$:    BITW    R2,(R4)                     ;PRINTER READY OR HAVE PAPER PROBLEM?
        BLEQ    30$                         ;IF LEQ NOT READY OR PAPER PROBLEM

        MOVB    (R3)+,(R0)                  ;OUTPUT NEXT CHARACTER
25$:    SOBGEQ  R1,20$                      ;ANY MORE CHARACTERS TO OUTPUT?
        BRB     70$                         ;

;
; PRINTER IS NOT READY OR HAS PAPER PROBLEM
;

30$:    BNEQ    40$                         ;IF NEQ PAPER PROBLEM
        ADDW3   #1,R1,UCB$W_BOFF(R5)        ;SAVE NUMBER OF CHARACTERS REMAINING
        DSBINT                              ;DISABLE INTERRUPTS

        BISB    #^X40,LP_CSR(R4)            ;SET INTERRUPT ENABLE
        WFIKPCH 40$,#12                     ;WAIT FOR INTERRUPT
        IOFORK                              ;CREATE A FORK PROCESS
        BRB     10$                         ;

;
; PRINTER HAS PAPER PROBLEM
;

40$:    CLRB    UCB$B_LP_OFLCNT(R5)         ;CLEAR OFFLINE COUNTER
        ADDW3   #1,R1,UCB$W_BOFF(R5)        ;SAVE NUMBER OF CHARACTERS REMAINING
50$:    CLRW    LP_CSR(R4)                  ;DISABLE PRINTER INTERRUPT
        SETIPL  UCB$B_FIPL(R5)              ;LOWER TO FORK LEVEL
        TSTW    LP_CSR(R4)                  ;PRINTER STILL HAVE PAPER PROBLEM?
        BGTR    STARTIO                     ;IF GTR NO
        BBS     #UCB$V_CANCEL,UCB$W_STS(R5),80$ ;IF SET, CANCEL I/O OPERATION
        ACBB    #15,#1,UCB$B_LP_OFLCNT(R5),60$ ;SKIP UNTIL TIMEOUT
        CLRB    UCB$B_LP_OFLCNT(R5)         ;RESET COUNTER
        PUSHR   #^M<R3,R4>                  ;SAVE REGISTERS
        MOVZBL  #MSG$_DEVOFFLIN,R4          ;SET UP MESSAGE TYPE
        MOVAB   G^SYS$GL_OPRMBX,R3          ;ADDRESS TARGET MAILBOX
        JSB     G^EXE$SNDEVMSG              ;SEND MESSAGE IGNORE ERROR
        POPR    #^M<R3,R4>                  ;RESTORE REGISTERS
60$:    DSBINT                              ;DISABLE INTERRUPTS
        WFIKPCH 50$,#2                      ;WAIT FOR AN INTERRUPT OR TIMEOUT 2 SEC
        IOFORK                              ;CREATE FOR PROCESS
        BRB     50$                         ;

;
; I/O OPERATION SUCCESSFULLY COMPLETED
;

70$:    MOVZWL  #SS$_NORMAL,R0              ;SET NORMAL COMPLETION STATUS
        CLRW    UCB$W_BOFF(R5)              ;CORRECT REMAINING CHARACTER COUNT
        BRB     90$                         ;

;
; I/O OPERATION CANCELED
;

80$:    MOVZWL  #SS$_ABORT,R0               ;SET OPERATION ABORTED STATUS
90$:    MOVL    UCB$L_IRP(R5),R3            ;RETRIEVE ADDRESS OF I/O PACKET
        MOVZWL  IRP$L_MEDIA(R3),R1          ;GET NUMBER OF LINES PRINTED
        SUBW    UCB$W_BOFF(R5),UCB$W_BCNT(R5) ;CALCULATE NUMBER OF CHARACTERS
        INSV    UCB$W_BCNT(R5),#16,#16,R0 ;INSERT NUMBER OF CHARACTERS IN STATUS
        REQCOM                            ;COMPLETE I/O REQUEST
```

Interrupt Service Routine Code

```
        .PAGE
        .SBTTL  LP11/LS11/LV11 LINE PRINTER INTERRUPT DISPATCHER
;+
; LP$INT - LP11/LS11/LV11 LINE PRINTER INTERRUPT DISPATCHER.
;
; THIS ROUTINE IS ENTERED VIA A JSB INSTRUCTION WHEN AN INTERRUPT OCCURS ON AN
; LP11/LS11/LV11 LINE PRINTER CONTROLLER. THE STATE OF THE STACK ON ENTRY IS:
;
;       00(SP) = ADDRESS OF IDB ADDRESS.
;       04(SP) = SAVED R3.
;       08(SP) = SAVED R4.
;       12(SP) = SAVED R5.
;       16(SP) = INTERRUPT PC.
;       20(SP) = INTERRUPT PSL.
;
; INTERRUPT DISPATCHING OCCURS AS FOLLOWS:
;
;       IF THE INTERRUPT IS EXPECTED, THEN THE DRIVER IS CALLED AT ITS INTERRUPT
;       WAIT ADDRESS. ELSE THE INTERRUPT IS DISMISSED.
;-

LP$INT::                                  ;ENTRY FROM DISPATCH
        MOVL    @(SP)+,R3                 ;GET ADDRESS OF IDB
        MOVQ    IDB$L_CSR(R3),R4          ;GET CONTROLLER CSR AND OWNER UCB ADDRESS
        BBCC    #UCB$V_INT,UCB$W_STS(R5),10$ ;IF CLR, INTERRUPT NOT EXPECTED
        CLRW    (R4)                      ;DISABLE OUTPUT INTERRUPTS
        MOVL    UCB$L_FR3(R5),R3          ;RESTORE REMAINDER OF DRIVER CONTEXT
        JSB     @UCB$L_FPC(R5)            ;CALL DRIVER AT INTERRUPT WAIT ADDRESS
10$:    MOVQ    (SP)+,R0                  ;RESTORE REGISTERS
        MOVQ    (SP)+,R2                  ;
        MOVQ    (SP)+,R4                  ;
        REI                               ;
        .PAGE
        .SBTTL  LINE PRINTER UNIT INITIALIZATION
;+
; LP_LX11_INIT - LINE PRINTER UNIT INITIALIZATION
;
; THIS ROUTINE IS CALLED AT SYSTEM STARTUP AND AFTER A POWER FAILURE. THE
; ONLINE BIT IS SET FOR THE SPECIFIED UNIT.
;
; INPUTS:
;
;       R5 = ADDRESS OF DEVICE UCB.
;
; OUTPUTS:
;
;       THE ONLINE BIT IS SET IN THE DEVICE UCB AND THE ADDRESS OF THE UCB
;       IS FILLED INTO THE IDB OWNER FIELD.
;-

LP_LX11_INIT:                             ;LINE PRINTER UNIT INITIALIZATION
        BISW    #UCB$M_ONLINE,UCB$W_STS(R5) ;SET UNIT ONLINE
        MOVL    UCB$L_CRB(R5),R0          ;GET ADDRESS OF CRB
        MOVL    CRB$L_INTD+VEC$L_IDB(R0),R0 ;GET ADDRESS OF IDB
        MOVL    R5,IDB$L_OWNER(R0)        ;SET ADDRESS OF DEVICE UCB
LP_LX11_CINIT:                            ;NULL CONTROLLER INITIALIZATION
        RSB                               ;
LP_END:                                   ;ADDRESS OF LAST LOCATION IN DRIVER

        .END
```

CHAPTER OVERVIEW

It is frequently important that processes be able to communicate with one another; to pass data or to share a resource or data structure. This chapter expands on the interprocess communications section of Chapter 1. It describes in detail the use of common event flags and mailboxes as structures by which processes pass status information and data to one another. Also explained, are the use of glóbal sections for sharing physical pages of memory and the lock management services for generic multiprocess resource sharing.

Topics are:

- Common Event Flags
- Mailboxes
- DECnet/VAX
- Global Sections
- Lock Management Service

CHAPTER 14

INTERPROCESS COMMUNICATION

INTRODUCTION

The VAX/VMS operating system provides interprocess communication facilities for synchronizing execution, for sending messages, and for sharing common data. The six communication techniques utilized by cooperating processes are:

- Common event flags
- Mailboxes
- DECnet
- Shared data and code
- Lock semaphors
- Shared disk files

Common event flags are associated with group identification. The others are more general purpose facilities which can be limited or unlimited in access.

COMMON EVENT FLAGS

Event flags are status posting bits that allow the programmer to incorporate a variety of control functions within the program. Event flag services capabilities include:

- Set or clear specific flags
- Test the current status of flags
- Place a program in a wait state pending the setting of a specific flag or a group of flags

Moreover, event flags can be used in common by more than one process, provided the cooperating processes are in the same group. Thus, if an application has been developed that requires the simultaneous execution of several processes, event flags can be used to establish communication and to synchronize their activity. A common event flag cluster is composed of 32 event flags, which can be assigned any meaning for the processes in the group. Four clusters are available to any process at any one time. Two are for process-local functions, two are for interprocess communication. As it may with local event flags, a process can read its group's common event flags, can set or clear them, can wait for a particular event flag to be set, or for any or all flags in the cluster to be set.

Associated with each common event flag cluster is a software control structure known as a common event block (CEB). The common event block provides the system with necessary information, such as the creator's user identification code, the cluster name and size in bytes, process protection, and a count of processes in wait queue.

System Services For Event Flag Handling
VAX/VMS system services for the handling of event flags and clusters provide the capability to perform the functions as described below.

Six general event flag services operate on both local and common event flags:

$SETEF	Set Event Flag
$CLREF	Clear Event Flag
$READEF	Read Event Flag
$WAITFR	Wait for Single Event Flag
$WFLOR	Wait for Logical OR of Event Flags
$WFLAND	Wait for Logical AND of Event Flags

Common event flag clusters must be associated before they can be used. Three services control their use:

$ASCEFC	Associate Common Event Flag Cluster
$DACEFC	Disassociate Common Event Flag Cluster
$DLCEFC	Delete Common Event Cluster

These services are explained in Chapter 11.

MAILBOXES

Mailboxes are virtual devices that can be used for communication between processes. Actual data transfer is accomplished by using higher-level language I/O statements, Record Management Services, or directly with the I/O system services. When a mailbox is created, a channel is assigned to it for use by the creating process.

The Create Mailbox and Assign Channel ($CREMBX) system service creates the mailbox. The $CREMBX system service can optionally create a user-specified logical name and assign it the physical mailbox name created. Other processes can then use the $ASSIGN or $OPEN system services (or higher level language OPEN statements), specifying the logical name, to assign other channels to the mailbox. A process can also determine the physical mailbox name by translating the logical name (with the $TRNLOG service), or it can call the Get I/O Channel Information ($GETCHN) service to obtain the unit number and device name.

Mailboxes are either temporary or permanent; user privileges are required to create either type. $CREMBX enters the logical name and equivalence name for a temporary mailbox in the group logical name table of the process that created it. The system deletes a temporary mailbox when no more channels are assigned to it.

The $CREMBX system service enters the logical name and equivalence name for a permanent mailbox in the system logical name table. Permanent mailboxes continue to exist until they are specifically marked for deletion with the Delete Mailbox ($DELMBX) system service.

The maximum number of messages and the maximum size of message that can be written to a mailbox can be specified when the mailbox is created. A mailbox can be protected when it is created, just as a device or disk volume can be protected when mounted.

The system uses mailboxes internally for interprocess messages between system processes, and between system processes and user processes. The following services create special mailbox messages for system processes:

- Send Message to Accounting Manager ($SNDACC)
- Send Message to Operator ($SNDOPR)
- Send Message to Symbiont Manager ($SNDSMB)

When a process creates another process, it can specify the name of a mailbox that is to receive the termination status when the created process is deleted.

When a channel is assigned to a terminal or a network link, a process can specify the name of a mailbox that is to receive unsolicited input or high priority network messages. When the message is written to the mailbox, an asynchronous system trap (AST) will be delivered, eliminating the need for an outstanding read to each terminal or network link.

DECNET/VAX

VAX/VMS provides the same interfaces for interprocess communication on a single node as DECnet/VAX provides in a multi-node configuration. This communication mechanism can be an effective alternative to mailboxes. Not only is it as easy to use, but it is more flexible because it also allows applications to expand to a multi-node environment without modification. DECnet/VAX is described in Chapter 7.

GLOBAL SECTIONS

The system supports a high degree of code and data sharing through the use of global sections. A global section is a copy of a portion of an image or data file that can be included in a process virtual address space at runtime.

Global sections either are created dynamically by a process or are permanently installed in the system. Dynamically created global sections are mapped into processes that reference them, and deleted when no more references are made to them. Permanent global sections may be known shareable images created using the linker, or may be created by program calls to system services. They are loaded into and removed from memory dynamically as references are made to them.

A global section can be created as a read-only or read/write global section to protect code and data.

Normally, only one copy of a global section actually resides in memory while cooperating processes reference it. However, should a global section contain pre-initialized data that processes using the data are expected to change, the global section can be declared to be copy-on-reference. This enables each process to have its own copy of these pages.

A read/write global section may include a demand allocate zero-filled page. When a process references the global section for the first time, zero-filled pages are mapped into its virtual address space. Such pages are created dynamically and eliminate the necessity of filling up space on secondary storage with pages of zeros. (Typical use of the demand allocate zero-filled page is buffers or stacks.)

A process can map to a global section explicitly or implicitly. The image itself can issue a Map Global Section system service, or it can reference a known shareable image. When an image references a known shareable image, the linker does not include the global section in the image. When the image is executed, the image activator calls the Map Global Section system service on behalf of the image. For example, the Run Time Procedure Library is a known shareable image implicitly mapped into images that reference library procedures. The use of known shareable images significantly reduces the size of programs using common library procedures.

Each process that maps a global section into its virtual address space can have a different access privilege to the section, depending on the protection code to the global section. When a global section is created, it is assigned a user identification code (UIC) identifying the group and family member to which the global section belongs, and a protection code identifying the read and write access privileges of processes in the system. Global sections can therefore be shared among processes in the system, or shared among processes within a group and protected from all other processes, or shared among processes within a single job and protected from all other jobs.

VAX/VMS LOCK MANAGEMENT SERVICES

For cooperating processes sharing resources (for example, files, data structures or I/O devices), VAX/VMS provides a lock management, or semaphore, facility. The VAX/VMS lock management services, like the common event flag services, provide a tool for synchronizing process action. While common event flags can be used only by processes within the same group, the lock management service can operate on a system-wide basis. In fact, VAX-11 RMS uses this service to regulate file sharing.

The lock management services allow users to develop complex resource-sharing applications, such as database systems, by providing user-determinable granularity in defining and locking a resource, and a flexible choice of locking modes.

Common Namespace

The lock management service does not directly control access to resources; rather, it provides a mechanism for assigning names to resources, which are represented in a common namespace. Processes request access to a resource name in a common namespace and cooperating processes understand that when they are granted access to a resource name, they may then access the resource itself.

The resource namespace is tree structured; that is, it is hierarchically organized with an arbitrary number of levels. Each name may have a number of "branch" names which in turn may have a number of branches, and so on, in a way that parallels the organization of the actual resource. (See figure 14-1)

A lock may be granted on a name at any level of the namespace hierarchy. The only names affected by the lock are the specified name

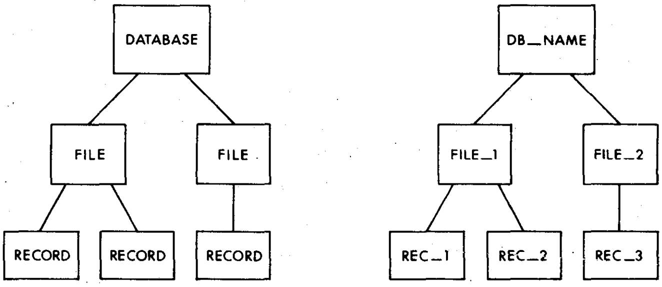


Figure 14-1 Paralleling Resource and Resource Namespace

and those names beneath it. In this way, a resource can be defined and access controlled to any depth of granularity required by an application, while allowing concurrent access by multiple processes.

Lock Modes
Concurrency can be increased further by an appropriate selection of lock modes. There are six lock modes to choose from, each allowing a different sharing scheme with other cooperating processes. See Table 14-1.

Deadlock Detection
The lock management services also provide deadlock detection. A deadlock occurs when a group of locks are waiting for each other in a circular fashion; for example, a Process A is waiting for a resource that Process B has, and Process B is waiting for a resource that Process C has, while Process C is in turn waiting for a resource that process A has. If a deadlock situation occurs, the lock management service selects one of the processes as the "victim", does not grant that process the lock it has been waiting for, and indicates to the process that the lock has been denied because a deadlock condition exists. The process can then do whatever cleanup is necessary and unlock (or convert its lock on) the resource it has — thus breaking the deadlock.

Using the Lock Management Services
A process requests a lock on a resource by issuing an $ENQ system service request, where it specifies the resource name and the type of lock, or lock mode, it wants. If another process has an incompatible lock on that resource, then the request is queued and the process can go about its business until the request is granted. Optionally, the process can request a lock and wait with the $ENQW system service; in fact, the options available for synchronization with a lock request are the same as with a QIO request.

When the lock is granted, the process is signalled that the resource is available, and it goes ahead and accesses the resource in accordance with its declared lock-mode.

Once it has completed its action on the resource, the process can either change its lock mode to a less exclusive one (for example, from an exclusive to a concurrent read), which would allow greater sharing while retaining access, or it can release the resource altogether by issuing a $DEQ system service on that lock.

Another useful option of the lock management service is the use of blocking AST's. If a process needs exclusive access to a resource, but wants to know when another process is trying to access it, it can request that a blocking AST be issued in that circumstance via a parameter specified in the $ENQ system service call. This mechanism can optimize sharing and potentially increase performance.

A process may have many locks — some granted, some waiting. The limit on the number of locks a process may have is determined by the lock quota assigned and defined in the User Authorization File.

NOTE

The locking service is not a general data protection mechanism. Since it does not directly control access to a resource, a non-cooperating process could access the resouce independently. For the lock management services to be effective, all processes must use agreed upon conventions.

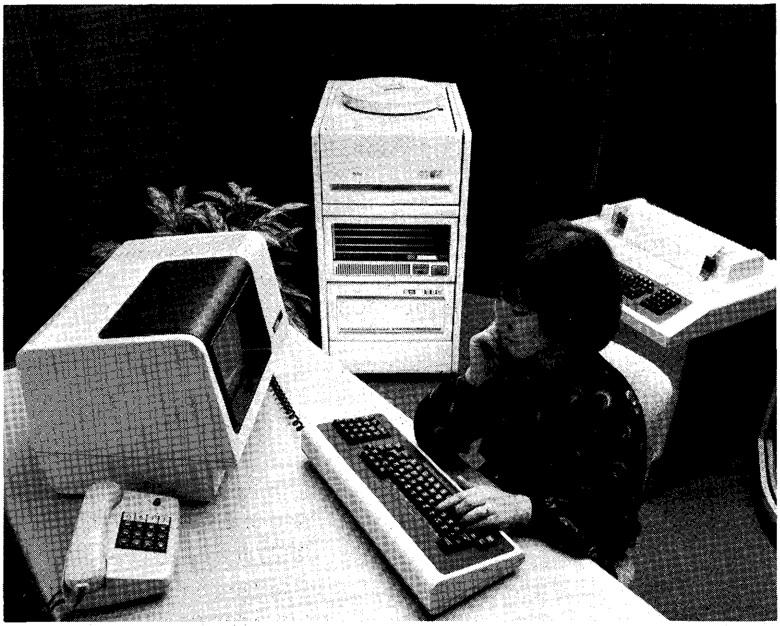
LOCK MODE	**DESIRED ACCESS**	**DESIRED SHARING**	**INDICATION**
Null (NL)	None	Read Write	Used as an interest lock, and to prevent namespace entry from being deleted
Concurrent Read (CR)	Read	Read Write	Used in conjunction with more restrictive locks at a lower level
Concurrent Write (CW)	Write	Read Write	Used in conjunction with more restrictive locks at a lower level
Protected Read (PR)	Read	Read	Traditional "share" lock
Protected Write (PW)	Write	Read	Traditional "update" lock
Exclusive (EX)	Read Write	None	Traditional "Exclusive" lock

Lock Modes

Table 14 - 1

SHARED DISK FILES

Compared to the three methods listed above, the use of shared files is more indirect and carries more restrictions. The VAX-11 Record Management Service (RMS) is the standard vehicle for file sharing. Information on file sharing using RMS can be found in the RMS section of Chapter 12, **Input/Output Services.**



CHAPTER OVERVIEW

One very important consideration in the design of the VAX computers and the VAX/VMS operating system was compatibility with the large base of PDP-11 computers and programs that already exists. Fulfillment of this goal helps protect customer investment in PDP-11 hardware and software, reduce retraining costs, and simplify the task of moving programs to VAX systems. In addition, it allows a VAX system to be used as a development environment for certain non-privileged PDP-11 tasks, namely those that will run under an RSX-11M operating system. In this chapter, compatibility is discussed.

Topics include:

- The Application Migration Executive (AME)
- Compatibility Mode
- Transportable Languages
- Compatibility Considerations

CHAPTER 15
PDP-11 COMPATIBILITY

INTRODUCTION

A major feature of the VAX/VMS operating system is its compatibility with the PDP-11 family of minicomputers.

The VAX system provides PDP-11 compatibility including the following features:

- The execution of a subset of PDP-11 instructions in VAX/VMS compatibility mode
- An RSX-11M compatibility mode Applications Migration Executive (AME) allowing most RSX-11M/S non-privileged tasks to run with minimal or no modification
- An RSX-11M/S Host Development Package that allows creation and partial testing of RSX-11M/S tasks as well as sysgening RSX-11M/S systems
- Transportable source-level programs in high-level languages
- Files-11 On-Disk Structure Level 1
- Compatible RMS file access methods on both RSX-11M and VMS operating systems
- DIGITAL Command Language (DCL) and RSX-11 MCR (Monitor Console Routine) command language

The VAX instruction set is a powerful extension of the PDP-11 instruction set. Therefore, the programmer with previous PDP-11 knowledge who is developing new VAX applications will experience a high level of adaptability. Similarly, VAX high-level languages are closely compatible with those of the PDP-11 family.

The VAX/VMS operating system may effectively serve as a high-performance RSX-11M/S program development system. RSX-11M/S programs can be edited, compiled, and linked on a VAX/VMS system. In addition, the task can be partially debugged on a VAX/VMS system. That is, the software development can largely be accomplished on a VAX system and need only migrate to the target RSX-11M/S system for final debugging and execution.

The VAX/VMS operating system, through DECnet communications software, supports downline loading of RSX-11S systems and RSX-11M tasks. However, the VAX/VMS system and the RSX-11M/S system must have either a common communications link or a mass storage peripheral of the same type on both systems in order to transfer the RSX-11M task or RSX-11S system to the target machine.

Under the VAX/VMS operating system, programs may execute in either of two modes, native or compatibility. Native mode programs use the VAX instruction set and execute under the VAX/VMS operating system. Compatibility mode programs, however, are those which can execute on other PDP-11 systems.

In order to provide cross-development and migration capability, an RSX-11M Applications Migration Executive has been implemented that allows most non-privileged RSX-11 tasks to execute on the VAX/VMS system with little or no modification to the task image. The Applications Migration Executive is part of the VAX/VMS system. It supports a mapped RSX-11M environment without supporting the directives to manipulate Program Logical Address Space (PLAS), DECnet calls, or RMS-11 file sharing. Under control of the Applications Migration Executive, the user's task is mapped into virtual memory and executes in compatibility mode. When the task issues an RSX-11M executive directive, a trap is initiated which automatically places the processor in native mode. The Applications Migration Executive then determines what directive the user is attempting to accomplish, and executes a VAX/VMS system service of equivalent function (except for the memory management directives, RMS-11 file sharing, and DECnet I/O calls, which are not supported). If there is no equivalent VAX/VMS function, such as the RSX-11M directive to "get sense switches," the executive will return an error code but will not cause the task to abort.

The PDP-11 compatibility mode environment will support the FORTRAN IV compiler as well as many existing PDP-11 utilities.

When programming in compatibility mode, certain points should be established:

- Users' images are limited to 64 Kbyte executable segments
- Compatibility mode and native mode code cannot be shared, i.e., compatibility mode routines and native mode routines cannot call each other directly
- It is possible for compatibility and native mode programs to share data and to communicate through mailboxes
- The Applications Migration Executive does not support the memory management directives
- The Applications Migration Executive does not support the RSX-11 DECnet I/O functions or RMS-11 file sharing
- Because the system environments differ, applications that involve cooperating tasks may require modification

IMPLEMENTATION CONSIDERATIONS

The processor can execute user mode PDP-11 instruction streams in the context of a process. The operating system supplements this feature by substituting its functionally equivalent system services for many of the RSX-11M operating system executive directives that user mode tasks may call. This enables the system to execute such non-privileged RSX-11M task images as:

- MACRO-11 assembler
- PDP-11 FORTRAN IV compiler
- RSX-11M program development and file management utilities, including the task builder and text editor

In addition, the operating system supports the FCS (File Control Services), RMS-11 and RMS-11K record management services procedures and can read and write the RSX/IAS on-disk structure, Files-11 Level 1 (ODS-1). Programs that call FCS or RMS-11 services can access Files-11 file-structured volumes.

The operating system contains two command language interpreters, MCR and DCL. The VMS MCR can accept many of the RSX-11M MCR commands, either typed directly on a terminal, or submitted as command files.

Any task linked for the RSX-11M operating system will run, assuming the task is non-privileged.

Any RSX-11M task image can be executed in compatibility mode without relinking, provided that it was linked with the RSX-11M task builder and it meets the following requirements:

- It must not execute PDP-11 privileged instructions
- It must have been built for a mapped system
- It must not depend on 32-word memory granularity
- It must not require mapping into the executive or I/O page
- It must not use the memory management executive directives
- It must not use the CONNECT executive directive
- It must not rely on environmental features of RSX-11M that the VAX/VMS operating system does not support, such as significant events or a task's STOP bit
- It must not use DECnet communications software or RMS-11 file sharing
- It only accesses ODS-I volumes

IAS or RSX-11D tasks that meet these requirements can also be executed. They must first be built with the RSX-11M task builder. For programs that do not meet these requirements, the VAX/VMS operating system provides the program development utilities (for example, the MACRO assembler and the task builder) for modifying programs to execute in compatibility mode.

File System and Data Management
Magnetic tape and Files-11 disk volumes can be transported between VAX/VMS and RSX systems. The VAX/VMS system can read and write both Files-11 Level 1 (ODS-1) disk structures and the RMS Level 2 disk structures (ODS-2). The extend access protection field in ODS-1 is used for execute access protection in ODS-2.

Overlays, Shareable Regions and PLAS
The VAX/VMS operating system supports the use of overlays and shared regions by RSX-11M images running in compatibility mode. RSX-11M images produced using the overlay descriptor language or the RSX-11M task builder run under the VAX/VMS operating system. The VAX/VMS operating system loads overlays at the appropriate point in image execution from the image file.

The VAX/VMS operating system also supports RSX-11M image use of dynamically loaded shared regions. RSX-11M images can access both shared commons and libraries. Permanently available shared regions are identified to the VAX/VMS operating system by the system manager.

The VAX/VMS operating system does not support the RSX-11M memory management directives used to extend the program logical address space (PLAS) of an RSX-11M task. Any task image issuing a memory management directive under the VAX/VMS operating system receives an error status return.

Command Languages
The operating system's MCR command language interpreter accepts both a subset of DCL (DIGITAL Command Language) commands and a subset of the RSX-11M MCR (Monitor Console Routine) commands. The VAX/VMS MCR command language consists of two types of commands:

- Those that duplicate an RSX-11M command
- Those that provide a VAX/VMS function using an MCR-like syntax

Thus, MCR allows the user access to a full range of VAX/VMS functions. There is no need to change to native DCL to perform commonly needed functions.

VAX/VMS support of RSX-11M task images provides an interface to the operating system that is similar to that found in RSX-11M operating system.

RSX-11M Directive Requests

In an RSX-11M system, a task image interfaces with the operating system by issuing directive requests. As a result of a directive request, the RSX-11M system performs the desired function and returns control to the task. The VAX/VMS operating system duplicates the task/system interaction achieved by a directive request in the RSX-11M system. When an RSX-11M task issues a directive, the hardware traps to the VAX/VMS operating system. With a few exceptions, including RSX-11M memory management directives, the VAX/VMS operating system duplicates the requested RSX-11M function with either of the following results:

- The RSX-11M directive function is duplicated in the VAX/VMS operating system and the task continues execution
- The VAX/VMS operating system cannot duplicate the requested function but does take the necessary action to allow continued task execution

The VAX/VMS operating system duplicates the function of a majority of RSX-11M directives. For example:

- A checkpoint enable/disable directive is interpreted as the Set Swap mode system service
- The send/receive directives are translated into mailbox write/read system services. Native mode and compatibility mode images can communicate using mailboxes
- The event flag directives are for the most part identical. Native mode and compatibility mode images can communicate using common event flags, provided they are in the same group
- A Logical Unit Number (LUN) assignment directive is interpreted as an Assign Channel system service

If the VAX/VMS operating system cannot duplicate an RSX-11M directive, it is because of differences in the basic concepts of the two systems, that is, differences in the environments provided by the two systems. For example:

- A task image is allowed to declare a significant event, but the directive is ignored. Therefore, the VAX/VMS operating system cannot declare a significant event upon directive request. Rather, it performs no operation and returns a success status to the requesting task, which continues execution normally
- A set priority directive is ignored, since the scheduling priorities ranges are different. To run at a given priority, the image must be run in the context of a process created for a user given that priority in the user authorization file

For the most part, however, many RSX-11M and VAX/VMS program environment characteristics correspond. Tasks can hibernate, receive asynchronous system traps, and schedule wake requests. Synchronous system trap routines can be declared as condition handlers for trace traps, breakpoint traps, illegal instruction traps, memory protection violations, and odd address errors.

RSX-11M Directives
The VAX/VMS operating system will support the following RSX-11M directives:

ABRT$	Abort Task
ALUN$	Assign Task
ASTX$	AST Service Exit
CLEF$	Clear Event Flag
CMKT$	Cancel Mark Time Requests
CRGF$	Create Group Global Event Flags
DSAR$	Disable AST Recognition
DSCP$	Disable Checkpointing
ELGF$	Eliminate Group Global Event Flags
ENAR$	Enable AST Recognition
ENCP$	Enable Checkpointing
EXIF$	Exit If
EXIT$S	Task Exit
EXST$	Exit With Status
EXTK$	Extend Task
GLUN$	Get LUN Information
GMCR$	Get MCR Command Line
GPRT$	Get Partition Parameters
GTIM$	Get Time Parameters
GTSK$	Get Task Parameters
MRKT$	Mark Time
QIO$	Queue I/O Request
QIOW$	Queue I/O Request and Wait

RCVD$	Receive Data
RCVX$	Receive Data or Exit
RDAF$	Read All Event Flags
RDXF$	Read Extended Event Flags
RQST$	Request
RSUM$	Resume
RUN$	Run
SDAT$	Send Data
SETF$	Set Event Flag
SFPA$	Specify Floating Point Processor Exception AST
SPND$	Suspend
SPRA$	Specify Power Recovery AST
SPWN$	Spawn
SRDA$	Specify Receive Data AST
STLO$	Stop for Logical OR of Event Flags
STOP$	Stop
STSE$	Stop for Single Event Flag
SVDB$	Specify SST Vector Table for Debugging Aid
SVTK$	Specify SST Vector Table for Task
USTP$	Unstop Task
WTLO$	Wait for Logical OR of Event Flags
WTSE$	Wait for Single Event Flag

The VAX/VMS operating system does **not** support a number of RSX-11M directives, principally because of different techniques of memory management in PDP-11 and VAX hardware.

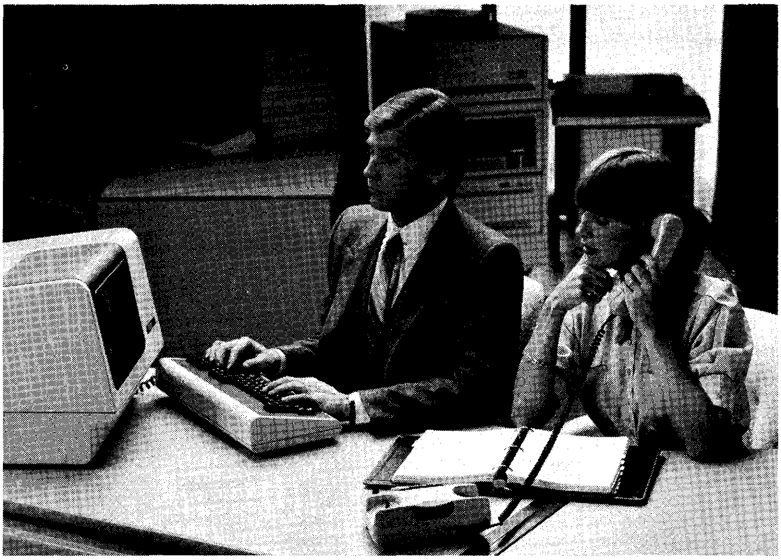
The VAX/VMS operating system returns an error status of IE.SDP (invalid directive) to any RSX-11M image that issues an unsupported directive.

The AME supports floating point instructions by emulating them in native mode.

The VAX processor does not have sense switches. Therefore, the VAX/VMS operating system handles the Get Sense Switch directive in the same manner as the RSX-11M operating system does for a system that disallowed access to sense switches during system generation. It returns the DSW status IE.SDP.

Some of the remaining unsupported directives are RSX-11M memory management directives. They are not supported because the VAX/VMS operating system controls memory management very differently from the way that the RSX-11M operating system does. The CONNECT directive is also not supported.

PART IV
SITE CONSIDERATIONS



CHAPTER OVERVIEW

This brief chapter lists some of the powers and responsibilities of a VAX/VMS system manager, from the initial bootstrapping to the assignment of privileges and quotas to individuals or classes of users. The VAX virtual memory operating system gives complete authority to the system manager, including the ability to deny or limit access, to imitate any user's identification code, and to assign priorities to real-time and interactive processes. But the operating system supplies tools and defaults that help make the job quite easy.

Topics include:

- Getting the System Up and Running
- User Accounts
- Monitoring System Activity
- Protection and Privilege
- Error Handling
- User Environment Test Package

CHAPTER 16

THE SYSTEM MANAGER

In a VAX/VMS operating system installation, the system manager controls two main areas:

- Decisions that optimize the performance and efficiency of the system
- Procedures that affect the overall management of the system

Assisting the manager in controlling these areas are many and varied tools supplied by DIGITAL, so that what might be complicated in some operating systems is, in the VAX/VMS operating system, straightforward and easy. In fact, system management need not be exercised full-time by a single person dedicated to that job; it may be shared by several persons, some of whom may serve additionally as system operators. However arranged, the management of a system has as its ultimate goal delivering efficient economical service to all users. The VAX/VMS operating system helps by providing such features as self-installation of layered products (e.g., higher level language compilers), autoconfiguration, a User Environment Test Package (UETP), and easy adjustment of parameter files.

Practically speaking, the job of the system manager is best defined in terms of the following six categories of tasks a manager typically oversees.

- Getting the system up and running
- Setting up users' accounts
- Managing public files and volumes
- Controlling the overall performance of the system
- Monitoring system activity
- Recognizing and dealing with errors and failures

GETTING THE SYSTEM UP AND RUNNING

Unlike some other operating systems, the VAX/VMS operating system makes it easy for the manager to get the system up. The time needed for this task is, therefore, reduced, while the degree of expertise required by the manager is lessened.

The VAX/VMS operating system comes pre-built. It is self-installing and autoconfiguring. That means that any valid VAX hardware configuration can be supported by the VAX/VMS operating system without special configuration considerations. Many of the parameters can be adjusted to suit specific needs. For example, the working set size can be increased or decreased from the default working set size by a simple instruction. In addition, tailoring of the parameter file to satisfy a customer's specific needs can go on while the system is running, so that there is no downtime nor lost productivity.

Updating the system is accomplished simply: DIGITAL supplies a command procedure to apply the update. The system manager merely runs the command procedure. The same is true for the installation of optional software—such as DIGITAL layered products—that a user wants. Even the installation of customer-supplied application and system software—including user-written device drivers—is quite easy, because the VAX/VMS system provides a "friendly" environment.

User Environment Test Package (UETP)

When a VAX/VMS system is first installed and bootstrapped, an installation verification package can be used to supplement the DIGITAL Field Service diagnostics. Such a package, the User Environment Test Package, is part of the VAX/VMS operating system. When run, it adapts to any VAX configuration and assures the manager that hardware and the operating system are working properly together. Errors are reported to the console terminal from which UETP was run and stored in a log file. In addition, the UETP serves as a quick check to help determine the cause when programs that once worked stop working or when any condition arises that gives the manager reason to doubt the functional integrity of the system.

The UETP performs three functions:

1. Exercises major peripherals
2. Validates VAX/VMS system services
3. Tests major VAX/VMS software components (e.g., VAX-11 RMS (Record Management Services) and the VAX-11 SORT/MERGE utility)

The UETP is fully automatic and requires no user interaction once started. Errors indicated during one test will not affect another test, although the same problem might occur in different parts of the UETP.

While the UETP is thorough, it is not exhaustive and should not be construed as fully diagnostic or as replacing a diagnostic test. It does not, for example, test layered products such as optional language compilers. Such products may have their own installation verification test packages in their distribution kits. The UETP is a functionality test that the system manager may employ to get a quick check of the system's condition.

SETTING UP AND USING A SYSTEM OF ACCOUNTS

Some of the main reasons for setting up a meaningful system of users' accounts are:

- To identify the users of the system
- To define important relationships among the users of the system. For example, groups of users may share data and other files. These relationships are the basis of a system of file protection, interprocess communication, and system accounting
- To grant to some users the privileges necessary to perform sensitive system functions, and thus to restrict other users from performing those functions
- To set limits on the use of reusable system resources.
- To give users priorities in using the system

Many of the account parameters may be assigned by default, as can a large number of other values in a VAX/VMS system; or the manager may want to assign particular values to particular users. In either case, a User Authorization File is set up for each user and contains critical accounting information.

The User Authorization File (UAF)

The User Authorization File (UAF) is one of the most important data structures with which the system manager must be concerned. The UAF contains one record for each user of the system; in effect, it defines the user to the system.

Besides the users' records, the UAF also contains a default value record and a system manager's record. In most cases, the manager will simply allow the default values for various parameters to stand. Thus, the manager may elect to choose characteristics only when warranted by a special case.

Why is the UAF so important in controlling the performance of the VAX/VMS system? Simply stated, each process in a VAX/VMS system is associated with a user. Each user is allotted system resources and is given a priority and privileges, and all such attributes are specified in the user's record in the UAF. When a user logs onto the system, a process is created on behalf of that user. The process acquires the characteristics of the user. These are the same characteristics as the system manager put into or defaulted into the user's record in the UAF.

Each user's record in the UAF contains the following types of information:

1. User's identification
   a. User name
   b. Password
   c. User identification code (UIC)
   d. Account name
2. User's default directory name and default device name
3. User's default command interpreter name
4. User's allotment of system resources
5. User's privileges
6. User's base priority

Through the User Authorization Program (AUTHORIZE), the system manager may add, delete, modify, or display records in the UAF.

Groups

A group is a collection of users whose processes normally have access to each others' files, file-structured volumes, mailboxes, shared pages of memory, common event flags, and the group logical name table. In addition, such processes may have special privileges to exercise control over each other. Therefore, the establishment of groups principally concerns interprocess communication and control.

In setting up a group, the system manager aims toward two goals: 1) to facilitate sharing of data and cooperation between users and their processes; 2) to protect users from unauthorized access to their processes and data.

The importance of properly setting up groups should not be underestimated. As the system is increasingly used and as more and more files and protected data structures arise, relationships among group members, processes, devices, and data structures grow inevitably more complex. In time, it becomes harder to redefine the basic relationships among the users.

A user's membership in a particular group is defined by the User Identification Code (UIC). The UIC consists of two octal numbers, each ranging from 0 to 377. The first is a group number; the second is a member number.

The UIC is the basis of the VAX/VMS data protection scheme, and it is one of the factors (along with privilege) that govern the ways in which processes can interact with one another. The system manager's assignment of UICs, therefore, should involve two important considerations:

1. Which users should be allowed to share data and file access, and which should not?
2. Which processes should be allowed to cooperate, and which should not?

Protection, and Owner, System, Group, and World

For purposes of data protection, four different categories of users are defined. They are:

1. Owner—users whose UICs are identical with the UIC of the owner of the data structure or device. For example, the owner of a file is usually the creator of that file.
2. Group—users of the system whose group numbers are the same.
3. System—users of the system with group numbers of octal 10 or less. Certain privileges appertain to system users.
4. World—all users.

All users potentially enjoy four types of access to protected data structures and devices: read (R), write (W), execute (E), and delete (D). Generally speaking, any category of user can be permitted or denied any type of access to data structures and devices. There are, however, exceptions, because not all types of access apply to all protected items. For example, execute access applies only to files that contain executable program images.

The scheme for protecting file-structured volumes is similar to that for protecting files, except that execute (E) access to a volume gives the user the right to create files on that volume.

Limits, Priority, and Privilege

The attributes which the system manager may assign or merely default to when creating the user's account record are:

- Limits on the use of reusable system resources
- The base priority used in scheduling the processes that the system creates for that user
- Privileges of using restricted and sensitive system functions

Limits

Limits are set on system resources that can be reused. An example is the amount of memory that a process can have in use for queued I/O requests. Most limit restrictions actually are placed on the use of system dynamic memory.

Usually the system manager simply assigns the default values of limits. However, the defaults can easily be overridden.

Priority

A user's priority is the base priority that is used in scheduling any process the system creates for that user. There are 32 levels of software priority in the VAX/VMS operating system. For normal processes, the priority range is 0 to 15; for realtime processes, it is 16 through 31.

Processes with realtime priority are scheduled strictly according to base priority. But processes with normal prority are scheduled according to a slightly different principle, one that promotes overlapping of computation and I/O activities. This scheduling is all done transparently to the programmer and manager.

Privileges

Many system services are protected by privileges which restrict their availability to certain users. These restrictions are intended to protect the integrity of performance of the operating system, and thus the integrity of service provided to all users. The manager grants privileges to each user depending upon two factors: 1) whether the user has the skill and experience to use the system service without disrupting the whole system; 2) whether the user has a legitimate need for the privilege.

Accounting for the Use of System Resources

For accounting purposes, the VAX/VMS system itself creates and maintains records of the use of system resources. These records are kept in an accounting log file.

Using the detailed accounting log records provided by the system, the system manager or a system programmer can establish programs for reporting on the use of system resources and for billing.

Because the users of system resources are identified in two ways, reports on the use of system resources and bills for the use of system resources can easily be generated in either of two ways: by user name or by account name.

The User Authorization Program

The User Authorization Program (AUTHORIZE) is a system utility required to maintain the User Authorization File (UAF). The AUTHORIZE program lets the manager:

- Create the UAF if one does not exist. A newly created UAF contains only the default value record and the system management account record; no users are yet known to the system

- Define a new user to the system by creating a record for that user in the UAF and thus granting privileges and specifying limits and priority
- Take away a user's right to the system by deleting that user's record from the UAF
- Change the default record of the UAF
- Change a user's privileges, limits, or priority by modifying that user's record in the UAF
- Display all information about a user's account, with the exception of the user's password
- Make a listing of all records in the UAF

For a description of commands and options, consult the documentation delivered with the system.

MANAGING PUBLIC FILES AND VOLUMES

Typically, overall planning and management of a system of public files and volumes are among the most important responibilities of the system manager. The aspects of public files and volumes management that the system manager is most concerned with are:

- Initializing and mounting public volumes
- Regularly backing-up public files and volumes
- Installing frequently used or privileged executable images as known images or images that may be shared at runtime
- Installing frequently used shareable images as permanent global sections or images that may be shared at runtime
- Establishing systemwide logical names needed for running the executable images provided by DIGITAL and for running other images available to all users at an installation
- Establishing disk quotas

Initializing and Mounting Public Volumes

Public volumes contain public files, which normally must be available to most users of a system. Public volumes may also contain files that users create for their own private use or for general use.

Public volumes contain the following kinds of public files supplied by DIGITAL.

- The operating system itself in executable form and files related to the operating system
- Utility programs in executable form. Utilities available from DIGITAL are self-installing

- Diagnostic and test programs in executable form and files related to these programs. Such packages as the User Environment Test Package are bundled in the system and installed along with it
- Various system libraries: macro libraries, object module libraries, and shared runtime libraries
- Text files; for example, the system error message file and help files, installed with the system
- Optional software in executable form, plus related libraries and other files. Some, such as language processors, are self-installing

In addition, the system manager can include on public volumes files that are unique to an installation. These typically are files that must be accessible to many, if not all, users of the installation. The system manager can also permit any user to create, catalog, and store files on a public volume.

Mounting a disk volume establishes a relationship among the volume, the device on which it is physically mounted, and one or more processes that may gain access to it.

Backing-Up Public Files and Volumes

To prevent the inadvertent loss or destruction of valuable information stored on disk file volumes, the system manager usually establishes a policy and a schedule for regularly backing-up files on public volumes.

The BACKUP utility allows users to create back-up copies of files and directories and to restore them. It can back up entire volume sets in one operation or perform selective back-ups by file or date. Wildcarding is available, as well as several file selection qualifiers.

The BACKUP utility is intended primarily for use by system managers and operators; however it can be used by individual users to make personal back-up copies and to transport files.

There are two kinds of back-ups of public disk files and volumes: 1) selective, or partial, backups and 2) system, or all-inclusive, backups. Either type of backup can be done either to disk or magnetic tape.

Installing Known Images and Creating Permanent Global Sections

The system manager can significantly improve system performance by installing certain executable and shareable images as known images and by creating permanant global sections.

There are two reasons for installing known images:

1. To permit systemwide sharing of images that are frequently used by more than one user at a time
2. To make image files more quickly accessible

Typically, the kinds of executable images that are installed as known images are:

1. Images that need more privileges than are commonly granted to users who need to execute them
2. Images that are executed frequently
3. Images that are executed by more than one user at a time

A number of images supplied by DIGITAL are ordinarily installed as known images in a site-independent startup procedure.

Shareable image sections produced by the linker are almost identical with executable image sections, except that they cannot be executed by use of the DIGITAL Command Language command RUN. They can, however, be linked with object modules to create executable images.

Sharing common procedures leads to three significant improvements in system performance:

1. Reduction of disk storage requirements
2. Reduction of physical memory requirements
3. Reduction of the amount of paging I/O needed

Assigning System Logical Names

A logical name is a user-specified name that may be equivalent to a file specification or to some portion of a file specification, such as a device name. A systemwide logical name is simply a logical name that can be referred to by all users of the system and by all processes created for those users.

Making sure that all needed system logical names have been assigned to equivalence names is an important part of the manager's role.

Except for default logical names, system logical names that are needed by nearly all or by all VAX/VMS installations are assigned in the startup command procedure file,STARTUP.COM. DIGITAL provides this as part of all software release distribution kits.

Usually the system manager is responsible for establishing the system logical names that are unique to an installation. As a rule, these names are assigned by the use of ASSIGN commands in the site-specific startup command procedure.

OVERALL CONTROL OF THE SYSTEM

Two important ways in which the manager exerts control over the behavior of a VAX/VMS system are:

- By maintaining command procedures of initialization commands that are essential to the proper operation of the system
- By establishing output spooling and setting up and controlling batch queues, print queues, and terminal queues

STARTUP.COM and SYSTARTUP.COM
The command procedure STARTUP.COM is a startup file that executes automatically immediately after the VAX/VMS operating system has been booted. This startup file is supplied by DIGITAL and contains commands for performing site-independent operations that must occur if the system is to run properly. The operations include assigning system logical names, installing images as known images, building the I/O database, and loading I/O drivers.

On the other hand, the command procedure SYSTARTUP.COM is a command file that the manager may tailor to the needs of a specific installation. Typically, this file contains commands for performing such operations as setting the characteristics of terminals and other devices, purging the operator's log file, and announcing that the system is up and running.

Spooling and Batch, Print, and Terminal Queues
Usually the manager performs the following four closely related functions, which establish spooled devices and control queues:

- Establishing input and output spooling. The VAX/VMS operating system supports input spooling of batch job files and transparent spooling of output files for lineprinters and terminals. Using DIGITAL Command Language commands, the system manager can easily specify which output devices are to be spooled
- Creating and controlling batch queues
- Creating and controlling print queues
- Creating and controlling terminal queues

A system manager need not learn the inner workings of spooling and queuing, but a pragmatic knowledge of how to establish spooled devices and how to create control queues is useful for efficient management of the system.

Spooling
Spooling is the technique of using a highspeed storage device to buffer data passing between lowspeed I/O devices and highspeed main memory. The lowspeed devices, which can be either the ultimate sources or the ultimate destinations of buffered I/O data, are called spooled devices; the highspeed mass storage devices are called intermediate devices.

Typically, the system manager chooses lowspeed peripheral devices to include in the system's basic complement of spooled devices. At a minimum, the system manager should see that at least one lineprinter is set spooled when the system is started-up. In a system with only one lineprinter, this is the default system printer. The system manager need not set a card reader spooled, because card readers are spooled by default.

Batch Queues

Batch jobs can enter the VAX/VMS system and be queued for initiation in two ways:

1. As batch job files submitted by use of the $JOB command from a card reader. These batch job files are spooled onto disk by an input symbiont and placed in a batch queue according to their priority. Unless the $JOB card specifies otherwise, the name of the batch queue is SYS$BATCH (by default). From the batch queue, batch jobs are selected for execution
2. As command procedure disk files submitted by use of the SUBMIT command. These files are also placed in a batch queue and selected for execution according to their priority. Again, by default, this is the batch queue SYS$BATCH

Print Queues

Unless a lineprinter is associated with a physical queue (a queue that has the same name as the lineprinter) and unless that queue has been started (along with a generic print queue), no queued output can occur on that lineprinter.

Print jobs are queued for processing by an output symbiont in one of two ways: without the direct intervention of a user (implicitly) or with the direct intervention of a user (explicitly).

When an implicitly spooled print file destined for a spooled printer is closed, the file is placed in a print queue. Both the spooling of the output file to an intermediate device and the subsequent queuing of a job consisting of this file occur without the direct intervention of a user.

Through the PRINT command a user can explicitly queue a disk file or several files for printing. The disk file or files specified by the PRINT command are queued as a print job; if several files make up a print job they will be printed together.

MONITORING SYSTEM ACTIVITY

Using the MONITOR Utility Program

The VAX/VMS operating system collects data on how the system is being used and how it responds to users' requests. The MONITOR utility program can be used to examine the collected data at a specifiable interval and produce three forms of output:

- Statistical display on any Digital terminal
- A statistical summary file
- A binary recording file

Displays take the forms of bar graphs and tables. MONITOR can be used to observe the statistics of a running system or to read collected data and play it back.

These statistics are useful for two purposes: 1) to aid system developers in understanding how the system operates; 2) to aid system managers in improving system performance. The types of information collected and displayed are:

- Network activity
- File system statistics
- I/O system activity
- Use of the lock management services
- Use of processor modes
- Page management statistics
- Non-paged pool statistics
- Activity in the scheduler state queues
- Principal users of CPU paging and I/O resources
- System process activity

Each time MONITOR is run, it starts accumulating a new set of performance measurement statistics.

Below is a sample of one of the several displays that a system manager can call up when he wants to examine the system statistics.

```
                                 VAX/VMS Monitor Utility
                                 TIME IN PROCESSOR MODES (%)
             +------+                 12-FEB-1982
             | CUR  |                   14:36:12
             +------+

                                        0%        25%       50%       75%      100%
                                        + - - - - + - - - - + - - - - + - - - - -+
 Interrupt Stack                   9    |***
                                        |         |         |         |          |
 Kernel Mode                       8    |***
                                        |         |         |         |          |
 Executive Mode                    3    |*
                                        |         |         |         |          |
 Supervisor Mode                        |
                                        |         |         |         |          |
 User Mode                        76    |******************************
                                        |         |         |         |          |
 Compatibility Mode                     |
                                        |         |         |         |          |
 Idle Time                         2    |
                                        |         |         |         |          |
                                        |         |         |         |          |
                                        + - - - - + - - - - + - - - - + - - - - -+
```

Figure 16-1

This particular graph measures current percentage of time in the processor modes available.

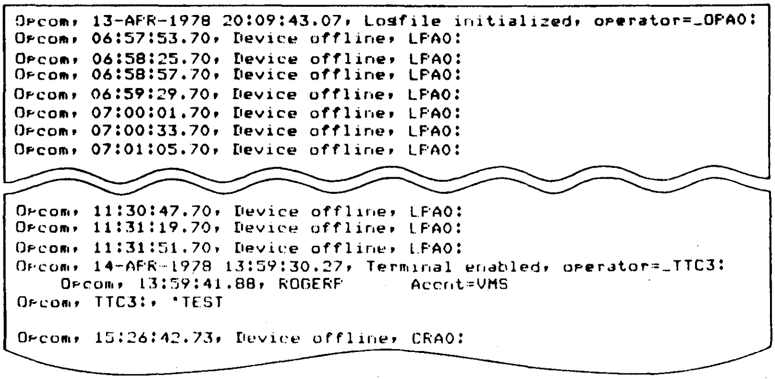
The processor modes monitored are:

- Interrupt Stack—percent of time processor was executing on interrupt stack
- Kernel Mode—percent of time processor was executing in kernel mode (does not include time on interrupt stack)
- Executive Mode—percent of time processor was executing in executive mode
- Supervisor Mode—percent of time processor was executing in supervisor mode
- User Mode—percent of time processor was executing in user mode (does not include time in compatibility mode)
- Compatibility Mode—percent of time processor was executing compatibility mode user images
- Idle Time—percent of time processor was executing the null process

The Operator's Log File

The operator's log file is a system management tool that is useful in anticipating and preventing failures of both the hardware and the software. By regularly examining it, the manager can often detect tendencies or trends toward failures and corrective action can be taken before such failures occur.

The system operator should, therefore, print out copies of the operator's log file regularly, and the system manager should retain copies for reference. Figure 3-2 below shows some typical messages in the operator's log files.



```
Opcom, 13-APR-1978 20:09:43.07, Logfile initialized, operator=_OPA0:
Opcom, 06:57:53.70, Device offline, LPA0:
Opcom, 06:58:25.70, Device offline, LPA0:
Opcom, 06:58:57.70, Device offline, LPA0:
Opcom, 06:59:29.70, Device offline, LPA0:
Opcom, 07:00:01.70, Device offline, LPA0:
Opcom, 07:00:33.70, Device offline, LPA0:
Opcom, 07:01:05.70, Device offline, LPA0:

Opcom, 11:30:47.70, Device offline, LPA0:
Opcom, 11:31:19.70, Device offline, LPA0:
Opcom, 11:31:51.70, Device offline, LPA0:
Opcom, 14-APR-1978 13:59:30.27, Terminal enabled, operator=_TTC3:
    Opcom, 13:59:41.88, ROGERP      Accnt=VMS
Opcom, TTC3:, 'TEST

Opcom, 15:26:42.73, Device offline, CRA0:
```

Figure 16-2

RECOGNIZING AND DEALING WITH ERRORS

Error Logging

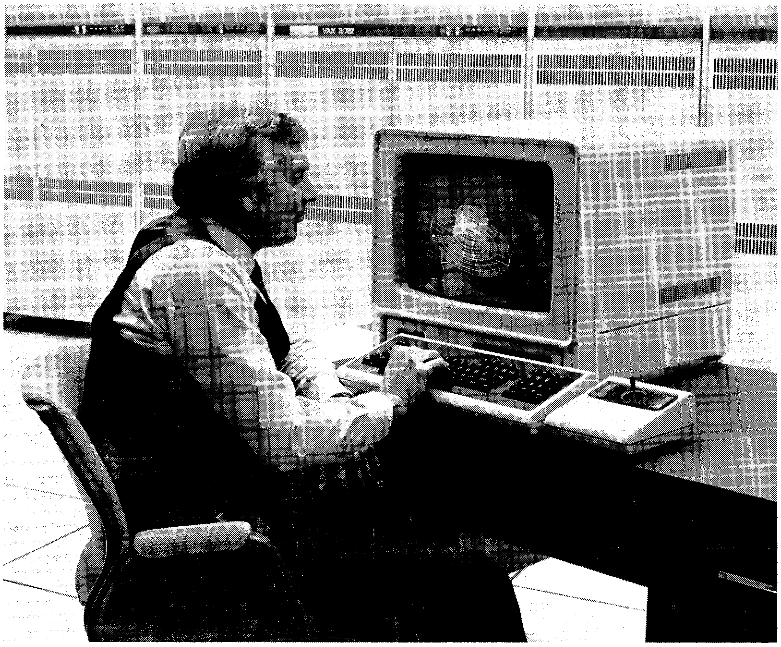
The purpose of the error logging facility is to gather and maintain information on system errors and events as they occur; this information provides a detailed record of system errors. By running the report generator program SYE, the manager or a DIGITAL Field Service Representative can obtain a report of the errors and events that have occurred within a specified period of time.

Using Error Reports

The error reports generated by SYE are useful tools in two basic ways:

- Reports aid preventive maintenance by identifying areas within the system that show potential for failure
- Reports speed the diagnosis of a failure by documenting the errors and events that led up to them

The detailed contents of the reports are most meaningful to DIGITAL Field Service personnel. However, the system manager can use the reports as an important indicator of the system's reliability. For example, when a report shows that a particular device is producing a relatively high number of errors, the system manager can consult DIGITAL Field Service. By running a diagnostic program to investigate the device, field service can attempt to isolate the source of the errors. Once identified, the source of the errors can possibly be eliminated and a failure averted.



CHAPTER OVERVIEW

The most powerful VAX system is the VAX-11/782 Attached Processor System. Although multi-processing under VAX/VMS is transparent to the applications programmer, there are some unique software considerations. This chapter gives a brief overview of how VAX/VMS performs multiprocessing and what this means to the VAX-11/782 system manager and system programmer.

Topics include:

- Multi-processing in general
- Software
- Programming considerations
- System management considerations

CHAPTER 17

ATTACHED PROCESSOR SYSTEM SUPPORT

ATTACHED PROCESSOR SYSTEM

There are essentially two classes of multiprocessing systems: tightly-coupled and loosely-coupled. In a loosely coupled system, each processor executes a separate copy of the operating system. In a tightly-coupled system, the processors share the same operating system code and data structures; that is, they share a common memory and a common copy of the operating system.

Of tightly-coupled multiprocessing systems there are two varieties: symmetric or asymmetric. In a symmetric system all the processors can execute all of the operating system code, though perhaps not concurrently. In an asymmetric system all processors cannot execute all operating system code.

The VAX-11/782 Attached Processor System is a tightly-coupled, asymmetric multiprocessing system. It consists of two VAX-11/780 processors, up to four MA780 multiport memory units, and various optional peripherals. No local memory is used; all active memory is contained in the MA780 and is accessible to both processors. (Actually, some local memory exists in each processor, in order to run micro-diagnostics.) One processor acts as the primary processor; all active peripherals are attached to it. The other processor is known as the attached processor, and it has no peripherals. Figure 17-1 illustrates an example of an 11/782 configuration.

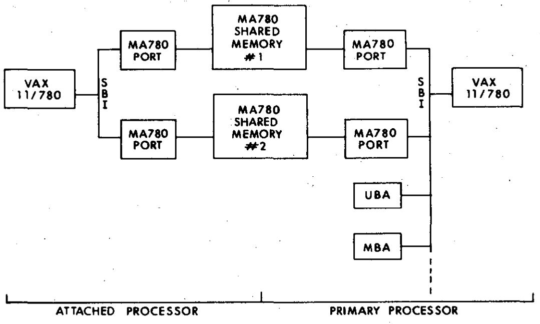


Figure 17-1 Example VAX-11/782 Configuration

SOFTWARE

The VAX-11/782 system is compatible with other members of the VAX family; most applications that run on any of the other VAX systems will run on the 11/782 without modification. (An exeption is an application that allows simultaneous modification of data by multiple processes, without using RMS. See "Programming Considerations" below.) It is the same operating system, VAX/VMS, that operates on a single system, with only minimal modifications made. The additional multiprocessing code added when multiprocessing is initiated, by an automated procedure during startup. Because no complex or pervasive changes were made to the VMS kernel-mode code, the user is assured of enjoying the reliability that has characterized VAX/VMS.

Structure

To repeat, the VAX-11/782 is a tightly-coupled, asymmetric multiprocessing system. There is a single copy of VAX/VMS. Both processors execute this copy simultaneously. All kernel-mode and interrupt code is executed by the primary processor, eliminating the need for complex synchronizing or locking mechanisms of various data structures.

The attached processor acts as a computational work-horse. The primary processor schedules all work on the system and performs all I/O, both for itself and for the attached processor. In essense, then, this is a master-slave arrangement, with the attached processor acting more like a computing peripheral.

Attached Processor States

A state variable is maintained to record the current state of the attached processor. The primary uses this state to determine whether to schedule work for the attached. Each state of the variable is "owned" by only one of the processors. Only the owner has the right to alter the state of the variable when it is in that state. This rule prevents various race conditions. Figure 17-2 shows the state transition diagram for the attached processor. The paths are marked to indicate which processor controls each transition from one state to another.

The attached processor is set to the INIT state when the DCL command, START/CPU, is executed on the primary processor. After the attached finishes executing the multiprocessing initialization code, it will change to the IDLE state. The primary notices this fact during the next reschedule operation and will attempt to reschedule a process for the attached to execute. When a process is found for the attached, the primary sets the state to BUSY. The attached, which has been busy-waiting checking the state variable, then does a load-process-context and sets the state to EXECUTE. The EXECUTE state is a unique state so that special conditions such as powerfail can be handled correctly.

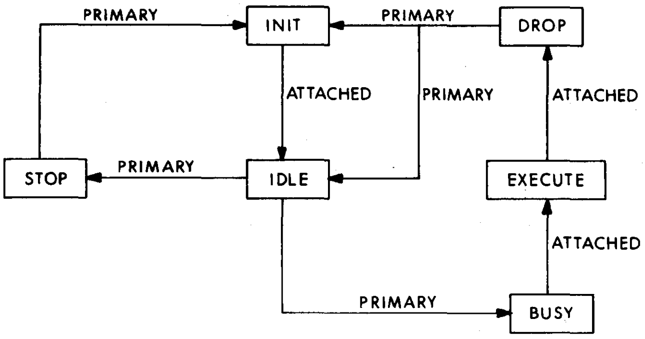


Figure 17-2 Attached Processor States

The attached processor will execute its current process until the process either receives its quantum or requests some action in kernel-mode. At this time, the attached will do a save-process-context and set its state to DROP. The attached then interrupts the primary, requesting the primary to schedule another process for it. Once the primary has taken the process back from the attached, it sets the state to IDLE. Thus the state transition has made an entire circuit.

There is one more state, the STOP state. This is used to request the attached processor to turn itself off. It can be requested by the system manager with the DCL command, STOP/CPU, or by the primary processor, for example, in a bugcheck situation. When the attached is in the INIT or STOP state, the primary knows it should not request any action or schedule any work for the attached. The attached can be restarted by another DCL command, START/CPU, or by rebooting as after bugcheck.

Scheduling

From the perspective of user applications programs, the scheduling for the 11/782 is the same as for a single processor VAX system, with the primary doing all scheduling and the attached processor acting as an additional computing machine. The scheduling algorithm is the same as that always used by VAX/VMS — round-robin, highest priority jobs first — except that any process executing in kernel-mode may not run on the attached processor and the attached processor will not be pre-empted by a higher priority process.

When a process on the attached processor completes its quantum of time or changes mode to kernel, its context is saved and it is given back to the primary. The primary is notified using the MA780 hardware interrupt feature. The primary then schedules another process for the attached. Scheduling for the attached processor is always done before scheduling for the primary.

If there is no process capable of being run on the attached, then the primary will start executing a process. An AST delivery interrupt is used to detect when a process running on the primary leaves kernel mode. The AST interrupt is treated as a rescheduling interrupt, and there is now work for the attached processor.

Exception Handling

Exceptions that cause transition to kernel mode are handled differently for the attached processor than for the primary. There is a separate System Control Block (SCB) for the attached, which provides the ability to execute different code for the same exception, depending on which processor it occurs on. When an exception occurs on the attached, the current process's context is saved and the process is passed back to the primary by requesting a rescheduling event.

Multiprocessing Interrupt Communication

The MA780 hardware provides an important feature that allows effecient asymmetric multiprocessing: the ability of any processor to interrupt any other processor. This is used extensively by the VAX-11/782, in both directions.

The primary processor interrupts the attached processor for the following reasons:

- To request an invalidate of a system space address
- Because an AST has arrived for the process currently executing on the attached
- To request a bugcheck

The attached processor interrupts the primary for other reasons:

- To request a reschedule event
- To log an error
- To request a bugcheck

Fault Handling

Numerous features of VAX/VMS have been extended to work with two processors. They include:

Powerfail	The attached processor may lose power indefinitely and the primary will continue to run without a single job being lost. If the primary powerfails, then the attached processor will wait until the primary restarts. Nothing is lost in any combination of powerfailures.
Bugcheck	Bugcheck may be requested by either processor. The attached processor will go idle while the primary writes the system dump file and reboots.
Machine-check	Machine-checks work similarly to those on a single-processor VMS system. Many machine-checks are recoverable and do not require a reboot.
Error logging	The same errors logged on a VAX-11/780 are logged on the 11/782.
Auto restart	Console floppies are provided for both processors. They allow automatic restart after powerfailures and automatic reboot after bugchecks and machine-checks.

PROGRAMMING CONSIDERATIONS

Although most applications will run without modification on a VAX-11/782, there is an exceptional situation: applications performing their own synchronization of multiple-process access to data structures. In other words, applications that do not employ VAX/VMS higher-level services such as RMS or, higher still, layered products such as VAX-11 DATATRIEVE for sharing data.

In this case, the programmer needs to use the MA780-specific interlocking queue instructions. They are:

ADAWI	Add Aligned Word Interlocked
BBCCI	Branch on Bit Clear and Clear Interlocked
BBSSI	Branch on Bit Set and Set Interlocked
INSQHI	Insert into Queue Head, Interlocked

INSQTI	Insert into Queue Tail, Interlocked
REMQHI	Remove from Queue Head, Interlocked
REMQTI	Remove from Queue Tail, Interlocked

These instructions should be used to acquire access to any mutual exclusion semaphores (mutexes) in a users application.

SYSTEM MANAGEMENT

The tasks required of a system manager are the same for the VAX-11/782 Attached Processor System as for other VAX systems, with the following additional considerations:

- Initialization
- Startup command file
- Special DCL commands

Initialization

A standard VAX/VMS system is booted on the primary processor, using only multiport memory. The DCL command, START/CPU, is executed, which loads the multiprocessing-specific code into non-paged pool. Usually, this command is part of the site-specific command file, SYSTARTUP.COM, set up by the system manager.

At this point a new System Control Block (SCB) is initialized for the attached processor and the primary SCB is modified to handle the multiprocessing scheduling code and MA780 interrupt communication. The attached processor is booted now. It performs a brief initialization process, then interrupts the primary, requesting a process to execute.

Both processors are now up and running, the primary scheduling work for both.

The Startup Command File

As suggested above, the command to load the multiprocessing code that enables the attached processor to be booted is usually included in the SYSTARTUP.COM command file. Typically, the system manager would follow the START/CPU command with command that indicates that the multiprocessor code has been loaded, and the attached processor can be booted. It might go something like this:

```
$ START/CPU             !Load multiprocessing code
$ WRITE SYS$OUTPUT "Ok to boot attached processor now."
```

If the system manager chooses not to boot the attached processor, the primary will operate alone. Once the attached is booted, the primary will automatically begin scheduling work for it.

Special DCL Commands

There are three DCL commands available to the system manager to help monitor and control the VAX-11/782 Attached Processor System. They are:

START/CPU	This command loads the multiprocessing code and is used in initializing multiprocessing.
SHOW/CPU	This command displays the the state of the attached processor, as described above in "Attached Processor States".
STOP/CPU	This command stops the attached processor, disabling multiprocessing.

APPENDIX A
COMMONLY USED MNEMONICS

ACP	Ancillary Control Process
ANSI	American National Standard Institute
ASCII	American Standard Code for Information Interchange
AST	Asynchronous System Trap
ASTLVL	Asynchronous System Trap Level
CCB	Channel Control Block
CDD	Common Data Dictionary
CEB	Common Event Block
CLI	Command Language Interpreter
CM	Compatibility Mode bit in the hardware PSL
CPU	Central Processing Unit
CRB	Channel Request Block
CRC	Cyclic Redundancy Check
CSR	Central Status Register
DAP	Data Access Protocol
DCL	DIGITAL Command Language
DDB	Device Data Block
DDCMP	DIGITAL Data Communication Message Protocol
DDT	Driver Data Table
DST	Debug Symbol Table
DV	Decimal Overflow trap enable bit in the PSW
ECB	Exit Control Block
ECC	Error Correction Code
ESP	Executive Mode Stack Pointer
ESR	Exception Service Routine
FAB	File Access Block
FCA	Fixed Control Area
FCB	File Control Block
FCS	File Control Services
FDL	File Definition Language
FDT	Function Decision Table
FMS	Forms Management System
FNM	File Name
FP	Frame Pointer
FPD	First Part (of an instruction) Done
FU	Floating Underflow trap enable bit in the PSW
GSD	Global Section Descriptor
GST	Global Symbol Table

IDB	Interrupt Dispatch Block
IOSB	I/O Status Block
IPL	Interrupt Priority Level
IRP	I/O Request Packet
IS	Interrupt Stack bit in PSL
ISECT	Image Section
ISD	Image Section Descriptor
ISP	Interrupt Stack Pointer
ISR	Interrupt Service Routine
IV	Integer Overflow trap enable bit in the PSW
JSB	Jump to Subroutine
KED	Keypad Editor
KSP	Kernel Mode Stack Pointer
MBA	MASSBUS Adapter
MBZ	Must be Zero
MCR	Monitor Console Routine
MFD	Master File Directory
MFPR	Move From Process Register instruction
MME	Memory Mapping Enable
MTPR	Move To Process Register instruction
MUTEX	Mutual Exclusion semaphore
NETACP	Network Ancillary Control Process
NCB	Network Connect Block
NSP	Network Services Protocol
OPCOM	Operator Communication Manager
P0BR	Program region base register
P0LR	Program region length register
P1BR	Control region base register
P1LR	Control region limit register
P1PT	Control region page table
PC	Program Counter
PCB	Process Control Block
PCBB	Process Control Block Base register
PFN	Page Frame Number
PID	Process Identification Number
PLAS	Program Logical Address Space
PME	Performance Monitor Enable bit in PCB
PSECT	Program Section
PSL	Processor Status Longword
PSW	Processor Status Word
PTE	Page Table Entry
QIO	Queue Input/Output Request system service
RAB	Record Access Block
REI	Return from Exception or Interrupt

RFA	Record's File Address
RMS	Record Management Services
RWED	Read, Write, Execute, Delete
SBI	Synchronous Backplane Interconnect
SBR	System Base Register
SCB	System Control Block
SCBB	System Control Block Base register
SLR	System Length Register
SP	Stack Pointer
SPT	System Page Table
SSP	Supervisor Mode Stack Pointer
SVA	System Virtual Address
TP	Trace trap Pending bit in PSL
UAF	User Authorization File
UBA	UNIBUS Adapter
UCB	Unit Control Block
UETP	User Environment Test Package
UFD	User File Directory
UIC	User Identification Code
USP	User Mode Stack Pointer
VAX	Virtual Address Extender
VBF	Variable-Length Bit Field
VCB	Volume Control Block
VMS	Virtual Memory Operating System
VPN	Virtual Page Number
WCB	Window Control Block
WCS	Writable Control Store
WDCS	Writable Diagnostic Control Store

GLOSSARY

abort An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, such that the instruction cannot necessarily be restarted.

absolute indexed mode An indexed addressing mode in which the base operand specifier is addressed in absolute mode.

absolute mode In absolute mode addressing, the PC is used as the register in autoincrement deferred mode. The PC contains the address of the location containing the actual operand.

absolute time Time values expressing a specific date (month, day, and year) and time of day. Absolute time values are always expressed in the system as positive numbers.

access mode 1. Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel (mode 0), executive (mode 1), supervisor (mode 2), and user (mode 3). When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. When the processor is in any other mode, the processor is inhibited from executing privileged instructions. The processor status longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is not more protected than normal user programs. 2. See record access mode.

access type 1. The way in which the processor accesses instruction operands. Access types are: read, write, modify, address, and branch. 2. The way in which a procedure accesses its arguments. 3. See record access type.

access violation An attempt to reference an address that is not mapped into virtual memory or an attempt to reference an address that is not accessible by the current access mode.

account name A string that identifies a particular account used to accumulate data on a job's resource use. This name is the user's accounting charge number, not the user's identification code (UIC).

address A number used by the operating system and user software to identify a storage location. See also virtual and physical address.

address access type The specified operand of an instruction is not directly accessed by the instruction. The address of the specified operand is the actual instruction operand. The context of the address calculation is given by the data type of the operand.

addressing mode The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers. The basic general register addressing modes are called: register, register deferred, autoincrement, autoincrement deferred, autodecrement, displacement, and displacement deferred. In addition, there are six indexed addressing modes using two general registers, and literal mode addressing. The PC addressing modes are called: immediate (for register deferred mode using the PC), absolute (for autoincrement deferred mode using the PC), and branch.

address space The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses sent out on the Synchronous Backplane Interconnect (SBI).

allocate a device To reserve a particular device unit for exclusive use. A user process can allocate a device only when that device is not allocated by any other process.

alphanumeric character An upper or lower case letter (A-Z, a-z), a dollar sign ($), an underscore(_), or a decimal digit (0-9).

American Standard Code for Information Interchange (ASCII) A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol.

Ancillary Control Process (ACP) A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management. Three examples of ACPs are: the Files-11 ACP, a magnetic tape ACP, and a networks ACP.

argument pointer General register 12 (R12). By convention, AP contains the address of the base of the argument list for procedures initiated using the CALL instructions.

assign a channel To establish the necessary software linkage between a user process and a device unit before a user process can transfer any data to or from that device.

assembler A program that translates a source language whose operations correspond directly to machine opcodes into an object language.

asynchronous record operation A mode of record processing in which a user program can continue to execute after issuing a record retrieval or storage request without having to wait for the request to be fulfilled.

Asynchronous System Trap A software-simulated interrupt to a user-defined service routine. ASTs enable a user process to be notified asynchronously, with respect to its execution, of the occurrence of a specific event. If a user process has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine exits, the system resumes the process at the point where it was interrupted.

Asynchronous System Trap level (ASTLVL) A value kept in an internal processor register that is the innermost access mode for which an AST is pending. When a Return from Exception or Interrupt (REI) is made to an access mode of privilege equal to or greater than the value in the process register ASTLVL, a software interrupt at Interrupt Priority Level (IPL) 2 is requested. Thus, an AST for an access mode will not be serviced while the processor is executing in a higher priority access mode.

authorization file See user authorization file.

autodecrement indexed mode An indexed addressing mode in which the base operand specifier uses autodecrement mode addressing.

autodecrement mode In autodecrement mode addressing, the contents of the selected register are decremented, and the result is used as the address of the actual operand for the instruction. The contents of the register are decremented according to the data type context of the register: 1 for byte, 2 for word, 4 for longword and floating, 8 for quadword and double floating.

autoincrement deferred indexed mode An indexed addressing mode in which the base operand specifier uses autoincrement deferred mode addressing.

autoincrement deferred mode In autoincrement deferred mode addressing, the specified register contains the address of a longword which contains the address of the actual operand. The contents of the register are incremented by 4 (the number of bytes in a longword). If the PC is used as the register, this mode is called absolute mode.

autoincrement indexed mode An indexed addressing mode in which the base operand specifier uses autoincrement mode addressing.

autoincrement mode In autoincrement mode addressing, the contents of the specified register are used as the address of the operand, then the contents of the register are incremented by the size of the operand.

backing store The collection of locations on secondary storage where data are held.

backing store address The address of a page in the backing store. As page frame numbers are removed from page tables, they are replaced by backing store addresses.

balance set The set of all process working sets currently resident in physical memory. The processes whose working sets are in the balance set have memory requirements that balance with available memory. The balance set is maintained by the system swapper process.

base operand address The address of the base of a table or array referenced by index mode addressing.

base operand specifier The register used to calculate the base operand address of a table or array referenced by index mode addressing.

base priority The process priority that the system assigns a process when it is created; it usually comes from the User Authorization File. A normal process's current priority is modified to reflect its execution history, but the current priority will never drop below the base priority. An image running in a suitably privileged process can, through a system service, alter its own current and base priority.

base register A general register used to contain the address of the first entry in a list, table, array, or other data structure.

binding See linking.

bit complement of a number (also called the one's complement) The result of exchanging 0s and 1s in the binary representation of a number. Thus, the bit complement of the binary number 11011001 (217_{10}) is 00100110. Bit complements are used in place of their corresponding binary numbers in some arithmetic computations in computers.

bit string See variable length bit field.

block 1. The smallest addressable unit of data that the specified device can transfer in an I/O operation. (Under VAX/VMS, a block is a logical entity—disk addresses are expressed as virtual or logical block numbers even when the disk has a smaller addressable unit.) 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

block I/O A data accessing technique in which the program manipulates the blocks (physical records) that make up a file, instead of its logical records.

bootstrap block A block in the index file on a system disk which contains a program that can load the operating system into memory and start its execution.

branch access type An instruction attribute which indicates that the processor does not reference an operand address, but that the operand is a branch displacement. The size of the branch displacement is given by the data type of the operand.

branch mode In branch addressing mode, the instruction operand specifier is a signed byte or word displacement. The displacement is added to the contents of the updated PC (which is the address of the first byte beyond the displacement), and the result is the branch address.

bucket A storage structure consisting of from 1 to 32 blocks, and used for building and processing relative and indexed files. A bucket contains one or more records or record cells.

bucket locking A facility that prevents access to any record in a bucket by more than one user until that user releases the bucket.

buffer A temporary data storage area in a process address space.

buffered I/O See system buffered I/O.

bug check The operating system's internal diagnostic check. The system logs the failure. It may write a crash dump file and it may crash the system.

byte A byte is eight contiguous bits starting on an addressable byte boundary. Bits are numbered from the right, 0 through 7, with bit 0 the low-order bit. When interpreted arithmetically, a byte is a 2's complement integer with significance increasing from bits 0 through 6. Bit 7 is the sign bit. The value of the signed integer is in the range −128 to 127 decimal. When interpreted as an unsigned integer, the value is in the range 0 to 255 decimal. A byte can be used to store one ASCII character.

cache memory A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor, and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

call frame See stack frame.

call instructions The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

call stack The stack, and conventional stack structure, used during a procedure call. Each access mode of each process context has one call stack, and interrupt service context has one call stack.

channel A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can transfer data to or from that device.

channel control block (CCB) A device-oriented data structure providing the link between a process and a device on which it is to perform I/O. One channel control block exists for each assigned channel.

channel request block (CRB) A device-oriented data structure used to arbitrate the use of a common controller for several device units. One channel request block is used for each controller.

character A symbol represented by an ASCII code. See also alphanumeric character.

character string A contiguous set of bytes. A character string is identified by two attributes: an address and a length. Its address is the address of the byte containing the first character of the string. Subsequent characters are stored in bytes of increasing addresses. The length is the number of characters in the string.

character string descriptor A quadword data structure used for passing character data (strings). The first word of the quadword contains the length of the character string. The second word can contain type information. The remaining longword contains the address of the string.

cluster 1. A set of contiguous blocks that is the basic unit of space allocation on a Files-11 disk volume. 2. A set of pages brought into memory in one paging operation. 3. An event flag cluster.

code 1. The vocabulary in which a computer is addressed. Code can be cryptographic (as in ASCII and binary) where digits and other characters represent information, or language mimetic (as in MACRO and FORTRAN) where English-like phrases represent information. 2. To code: To organize and write instructions and data for a computer in vocabulary it understands.

command An instruction, generally an English word, typed by the user at a terminal or included in a command file, which requests the software monitoring a terminal or reading a command file to perform some well-defined activity. For example, typing the COPY command requests the system to copy the contents of one file into another file.

command file A file containing command strings. See also command procedure.

command interpreter Procedure-based system code that executes in supervisor mode in the context of a process to receive, check the syntax, and parse commands typed by the user at a terminal or submitted in a command file.

command parameter The positional operand of a command delimited by spaces, such as a file specificiation, option, or constant.

command procedure A file containing commands and data that the command interpreter can accept in lieu of the user's typing the commands individually on a terminal.

command string A line (or set of continued lines), normally terminated by typing the carriage return key, containing a command and, optionally, information modifying the command. A complete command string consists of a command, its qualifiers, if any, and its parameters (file specifications, for example), if any, and their qualifiers, if any.

common 1. A FORTRAN term for a program section that contains only data. 2. Shared (used or held "in common"), e.g., common event flag cluster.

common event block The data structure created when a common event flag cluster is created. It provides the control and coordination mechanism for the structure. One is associated with each cluster of common event flags.

common event flag cluster A set of 32 event flags that enables cooperating processes to post event notification to each other. Common event flag clusters are created as they are needed. A process can associate with up to two common event flag clusters.

compatibility mode A mode of execution that enables the central processor to execute non-privileged PDP-11 instructions. The operating system supports compatibility mode execution by providing an RSX-11M execution environment for an RSX-11M task image. The operating system compatibility mode procedures reside in the program region of the process executing a compatibility mode image. The procedures intercept calls to the RSX-11M executive and convert them to the appropriate operating system functions.

compiler A code that translates a program written in a high-level language (such as COBOL, PASCAL, or FORTRAN) into an object program.

condition An exception condition detected and declared by software. For example, see failure exception mode.

condition codes Four bits in the processor status word that indicate the results of the previously executed instruction.

condition handler A procedure that a process wants the system to execute when an exception condition occurs. When an exception condition occurs, the operating system searches for a condition handler and, if it is found, initiates the handler immediately. The condition handler may perform some act to change the situation that caused the exception condition and continue execution for the process that incurred the exception condition. Condition handlers execute in the context of the process at the access mode of the code that incurred the exception condition.

condition value A 32-bit quantity that uniquely identifies an exception condition.

context The environment of an activity. See also process context, hardware, and software context.

context indexing The ability to index through a data structure automatically because the size of the data type is known and used to determine the offset factor.

context switching Interrupting the activity in progress and switching to another activity. Context switching occurs as one process after another is scheduled for execution. The operating system saves the interrupted process's hardware context in its hardware PCB using the Save Process Context instruction, loads another process's hardware PCB into the hardware context using the Load Process Context instruction, and schedules that process for execution.

contiguous Physically adjacent and/or consecutively numbered units of data.

contiguous area A space allocation on disk where the reserved areas for all blocks in a file are physically adjacent on the recording medium.

continuation character A hyphen at the end of a command line signifying that the command string continues on to the next command line.

control region The highest-addressed half of per-process space (the P1 region). Control region virtual addresses refer to the process-related information used by the system to control the process, such as: the kernel, executive, and supervisor stacks, the permanent I/O channels, exception vectors, and dynamically used system procedures (such as the command interpreter procedures). The user stack is also normally found in the control region.

Control Region Base Register (P1BR) The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of a process control region page table.

Control Region Length Register (P1LR) The processor register, or its equivalent in a hardware process control block, that contains the number of non-existent page table entries for virtual pages in a process control region.

copy-on-reference A method used in memory management for sharing data until a process accesses it, in which case it is copied before the access. Copy-on-reference allows sharing of the initial values of a global section whose pages have read/write access but contain pre-initialized data available to many processes.

counted string A character string data structure consisting of a byte-sized length followed by the string. Although a counted string is not used as a procedure argument, it is a convenient representation in memory.

current access mode The processor access mode of the currently executing software. The current mode field of the processor status longword indicates the access mode of the currently executing software.

cylinder The tracks at the same radius on all recording surfaces of a disk.

data base A collection of related data structures.

data structure Any table, list, array, queue, or tree whose format and access conventions are well defined for reference by one or more images.

data type In general, the way in which bits are grouped and interpreted. In reference to the processor instructions, the data type of an operand identifies the size of the operand and the significance of the bits in the operand. Operand data types include: byte, word, longword, and quadword integer, floating and double floating, character string, packed decimal string, and variable-length bit field.

default The omission of certain information. The system assumes pre-arranged values for the defaulted (omitted) information.

deferred echo Refers to the fact that terminal echoing does not occur until a process is ready to accept input entered by type ahead.

delta time A time value expressing an offset from the current date and time. Delta times are always expressed in the system as negative numbers whose absolute value is used as an offset from the current time.

demand paging One technique that enables a program to execute without having all of its pages resident in physical memory. In demand paging, a program page is not brought into physical memory until it is actually needed. For the technique used by VAX/VMS, see paging.

demand zero page A page, typically of an image stack or buffer area, that is initialized to contain all zeros when dynamically created in memory as a result of a page fault. This feature eliminates the waste of disk space that would otherwise be required to store blocks (pages) that contain only zeros.

descriptor A data structure used in calling sequences for passing argument types, addresses and other optional information. See character string descriptor.

detached process A process that has no owner. A parent process of a tree of subprocesses. Detached processes are created by the job controller when a user logs on the system or when a batch job is initiated. The job controller does not own the user processes it creates; these processes are therefore detached.

device The general name for any physical terminus or link connected to the processor that is capable of receiving, storing, or transmitting data. Card readers, line printers, and terminals are examples of record-oriented devices. Magnetic tape devices and disk devices are examples of mass storage devices. Terminal line interfaces and interprocessor links are examples of communications devices.

device driver See driver.

device interrupt An interrupt received on interrupt priority level 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device name The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable), and ends with a colon (:).

device queue See spool queue.

device register A location in device controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit One drive, and its controlling logic, of a mass storage device system. A mass storage system can have several drives connected to it.

diagnostic A program that tests hardware/firmware logic and peripherals and reports any faults it detects.

direct I/O See system buffered I/O.

direct mapping cache A cache organization in which only one address comparison is needed to locate any data in the cache because any block of main memory data can be placed in only one possible position in the cache. Contrast with fully associative cache.

directory A file, used to locate files on a volume, that contains a list of file names (including extension and version number) and their unique internal identifications.

directory name The field, in a file specification, that identifies the directory file in which a file is listed.

displacement deferred indexed mode An indexed addressing mode in which the base operand specifier uses displacement deferred mode addressing.

displacement deferred mode In displacement deferred mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of a longword which contains the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

displacement indexed mode An indexed addressing mode in which the base operand specifier uses displacement mode addressing.

displacement mode In displacement mode addressing, the specifier extension is a byte, word, or longword displacement. The displacement is sign extended to 32 bits and added to a base address obtained from the specified register. The result is the address of the actual operand. If the PC is used as the register, the updated contents of the PC are used as the base address. The base address is the address of the first byte beyond the specifier extension.

domain A DATATRIEVE term that describes an entire collection of records of a single type plus a name (or record definition). The size of the domain changes as appropriate records are inserted or removed. The record definition for each domain is stored in the Data Dictionary.

double floating datum Eight contiguous bytes (64 bits), starting on an addressable byte boundary, which are interpreted as containing a floating point number. The bits are labeled from right to left, 0 to 63. An eight-byte floating point number is identified by the address of the byte containing bit 0. Bit 15 contains the sign of the number. Bits 14 through 7 contain the excess 128 binary exponent. Bits 63 through 16 and 6 through 0 contain a normalized 56-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from 6 through 0, 31 through 16, 47 through 32, then 63 through 48. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of −128 to 127. An exponent value of 0 together with a sign bit of 0 represent a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range (±) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{55} or 16 decimal digits.

drive The electro-mechanical unit of a mass storage device system on which a recording medium (disk cartridge, disk pack, or magnetic tape reel) is mounted.

driver The set of code that handles physical I/O to a device.

dynamic access A technique in which a program switches from one access mode to another while processing a file.

echo A terminal handling characteristic in which the characters typed by the terminal user on the keyboard are also displayed on the screen or printer.

effective address The address obtained after indirect or indexing modifications are calculated.

entry mask A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the call and return instructions.

entry point A location that can be specified as the object of a call. It contains an entry mask and exception enables known as the entry point mask.

equivalence name The string associated with a logical name in a logical name table. An equivalence name can be, for example, a device name, another logical name, or a logical name concatenated with a portion of a file specification.

error logger A system process that empties the error log buffers and writes the error messages into the error file. Errors logged by the system include memory system errors, device errors and timeouts, and interrupts with invalid vector addresses.

escape sequence An escape is a transition from the normal mode of operation to a mode outside the normal mode. An escape character is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle escape sequences.

event A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process's ability to execute. Events can be synchronous with the process's execution (a wait request), or they can be asynchronous (I/O completion). Some other events include swapping, wake request, and page fault.

event flag A bit in an event flag cluster that can be set or cleared to indicate the occurrence of the event associated with that flag. Event flags are used to synchronize activities in a process or among many processes.

event flag cluster A set of 32 event flags that are used for event posting. Four clusters are defined for each process: two process-local clusters, and two common event flag clusters. Of the process-local flags, eight are reserved for system use.

exception An event detected by the hardware (other than an interrupt or jump, branch, case, or call instruction) that changes the normal flow of instruction execution. An exception is always caused by the execution of an instruction or set of instructions (whereas an interrupt is caused by an activity in the system independent of the current instruction). There are three types of hardware exceptions: traps, faults, and aborts. Examples are: attempts to execute a privileged or reserved instruction, trace traps, compatibility mode faults, breakpoint instruction execution, and arithmetic traps such as overflow, underflow, and divide by zero.

exception condition A hardware- or software-detected event other than an interrupt or jump, branch, case, call, jump to subroutine, or branch to subroutine instruction that changes the normal flow of instruction execution.

exception dispatcher An operating system procedure that searches for a condition handler when an exception condition occurs. If no exception handler is found for the exception or condition, the image that incurred the exception is terminated.

exception enables See trap enables.

exception vector See vector.

executable image Images that are capable of being run in a process. When run, an executable image is read from a file for execution in a process.

executive The generic name for the collection of procedures included in the operating system software that provide the basic control and monitor functions of the operating system.

executive mode The second most privileged processor access mode (mode 1). The record management services (RMS) and many of the operating system's system service procedures execute in executive mode.

exit An image exit is a rundown activity that occurs when image execution terminates either normally or abnormally. Image rundown activities include deassigning I/O channels and disassociation of common event flag clusters. Any user- or system-specified exit handlers are called.

exit handler A procedure executed when an image exits. The declaration of an exit handler enables a procedure that is not on the call stack to gain control and clean up procedure-own data bases before the actual image exit occurs.

extended attribute block (XAB) An RMS user data structure that contains additional file attributes beyond those expressed in the file access block (FAB), such as boundary types (aligned on cylinder, logical block number, virtual block number) and file protection information.

extent The contiguous area on a disk containing a file or a portion of a file. Consists of one or more clusters.

failure exception mode A mode of execution selected by a process indicating that it wants an exception condition declared if an error occurs as the result of a system service call. The normal mode is for the system service to return an error status code for which the process must test.

fault A hardware exception condition that occurs in the middle of an instruction and that leaves the registers in memory in a consistent state, such that elimination of the fault and restarting the instruction will give correct results.

field 1. See variable-length bit field. 2. A set of contiguous bytes in a logical record.

file A logically related collection of data treated as a physical entity that occupies one or more blocks on a volume such as disk or magnetic tape. A file can be referenced by a name assigned by the user. A file normally consists of one or more logical records.

file access block (FAB) An RMS user data structure that represents a request for data operations related to the file as a whole, such as OPEN, CLOSE, or CREATE.

file header A block in the index file describing a file on a Files-11 disk structure. The file header identifies the locations of the file's extents. There is a file header for every file on the disk.

file name extension See file type.

file organization The particular file structure used to record the data constituting a file on a mass storage medium. RMS file organizations are: sequential, relative, and indexed.

Files-11 The name of the on-disk structure used by the RSX-11, IAS and VAX/VMS operating systems.

file specification A unique name for a file on a mass storage medium. It identifies the node, the device, the directory name, the file name, and the version number under which a file is stored.

file structure The way in which the blocks forming a file are distributed on a disk or magnetic tape to provide a physical accessing technique suitable for the way the data in the file are processed.

file system A method of recording, cataloging, and accessing files on a volume.

file type The field in a file specification that consists of a period (.) followed by a 0- to 3-character type identification. By convention, the type identifies a generic class of files that have the same use or characteristics, such as ASCII text files, binary object files, etc.

fixed control area An area associated with a variable length record available for controlling or assisting record access operations. Typical uses include line numbers and printer format control information.

fixed length record format A file format in which all records have the same length.

floating (point) datum Four contiguous bytes (32 bits) starting on an addressable byte boundary. The bits are labeled from right to left from 0 to 31. A four-byte floating point number is identified by the address of the byte containing bit 0. Bit 15 contains the sign of the number. Bits

14 through 7 contain the excess 128 binary exponent. Bits 31 through 16 and 6 through 0 contain a normalized 24-bit fraction with the redundant most significant fraction bit not represented. Within the fraction, bits of decreasing significance go from bit 6 through 0, then 31 through 16. Exponent values of 1 through 255 in the 8-bit exponent field represent true binary exponents of −128 to 127. An exponent value of 0 together with a sign bit of 0 represent a floating value of 0. An exponent value of 0 with a sign bit of 1 is a reserved representation; floating point instructions processing this value return a reserved operand fault. The value of a floating datum is in the approximate range ($\pm$) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{32} or seven decimal digits.

foreign volume Any volume other than a Files-11 formatted volume which may or may not be file structured.

fork dispatcher A software interrupt service routine that selects a fork process for execution. The fork dispatcher selects a fork process from a queue of fork blocks. There is one queue for all the fork processes that share a given interrupt priority level.

fork process A dynamically created system process such as a process that executes device driver code. Fork processes have minimal context and reside entirely in system space. They execute at software interrupt levels and are dispatched for execution immediately. Fork processes remain resident until they terminate.

frame pointer General register 13 (R13). By convention, FP contains the base address of the most recent call frame on the stack.

full process A synonym for "process."

fully associative cache A cache organization in which any block of data from main memory can be placed anywhere in the cache. Address comparison must take place against each block in the cache to find any particular block. Contrast with direct mapping cache.

general register Any of the sixteen 32-bit registers used as the primary operands of the native mode instruction. The general registers include 12 general purpose registers which can be used as accumulators, as counters, and as pointers to locations in main memory, and the frame pointer (FP), argument pointer (AP), stack pointer (SP), and program counter (PC) registers.

generic device name A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted.

giga Metric term used to represent the number 1 followed by nine zeros.

global page table The page table containing the master page table entries for global sections.

global section A data structure (e.g., FORTRAN global common) or shareable image section potentially available to all processes in the system. Access is protected by privilege and/or group number of the UIC.

global symbol A symbol defined in a module that is potentially available for reference by another module. The linker resolves (matches references with definitions) global symbols. Contrast with local symbol.

global symbol table (GST) In a library, an index of strongly defined global symbols used to access the modules defining the global symbols. The linker will also put global symbol tables into an image. For example, the linker appends a global symbol table to executable images that are intended to run under the symbolic debugger, and it appends a global symbol table to all shareable images.

group 1. A set of users who have special access privileges to each other's directories and files within those directories (unless protected otherwise), as in the context "system, owner, group, world," where group refers to all members of a particular owner's group. 2. A set of jobs (processes and their subprocesses) that have access privileges to a group's common event flags and logical name tables, and may have mutual process controlling privileges, such as scheduling, hibernation, etc.

group number The first number in a User Identification Code (UIC).

hardware context The values contained in the following registers while a process is executing: the program counter (PC); the processor status longword (PSL); the fourteen general registers (R0 through R13); the four processor registers (P0BR, P0LR, P1BR, and P1LR) that describe the process virtual address space; the stack pointer (SP) for the current access mode in which the processor is executing; plus the contents to be loaded in the stack pointer for every access mode other than the current access mode. While a process is executing, its hardware context is continually being updated by the processor. While a process is not executing, its hardware context is stored in its hardware PCB.

hardware process control block (PCB) A data structure used by the processor to load and save process context. A process's hardware PCB resides in its process header.

hibernation A state in which a process is inactive, but known to the system with all of its current status. A hibernating process becomes active again when a wake request is issued. It can schedule a wake request before hibernating, or another process can issue its wake request. A hibernating process also becomes active for the time sufficient to service any AST it may receive while it is hibernating. Contrast with suspension.

home block A block in the index file that contains the volume identification, such as volume label and protection.

image An image consists of procedures and data that have been bound together by the linker. There are three types of images: executable, shareable, and system.

image activator A set of system procedures that prepares an image for execution. The image activator establishes the memory management data structures required both to map the image's virtual pages to physical pages and to perform paging.

image exit See exit.

image I/O segment That portion of the control region that contains the RMS internal file access blocks (IFAB) and I/O buffers for the image currently being executed by a process.

image name The file name of the file in which an image is stored.

image section (isect) A group of program sections (psects) with the same attributes (such as read-only access, read/write access, absolute, relocatable, etc.) that is the unit of virtual memory allocation for an image.

image section descriptor (ISD) A software data structure created by the linker when it produces an image section of a shareable or executable image. The image section descriptor contains information that allows the system to locate, characterize, and control the image section. The ISD is located in the image header.

indexed file organization A file organization in which a file contains records and a primary key index (and optionally one or more alternate key indices) used to process the records sequentially by index or randomly by index.

index file The file on a Files-11 volume that contains the access information for all files on the volume and enables the operating system to identify and access the volume.

index file bit map A table in the index file of a Files-11 volume that indicates which file headers are in use.

index register A register used to contain an address offset.

indexed addressing mode In indexed mode addressing, two registers are used to determine the actual instruction operand: an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array. The base operand specifier supplies the base address of the array (the base operand address or BOA). The address of the actual operand is calculated by multiplying the contents of the index register by the size (in bytes) of the actual operand and adding the result to the base operand address. The addressing modes resulting from index mode addressing are formed by adding the suffix "indexed" to the addressing mode of the base operand specifier: register deferred indexed, autoincrement indexed, autoincrement deferred indexed (or absolute indexed), autodecrement indexed, displacement indexed, and displacement deferred indexed.

indirect command file See command procedure.

instruction buffer An eight-byte buffer in the processor used to contain bytes of the instruction currently being decoded and to pre-fetch instructions in the instruction stream. The control logic continuously fetches data from memory to keep the eight-byte buffer full.

interleaving Assigning consecutive physical memory addresses alternately between two memory controllers.

interprocess communication facility A common event flag, mailbox, global section or shared file used to pass information between two or more processes.

interrecord gap A blank space deliberately placed between data records on the recording surface of a magnetic tape.

interrupt An event other than an exception or branch, jump, jump to subroutine, branch to subroutine, case, or call instruction that changes the normal flow of instruction execution. Interrupts are generally external to the process executing when the interrupt occurs. See also device interrupt, software interrupt, and urgent interrupt.

interrupt priority level The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels. IPL 1 is lowest and IPL 31 is highest. The levels arbitrate contention for processor service. For example, a device cannot interrupt the processor if the processor is currently executing at an interrupt priority level greater than or equal to the interrupt priority level of the device's interrupt service routine.

interrupt service routine The routine executed when a software interrupt occurs.

interrupt stack The system-wide stack used when executing in interrupt service context. At any time, the processor is either in a process context executing in user, supervisor, executive, or kernel mode, or in system-wide interrupt service context operating in kernel access mode, as indicated by the interrupt stack and current mode bits in the PSL. The interrupt stack is not context switched.

interrupt stack pointer The stack pointer for the system-wide interrupt stack.

interrupt vector See vector.

I/O Data Base A group of data control blocks that are an important part of the communications link between the operating sytem and the devices handling I/O processing. The blocks are: the Device Data Block, the Unit Control Block, the Channel Request Block, the Interrupt Dispatch Block, and the Adapter Control Block.

I/O driver See driver.

I/O function code A six-bit value specified in a Queue I/O Request system service that describes the particular I/O operation to be performed (e.g., read, write, rewind, open file).

I/O function modifier A 10-bit value specified in a Queue I/O Request system service that modifies an I/O function code (e.g., read terminal input no echo).

I/O lockdown The state of a page such that it cannot be paged or swapped out of memory until any I/O in progress to that page is completed.

I/O rundown An operating system function in which the system cleans up any I/O in progress when an image exits.

I/O space The region of physical address space that contains the configuration registers and device control/status and data registers.

I/O status block A data structure associated with the Queue I/O Request system service. This service optionally returns a status code, number of bytes transferred, and device- and function-dependent information in an I/O status block. It is not returned from the service call, but filled in when the I/O request completes.

job 1. A job is the accounting unit equivalent to a process and the collection of all the subprocesses, if any, that it and its subprocesses create. Jobs are classified as batch and interactive. For example, the job controller creates an interactive job to handle a user's requests when the user logs onto the system and it creates a batch job when the symbiont manager passes a command input file to it. 2. A print job.

job controller The system process that establishes a job's process context, starts a process running the LOGIN image for the job, maintains the accounting record for the job, manages symbionts, and terminates a process and its subprocesses.

job queue A list of files that a process has supplied for processing by a specific device, for example, a line printer.

kernel mode The most privileged processor access mode (mode 0). The operating system's most privileged services, such as I/O drivers and the pager, run in kernel mode.

lexical function A command language construct that the command interpreter evaluates and substitutes before it performs expression analysis on a command string. Lexical functions return information about the current process, such as UIC or default directory; and about character strings, such as length or substring locations.

librarian A program that allows the user to create, update, modify, list, and maintain object library and assembler macro library files.

library file A direct access file containing one or more modules of the same module type.

limit The size or number of given items requiring system resources (such as mailboxes, locked pages, I/O requests, open files, etc.) that a job is allowed to have at any one time during execution, as specified by the system manager in the user authorization file. See also quota.

line number A number used to identify a line of text in a file processed by a text editor.

linker A program that reads one or more object files created by language processors and produces an executable image file, a shareable image file or a system image file.

linking The resolution of external references between object modules used to create an image, the acquisition of referenced library routines, service entry points, and data for the image, and the assignment of virtual addresses to components of an image.

literal mode In literal mode addressing, the instruction operand is a constant whose value is expressed in a six-bit field of the instruction. If the operand data type is byte, word, longword, or quadword, the operand is zero-extended and can express values in the range 0 through 63_{10}. If the operand data type is floating or double floating, the six-bit field is composed of two three-bit fields, one for the exponent and the other for the fraction. The operand is extended to floating or double floating format.

local symbol A symbol that is meaningful only to the module that defines it. Symbols not identified to a language processor as global symbols are considered to be local symbols. A language processor resolves (matches references with definitions) local symbols. They are not known to the linker and cannot be made available to another object module. They can, however, be passed through the linker to the symbolic debugger. Contrast with global symbol.

locality See program locality.

locate mode A record access technique in which a program reads records in an RMS block buffer working storage area to reduce overhead. See also move mode.

locking a page in memory Making a page within a process ineligible for either paging or swapping. A page stays locked in memory until it is specifically unlocked.

locking a page in the working set Making a page within a process ineligible for paging out of the working set for the process. The page can be swapped when the process is swapped. A page stays locked in a working set until it is specifically unlocked.

logic The circuitry for accomplishing a particular operation within the computer firmware or hardware.

logical block number A block on a mass storage device identified using a volume-relative address rather than its physical (device-oriented) address or its virtual (file-relative) address. The blocks that constitute the volume are labeled sequentially starting with logical block 0.

logical I/O function A set of I/O operations (e.g., read and write logical block) that allows restricted direct access to device level I/O operations using logical block addresses.

logical name A user-specified name for any portion or all of a file specification. For example, the logical name INPUT can be assigned to a terminal device from which a program reads data entered by a user. Logical name assignments are maintained in logical name tables for each process, each group, and the system. A logical name can be created and assigned a value permanently or dynamically.

logical name table A table that contains a set of logical names and their equivalence names for a particular process, a particular group, or the system.

logical record A group of related fields treated as a unit.

longword Four contiguous bytes (32 bits) starting on any byte boundary. Bits are numbered from right to left with 0 through 31. The address of the longword is the address of the byte containing bit 0.

When interpreted arithmetically, a longword is a 2's complement integer with significance increasing from bit 0 to bit 30. When interpreted as a signed integer, bit 31 is the sign bit. The value of the signed integer is in the range −2,147,483,648 to 2,147,483,647. When interpreted as an unsigned integer, significance increases from bit 0 to bit 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

macro A statement that requests a language processor to generate a predefined set of instructions.

mailbox A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders perform a write to a mailbox; the receiver performs a read from that mailbox. Some system-wide mailboxes are defined: the error logger and OPCOM read from system-wide mailboxes.

main memory See physical memory.

mapping window A subset of the retrieval information for a file that is used to translate virtual block numbers to logical block numbers.

mass storage device A device capable of reading and writing data on mass storage media such as disk packs or a magnetic tape reels.

member number The second number in a user identification code that uniquely identifies that code.

memory management The system functions that include the hardware's page mapping and protection and the operating system's image activator and pager.

Memory Mapping Enable (MME) A bit in a processor register that governs address translation.

modify access type The specified operand of an instruction or procedure is read, and is potentially modified and written, during that instruction's or procedure's execution.

module 1. A portion of a program or program library, as in a *source module*, *object module*, or *image module*. 2. A board, usually made of plastic covered with an electrical conductor, on which logic devices (such as transistors, resistors, and memory chips) are mounted, and circuits connecting these devices are etched, as in a *logic module*.

monitor See executive.

Monitor Console Routine (MCR) The command interpreter in an RSX-11 system. Also, the command language interpreter when running the Application Migration Executive.

mount a volume 1. To logically associate a volume with the physical unit on which it is loaded (an activity accomplished by system software at the request of an operator). 2. To load or place a magnetic tape or disk pack on a drive and place the drive on-line (an activity accomplished by a system operator).

move mode A record I/O access technique in which a program accesses records in its own working storage area. See also locate mode.

mutex A semaphore that is used to control exclusive access to a region of code that can share a data structure or other resource. The mutex (mutual exclusion) semaphore ensures that only one process at a time has access to the region of code.

name block (NAM) An RMS user data structure that contains supplementary information used in parsing file specifications.

native image An image whose instructions are executed in native mode.

native mode The processor's primary execution mode, in which the programmed instructions are interpreted as byte-aligned, variable length instructions that operate on byte, word, longword, quadword, integer, floating and double-floating, character string, packed decimal, and variable length bit field data. The instruction execution mode other than compatibility mode.

network A collection of interconnected individual computer systems.

nibble The low-order or high-order four bits of a byte.

node An individual computer system in a network.

numeric string A contiguous sequence of bytes representing up to 31 decimal digits (one per byte) and possibly a sign. The numeric string is specified by its lowest addressed location, its length, and its sign representation.

null process A small system process that is the lowest priority process in the system and takes one entire priority class. The sole function of the null process is to accumulate idle processor time.

object module The binary output of a language processor such as the assembler or a compiler, which is used as input to the linker.

object time system (OTS) See Run Time Procedure Library.

offset A fixed displacement from the beginning of a data structure. System offsets for items within a data structure normally have an associated symbolic name used instead of the numeric displacement.

Where symbols are defined, programmers always reference the symbolic names for items in a data structure instead of using the numeric displacement.

opcode The pattern of bits within an instruction that specify the operation to be performed.

operand specifier The pattern of bits in an instruction that indicate the addressing mode, a register and/or displacement, which, taken together, identify an instruction operand.

operand specifier type The access type and data type of an instruction's operand(s). For example, the test instructions are of read access type, since they only read the value of the operand. The operand can be of byte, word, or longword data type, depending on whether the opcode is for the TSTB (test byte), TSTW (test word), or TSTL (test longword) instruction.

Operator Communication Manager (OPCOM) A system process that is always active. OPCOM receives input from a process that wants to inform an operator of a particular status or condition, passes a message to the operator, and tracks the message.

owner In the context "system, owner, group, world," an owner is the particular member (of a group) to which a file, global section, mailbox, or event flag cluster belongs.

owner process The process (with the exception of the job controller) or subprocess that created a subprocess.

packed decimal A method of representing a decimal number by storing a pair of decimal digits in one byte, taking advantage of the fact that only one nibble is required to represent the numbers zero through nine.

packed decimal string A contiguous sequence of up to 16 bytes interpreted as a string of nibbles. Each nibble represents a digit except the low-order nibble of the highest addressed byte, which represents the sign. The packed decimal string is specified by its lowest addressed location and the number of digits.

page 1. A set of 512 contiguous byte locations that begins at an even 512-byte boundary and is used as the unit of memory mapping and protection. 2. The data between the beginning of a file and a page marker, between two markers, or between a marker and the end of a file.

page fault An exception generated by a reference to a page which is not mapped into a working set.

page fault cluster size The number of pages read in on a page fault.

page frame number (PFN) The address of the first byte of a page in physical memory. The high-order 21 bits of the physical address of the base of a page.

page marker A character or characters (generally a form feed) that separates pages in a file that is processed by a text editor.

pager A set of kernel mode procedures that executes as the result of a page fault. The pager makes the page for which the fault occurred available in physical memory so that the image can continue execution. The pager and the image activator provide the operating system's memory management functions.

page table entry (PTE) The data structure that identifies the location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number needed to map the virtual page to a physical page. When it is not in memory, the page table entry contains the information needed to locate the page on secondary storage (disk).

paging The action of bringing pages of an executing process into physical memory when referenced. When a process executes, all of its pages are said to reside in virtual memory. Only the actively used pages, however, need to reside in physical memory. The remaining pages can reside on disk until they need to reside in physical memory. In this system, a process is paged either when it references more pages than it is allowed to have in its working set or when it first starts up an image in memory. When a process refers to a page not in its working set, a page fault occurs. This causes the operating system's pager to read in the referenced page if it is on disk (and, optionally, other related pages depending on a cluster factor), replacing the least recently faulted pages as needed. This system pages a process only against itself.

parameter See command parameter.

per-process address space See process address space.

physical address The address used by hardware to identify a location in physical memory or on directly addressable secondary storage devices such as disk. A physical memory address consists of a page frame number and the number of a byte within the page. A physical disk block address consists of a cylinder or track and sector number.

physical address space The set of all possible 30-bit physical addresses that can be used to refer to locations in memory (memory space) or device registers (I/O space).

physical block number The number of a block on a mass storage device referred to by its physical (device-oriented) address rather than a logical (volume-relative) or virtual (file-relative) address.

physical I/O functions A set of I/O functions that allows access to all device level I/O operations except maintenance mode.

physical memory The memory modules connected to the SBI that are used to store: 1) instructions that the processor can directly fetch and execute, and 2) any other data that a processor is instructed to manipulate. Also called main memory.

pointer A datum that gives the address of ("points to") another datum, data structure, or process.

position-dependent code Code that can execute properly only in the locations in virtual address space that are assigned to it by the linker.

position-independent code Code that can execute properly without modification wherever it is located in virtual address space, even if its location is changed after it has been linked. Generally, this code uses addressing modes that form an effective address relative to the PC.

primary vector A software vector that contains the starting address of a condition handler to be executed when an exception condition occurs. If a primary vector is declared, that condition handler is the first handler to be executed.

priority The rank assigned to an activity that determines its level of service. For example, when several jobs contend for system resources, the job with the highest priority receives service first. See also software priority and interrupt priority level.

private section An image section of a process that is not shareable among processes. See also global section.

privilege See process privilege, user privilege.

privileged instructions In general, any instructions intended for use by the operating system or privileged system programs. In particular, instructions that the processor will not execute unless the current access mode is kernel mode (e.g., HALT, SVPCTX, LDPCTX, MTPR, and MFPR).

procedure 1. A routine entered via a CALL instruction. 2. See command procedure.

process The basic entity scheduled by the system software that provides the context in which an image executes. A process consists of an address space and both hardware and software context.

process address space See process space.

process context The hardware and software context of a process.

process control block (PCB) A data structure used to contain process context. The hardware PCB contains the hardware context. The software PCB contains the software context, which includes a pointer to the hardware PCB.

process header A data structure that contains the hardware PCB, accounting and quota information, process section table, working set list, and the page tables defining the virtual layout of the process.

process header slots That portion of the system address space in which the system stores the process headers for the processes in the balance set. The number of process header slots in the system determines the number of processes that can be in the balance set at any one time.

process identification (PID) The operating system's unique 32-bit binary value assigned to a process. Each process has a process identification and a process name.

process I/O segment That portion of a process control region that contains the process permanent RMS internal file access block for each open file, and the I/O buffers, including the command interpreter's command buffer and command descriptors.

process name A 1- to 15-character ASCII string that can be used to identify processes executing under the same group number.

process page tables The page tables used to describe process virtual memory.

process priority The priority assigned to a process for scheduling purposes. The operating system recognizes 32 levels of process priority, where 0 is low and 31 high. Levels 16 through 31 are used for time-critical processes. The system does not modify the priority of a time-critical process (although the system manager or process itself may). Levels 0 through 15 are used for normal processes. The system may temporarily increase the priority of a normal process based on the activity of the process.

process privileges The privileges granted to a process by the system, which are a combination of user privileges and image privileges. They include, for example, the privilege to: affect other processes associated with the same group as the user's group, affect any process in the system regardless of UIC, set process swap mode, create permanent event flag clusters, create another process, create a mailbox, or perform direct I/O to a file-structured device.

process section See private section.

process section table A data structure used by the image activator and page to interpret the image files produced by the linker.

process space (Also sometimes called per-process space.) The lower addressed half of Virtual Address Space. Process space is itself divided into the Program Region (lower half) and the Control Region (upper half).

processor register A part of the processor used by the operating system software to control the execution states of the computer system. They include the system base and length registers, the program and control region base and length registers, the system control block base register, the software interrupt request register, and many more.

processor status longword (PSL) A system programmed processor register consisting of a word of privileged processor status and the PSW. The privileged processor status information includes: the current IPL (interrupt priority level), the previous access mode, the current access mode, the interrupt stack bit, the trace trap pending bit, and the compatibility mode bit.

Processor Status Word (PSW) The low-order word of the processor status longword. Processor status information includes: the condition codes (carry, overflow, zero, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

Program Counter (PC) General (hardware) register number 15 (R15). At the beginning of execution of an instruction, the PC normally contains the address of the location in memory from which the processor will fetch the next instruction.

program locality A characteristic of a program that indicates how close or far apart the references to locations in virtual memory are over time. A program with a high degree of locality does not refer to many widely scattered virtual addresses in a short period of time.

program region The lowest-addressed half of process address space (P0 space). The program region contains the image currently being executed by the process and other user code called by the image.

program region base register (P0BR) The processor register, or its equivalent in a hardware process control block, that contains the base virtual address of the page table entry for virtual page number 0 in a process program region.

program region length register (P0LR) The processor register, or its equivalent in a hardware process control block, that contains the number of entries in the page table for a process program region.

program section (psect) A portion of a program with a given protection and set of storage management attributes. Program sections that have the same attributes are gathered together by the linker to form an image section.

pure code See re-entrant code.

quadword Eight contiguous bytes (64 bits) starting on any byte boundary. Bits are numbered from right to left, 0 to 63. A quadword is identified by the address of the byte containing the low-order bit (bit 0). When interpreted arithmetically, a quadword is a 2's complement integer with significance increasing from bit 0 to bit 62. Bit 63 is used as the sign bit. The value of the integer is in the range -2^{63} to $2^{63}-1$.

qualifier A portion of a command string that modifies a command verb or command parameter by selecting one of several options. A qualifier, if present, follows the command verb or parameter to which it applies and is in the format: "/qualifier:option." For example, in the command string "PRINT filename/COPIES:3," the COPIES qualifier indicates that the user wants three copies of a given file printed.

queue n. A circular, doubly-linked list. See system queues. v. To make an entry in a list or table, perhaps using the INSQUE instruction.

queue priority The priority assigned to a job placed in a spooler queue or a batch queue.

quota The total amount of a system resource, such as CPU time, that a job is allowed to use in an accounting period, as specified by the system manager in the user authorization file. See also limit.

random access by record's file address The retrieval of a record by its unique address, which is provided to the program by RMS. The method of access can be used to randomly access a sequentially organized file containing variable length records.

random access by relative record number The retrieval or storage of a record by specifying its position relative to the beginning of the file.

read access type An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

real-time process A process assigned to a software priority level between 16 and 31, inclusive. The scheduling priority assigned to a real-time process is never modified by the scheduler, although it can be modified by the system manager or process itself.

record A collection of adjacent items of data treated as a unit. A logical record can be of any length whose significance is determined by the programmer. A physical record is a device-dependent collection of contiguous bytes such as a block on a disk, or a collection of bytes sent to or received from a record-oriented device.

record access block (RAB) An RMS user data structure that represents a request for a record access stream. An RAB relates to operations on the records within a file, such as UPDATE, DELETE, or GET.

record access mode The method used in RMS for retrieving and storing records in a file. One of four methods: sequential, or random access by key, by record's file address, or by relative record number.

record cell A fixed-length area in a relatively organized file that is used to contain one record.

Record Management Services A set of operating system system procedures that are called by programs to process files and records within files. RMS allows programs to issue GET and PUT requests at the record level (record I/O) as well as read and write blocks (block I/O). RMS is an integral part of the system software. RMS procedures run in executive mode.

record-oriented device A device such as a terminal, line printer, or card reader, on which the largest unit of data that a program can access is the device's physical record.

record's file address The unique address of a record in a file that allows records to be accessed randomly regardless of file organization.

record slot A fixed length area in a relatively organized file that is used to contain one record.

re-entrant code Code that is never modified during execution. It is possible to let many users share the same copy of a procedure or program written as re-entrant code.

register A storage location in hardware logic other than main memory. See also general register, processor register, and device register.

relocatable object module An object module whose addresses are relative, not absolute. The module can be linked to different portions of the virtual address space (relocated) without damaging the internal consistency of address references.

register deferred indexed mode An indexed addressing mode in which the base operand specifier uses register deferred mode addressing.

register deferred mode In register deferred mode addressing, the contents of the specified register are used as the address of the actual instruction operand.

register mode In register mode addressing, the contents of the specified register are used as the actual instruction operand.

relative file organization A file organization in which the file contains fixed length record cells. Each cell is assigned a consecutive number that represents its position relative to the beginning of a file. Records within each cell can be as big as or smaller than the cell. Relative file organization permits sequential record access, random record access by record number, and random record access by record's file address.

resource A physical part of the computer system such as a device or memory, or an interlocked data structure such as a mutex. Quotas and limits control the use of physical resources.

resource wait mode An execution state in which a process indicates that it will wait until a system resource becomes available when it issues a service request requiring a resource. If a process wants notification when a resource is not available, it can disable resource wait mode during program execution.

Run Time Procedure Library The collection of procedures available to native mode images at run time. These library procedures (such as trigonometric functions) may be used by all native mode images, regardless of the language processor used to compile or assemble the program.

scatter/gather The ability to transfer in one I/O operation data from discontiguous pages in memory to contiguous blocks on disk, or data from contiguous blocks on disk to discontiguous pages in memory.

scheduler The interrupt service routine responsible for process execution scheduling.

secondary storage Random access mass storage.

secondary vector A location that identifies the starting address of an exception handler to be executed when an exception occurs and either the primary vector contains zero or the handler to which the primary vector points chooses not to handle the exception condition.

section A portion of process virtual memory that has common memory management attributes (protection, access, cluster factor, etc.). It is created from an image section, a disk file, or as the result of a Create Virtual Address Space system service. See global section, private section, image section, and program section.

sequential access mode The retrieval or storage of records in which a program successively reads or writes records one after the other in the order in which they appear, starting and ending at any arbitrary point in the file.

sequential file organization A file organization in which records appear in the order in which they were originally written. The records can be fixed length or variable length. Although one does not speak of record slots with sequentially organized files, for purposes of comparison with relatively organized files one can say that the record itself is the same as its record slot, and its record number is the same as its relative slot number. Sequential file organization permits sequential record access and random access by record's file address. Sequential file organization with fixed length records also permits random access by relative record number.

shareable image An image that has all of its internal references resolved, but which must be linked with an object module(s) to produce an executable image. A shareable image cannot be executed. A shareable image file can be used to contain a library of routines and can be installed as a global section by the system manager.

shell process A predefined process that the job initiator copies to create the minimum context necessary to establish a process.

signal 1. An electrical impulse conveying information. 2. The software mechanism used to indicate that an exception condition was detected.

slave terminal A terminal from which it is not possible to issue commands to the command interpreter. A terminal assigned to application software.

small process A system process that has no control region in its virtual address space and has an abbreviated context. Examples are the working set swapper and the null process. A small process is scheduled in the same manner as user processes, but must remain resident during its execution.

software context The context maintained by the operating system that describes a process. See software process control block (PCB).

software interrupt An interrupt generated on interrupt priority level 1 through 15 which can be requested only by software.

software process control block (PCB) The data structure used to contain a process's software context. The operating system defines a software PCB for every process when the process is created. The software PCB includes the following kinds of information about the process: current state; storage address if it is swapped out of memory; unique identification of the process, and address of the process header (which contains the hardware PCB). The software PCB resides in system region virtual address space. It is not swapped with a process.

software priority See process priority and queue priority.

spooling Output spooling: The method by which output to a low-speed peripheral device (such as a line printer) is placed into queues maintained on a high-speed device (such as a disk) to await transmission to the low-speed device. Input spooling: The method by which input from a low-speed peripheral (such as the card reader) is placed into queues maintained on a high-speed device (such as a disk) to await transmission to a job processing that input.

spool queue The list of files (supplied by processes) that are to be processed by a symbiont. For example, a line printer queue is a list of files to be printed on the line printer.

stack An area of memory set aside for temporary storage, or for procedure and interrupt service linkages. A stack uses the last-in, first-out concept. As items are added to ("pushed on") the stack, the stack pointer decrements. As items are retrieved from ("popped off") the stack, the stack pointer increments.

stack frame A standard data structure built on the stack during a procedure call, starting from the location addressed by the FP to lower addresses, and popped off during a return from procedure. Also called call frame.

stack pointer General register 14 (R14). SP contains the address of the top (lowest address) of the processor-defined stack. Reference to SP will access one of the five possible stack pointers, kernel, executive, supervisor, user, or interrupt, depending on the value in the current mode and interrupt stack bits in the Processor Status Longword (PSL).

state queue A list of processes in a particular processing state. The scheduler uses state queues to keep track of processes's eligibility to execute. They include: processes waiting for a common event flag, suspended processes, and executable hibernating processes.

status code A longword value that indicates the success or failure of a specific function. For example, system services always return a status code in R0 upon completion.

store through See write through.

strong definition Definition of a global symbol that is explicitly available for reference by modules linked with the module in which the definition occurs. The linker always lists a global symbol with a strong definition in the symbol portion of the map. The librarian always includes a global symbol with a strong definition in the global symbol table of a library.

strong reference A reference to a global symbol in an object module that requests the linker to report an error if it does not find a definition for the symbol during linking. If a library contains the definition, the linker incorporates the library module defining the global symbol into the image containing the strong reference.

subprocess A subsidiary process created by another process. The process that creates a subprocess is its owner. A subprocess receives resource quotas and limits from its owner. When an owner process is removed from the system, all its subprocesses (and their subprocesses) are also removed.

supervisor mode The third most privileged processor access mode (mode 2). The operating system's command interpreter runs in supervisor mode.

suspension A state in which a process is inactive, but known to the system. A suspended process becomes active again only when another process requests the operating system to resume it. Contrast with hibernation.

swap mode A process execution state that determines the eligibility of a process to be swapped out of the balance set. If process swap mode is disabled, the process working set is locked in the balance set.

swapping The method for sharing memory resources among several processes by writing an entire working set to secondary storage (swap out) and reading another working set into memory (swap in). For example, a process's working set can be written to secondary storage while the process is waiting for I/O completion on a slow device. It is brought back into the balance set when I/O completes. Contrast with paging.

switch See (command) qualifier.

symbiont A full process that transfers record-oriented data to or from a mass storage device. For example, an input symbiont transfers data from card readers to disks. An output symbiont transfers data from disks to line printers.

symbiont manager The function (in the system process called the job controller) that maintains spool queues, and dynamically creates symbiont processes to perform the necessary I/O operations.

symbol See local symbol, global symbol, and universal global symbol.

Synchronous Backplane Interconnect (SBI) The part of the hardware that interconnects the processor, memory controllers, MASSBUS adaptors, the UNIBUS adaptor.

synchronous record operation A mode of record processing in which a user program issues a record read or write request and then waits until that request is fulfilled before continuing to execute.

system In the context "system, owner, group, world," system is the category of group numbers less than 8 in the User Identification Code. Such UICs belong to the operating system and its controlling privileged users, the system operators and system manager.

system address space See system space and system region.

system base register (SBR) A processor register containing the physical address of the base of the system page table.

system buffered I/O An I/O operation, such as terminal or mailbox I/O, in which an intermediate buffer from the system buffer pool is used instead of a process-specified buffer. Contrast with direct I/O.

system control block (SCB) The data structure in system space that contains all the interrupt and exception vectors known to the system.

system control block base register (SCBB) A processor register containing the base address of the system control block.

system device The random access mass storage device unit on which the volume containing the operating system software resides.

system dynamic memory Memory reserved for the operating system to allocate as needed for temporary storage. For example, when an image issues an I/O request, system dynamic memory is used to contain the I/O request packet. Each process has a limit on the amount of system dynamic memory that can be allocated for its use at one time.

System Identification Register A processor register which contains the processor type and serial number.

system image The image that is read into memory from secondary storage when the system is started up.

system length register (SLR) A processor register containing the length of the system page table in longwords, that is, the number of page table entries in the system region page table.

system page table (SPT) The data structure that maps the system region virtual addresses, including the addresses used to refer to the process page tables. The system page table (SPT) contains one page table entry (PTE) for each page of system region virtual memory. The physical base address of the SPT is contained in a register called the SBR.

system process A process that provides system-level functions. Any process that is part of the operating system. See also full process, small process, fork process.

system programmer A person who designs and/or writes operating systems, or who designs and writes procedures or programs that provide general purpose services for an application system.

system queues A queue used and maintained by operating system procedures. See also state queues.

system region The third quarter of virtual address space. The lowest-addressed half of system space. Virtual addresses in the system region are shareable between processes. Some of the data structures mapped by system region virtual addresses are: system service vectors, the executive, the Record Management Services, the system control block (SCB), the system page table (SPT), and process page tables.

system services Procedures provided by the operating system that can be called by user images.

system space The highest-addressed half of virtual address space. See also system region.

system virtual address A virtual address identifying a location in system space.

system virtual space See system space.

task An RSX-11/IAS term for a process and image bound together.

terminal The general name for those peripheral devices that have keyboards and video screens or printers. Under program control, a terminal enables people to type commands and data on a keyboard and receive messages on a video screen or printer. Examples of terminals are the LA36 DECwriter hard-copy terminal and VT52 video display terminal.

time-critical process See real-time process.

timer An interrupt service routine, the hardware timer ISR, maintains the time of day. A software ISR subroutine scans for device timeouts and performs time dependent scheduling upon request.

track A collection of blocks at a single radius on one recording surface of a disk.

transfer address The address of the location containing a program entry point (the first instruction to execute).

transition A page is said to be in transition when it is being written to backup storage from the modified page list.

translation buffer An internal processor cache containing translation for recently used virtual addresses.

trap An exception condition that occurs at the end of the instruction that caused the exception. The PC saved on the stack is the address of the next instruction that would normally have been executed. All software can enable and disable some of the trap conditions with a single instruction.

trap enables Three bits in the Processor Status Word that control the processor's action on certain arithmetic exceptions.

two's complement A binary representation for integers in which a negative number is one greater than the bit complement of the positive number.

two-way associative cache A cache organization which has two groups of directly mapped blocks. Each group contains several blocks for each index position in the cache. A block of data from main memory can go into any group at its proper index position. A two-way associative cache is a compromise between the extremes of fully associative and direct mapping cache organizations that takes advantage of the features of both.

type ahead A terminal handling technique in which the user can enter commands and data while the software is processing a previously entered command. The commands typed ahead are not echoed on the terminal until the command processor is ready to process them. They are held in a type ahead buffer.

unit record device A device such as a card reader or line printer.

universal global symbol A global symbol in a shareable image that can be used by modules linked with that shareable image. Universal global symbols are typically a subset of all the global symbols in a shareable image. When creating a shareable image, the linker ensures that universal global symbols remain available for reference after symbols have been resolved.

unwind the call stack To remove call frames from the stack by tracing back through nested procedure calls using the current contents of the FP register and FP register contents stored on the stack for each call frame.

user authorization file A file containing an entry for every user that the system manager authorizes to gain access to the system. Each entry identifies the user name, password, default account, User Identification Code (UIC), quotas, limits, and privileges assigned to individuals who use the system.

user environment test package (UETP) A collection of routines that verify that the hardware and software systems are complete, properly installed, and ready to be used.

User File Directory (UFD) See directory.

User Identification Code (UIC) The pair of numbers assigned to users, and to files, global sections, common event flag clusters, and mailboxes. It consists of a group number and a member number separated by a comma. The UIC specifies the type of access (read and/or write access, and in the case of files, execute and/or delete access) available to the owners, group, world, and system.

user mode The least privileged processor access mode (mode 3). User processes and the Run Time library procedures run in user mode.

user name The name that a person types on a terminal to log on to the system.

user number See member number.

user privileges The privileges granted a user by the system manager. See process privileges.

utility A program that provides a set of related general purpose functions, such as a program development utility (an editor, a linker, etc.), a file management utility (file copy or file format translation program), or operations management utility (disk backup/restore, diagnostic program, etc.).

variable-length record format A file format in which records are not necessarily the same length.

variable with fixed-length control record format A file format in which records of variable length contain an additional fixed-length control area. The control area may be used to contain file line numbers and/or print format control characters.

vector 1. A interrupt or exception vector is a storage location, known to the system, that contains the starting address of a procedure to be executed when a given interrupt or exception occurs. The system defines separate vectors for each interrupting device controller and for classes of exceptions. Each system vector is a longword. 2. For the purposes of exception handling, users can declare up to two software exception vectors (primary and secondary) for each of the four access modes. Each vector contains the address of a condition handler. 3. A one-dimensional array.

version number 1. The field following the file type in a file specification. It begins with a period (.) or a semicolon (;) and is followed by a decimal number which generally identifies it as the latest file created of all files having the identical file specification but for version number. 2. The number used to identify the revision level of program.

virtual address A 32-bit integer identifying a byte "location" in virtual address space. The memory management hardware translates a virtual address to a physical address. The term virtual address may also refer to the address used to identify a virtual block on a mass storage device.

virtual address space The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction or data. The virtual address space seen by the programmer is a linear array of 4,294,967,296 (2^{32}) byte addresses.

virtual block A block on a mass storage device referred to by its file-relative address rather than its logical (volume-oriented) or physical (device-oriented) address. The first block of a file is always virtual block 1.

virtual I/O functions A set of I/O functions that must be interpreted by an ancillary control process.

virtual memory The set of storage locations in physical memory and on disk that are referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for non-resident virtual memory.

virtual page number The virtual address of a page of virtual memory.

volume A mass storage medium such as a disk pack or reel of magnetic tape.

volume set The file-structured collection of data residing on one or more mass storage media.

wait To become inactive. A process enters a process wait state when the process suspends itself, hibernates, or declares that it needs to wait for an event, resource, mutex, etc.

wake To activate a hibernating process. A hibernating process can be awakened by another process or a scheduled wake-up call.

weak definition Definition of a global symbol that is not explicitly available for reference by modules linked with the module in which the definition occurs. The librarian does not include a global symbol with a weak definition in the global symbol table of a library. Weak definitions are often used when creating libraries to identify those global symbols that are needed only if the module containing them is otherwise linked with a program.

weak reference A reference to a global symbol that requests the linker not to report an error or to search the default library's global symbol table to resolve the reference if the definition is not in the modules explicitly supplied to the linker. Weak references are often used when creating object modules to identify those global symbols that may not be needed at run time.

wild card A symbol, such as an asterisk, that is used in place of a file name, file type, directory name, or version number in a file specification to indicate "all" for the given field.

window See mapping window.

word Two contiguous bytes (16 bits) starting on an addressable byte boundary. Bits are numbered from the right, 0 through 15. A word is identified by the address of the byte containing bit 0. When interpreted arithmetically, a word is a 2's complement integer with significance increasing from bit 0 to bit 14. If interpreted as a signed integer, bit 15 is the sign bit. The value of the integer is in the range −32,768 to 32,767. When interpreted as an unsigned integer, significance increases from bit 0 through bit 15 and the value is in the range 0 through 65,535.

working set The set of pages in process address space to which an executing process can refer without incurring a page fault. The working set must be resident in memory for the process to execute. The remaining pages of that process, if any, are either in memory and not in the process working set or they are on secondary storage.

working set swapper A system process that brings process working sets into the balance set and removes them from the balance set.

world In the context "system, owner, group, world," world refers to all users, including the system operators, the system manager, and users, both in an owner's group and in any other group.

write access type The specified operand of an instruction or procedure is written only during that instruction's or procedure's execution.

write allocate A cache management technique in which cache is allocated on a write miss as well as on the usual read miss.

write back A cache management technique in which data from a write operation to cache is copied into main memory only when the data in cache must be overwritten. This results in temporary inconsistencies between cache and main memory. Contrast with write through.

write through A cache management technique in which data from a write operation is copied in both cache and main memory. Cache and main memory data are always consistent. Contrast with write back.

INDEX

NOTES

digital

ORDER CODE: EB-21812-20

PRINTED IN U.S.A. EB 21812 20/83 040 04 40 COPYRIGHT © 1983 DIGITAL EQUIPMENT CORPORATION ALL RIGHTS RESERVED

digital

VAX SOFTWARE HANDBOOK

1982-83