SOFTWARE HANDBOOK
DIGITAL Facility, Westminster, MA

CORPORATE PROFILE

Digital Equipment Corporation designs, manufactures, sells and services computers and associated peripheral equipment, and related software and supplies. The Company's products are used world-wide in a wide variety of applications and programs, including scientific research, computation, communications, education, data analysis, industrial control, timesharing, commercial data processing, word processing, health care, instrumentation, engineering and simulation.
VAX

SOFTWARE HANDBOOK

digital
Digital Equipment Corporation makes no representation that the interconnection of its products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting of license to make, use or sell equipment constructed in accordance with this description.

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this manual.

DEC, DECnet, DECSYSTEM-10, DECSYSTEM-20, DECTape, DECUS, DECwriter, DIBOL, Digital logo, IAS, MASSBUS, OMNIBUS, PDP, PDT, RSTS, RSX, SBI, UNIBUS, VAX, VMS, VT are trademarks of Digital Equipment Corporation

This handbook was designed, produced and typeset by DIGITAL’s New Products Marketing using an in-house text-processing system operating on a DECSYSTEM-20.

Copyright© 1980 Digital Equipment Corporation. All Rights Reserved.
# TABLE OF CONTENTS

**PREFACE HOW TO USE THIS HANDBOOK** .................................. vii

**CHAPTER 1 WHAT IS AN OPERATING SYSTEM?** ............................. 1
  INTRODUCTION ............................................................................. 1
  DESIGN OF AN OPERATING SYSTEM .......................................... 2
  INFORMATION IN AN OPERATING SYSTEM ................................. 3
  VIRTUAL ADDRESS SPACE .......................................................... 7
  COMPUTER NUMBERS .................................................................. 7
  A TYPICAL PROCESS .................................................................... 8
  DIGITAL COMMAND LANGUAGE .................................................. 9

**CHAPTER 2 INTRODUCTION TO THE VAX SOFTWARE** .................. 10
  SYSTEM INTRODUCTION .............................................................. 11
  USER PROCESS ........................................................................... 12
  VIRTUAL MEMORY AND MEMORY MANAGEMENT ...................... 13
  SWAPPING AND SCHEDULING ....................................................... 15
  SYSTEM SERVICES ...................................................................... 16
  INTERPROCESS COMMUNICATION AND SYNCHRONIZATION ........... 18
  INPUT/OUTPUT .......................................................................... 18
  REAL-TIME ENVIRONMENT .......................................................... 19
  I/O DRIVERS .............................................................................. 21
  COMMUNICATIONS SERVICES ....................................................... 21
  PROGRAMMING LANGUAGES ....................................................... 22
  PROGRAM DEVELOPMENT TOOLS ............................................ 23
  UTILITY LANGUAGES ................................................................. 25

**CHAPTER 3 THE SYSTEM MANAGER** ......................................... 26
  INTRODUCTION ........................................................................... 27
  GETTING THE SYSTEM UP ........................................................... 27
  USER ENVIRONMENT TEST PACKAGE ........................................... 28
  A SYSTEM OF ACCOUNTS ............................................................. 29
  MANAGING PUBLIC FILES AND VOLUMES ................................. 33
  CONTROLLING SYSTEM PERFORMANCE .................................... 36
  MONITORING SYSTEM ACTIVITY .................................................. 38
  DEALING WITH ERRORS AND FAILURES ..................................... 40

**CHAPTER 4 THE SYSTEM USER** .................................................. 42
  SYSTEM ACCESS ......................................................................... 43
  FILES ......................................................................................... 46
  LOGICAL FILE NAMES ............................................................... 51
  PROGRAM DEVELOPMENT .......................................................... 53
SWAPPING ................................................................. 362
PAGING IN SYSTEM SPACE ........................................ 363

CHAPTER 14 PROCESS SCHEDULING AND SWAPPING .... 364
INTRODUCTION ......................................................... 365
SCHEDULING .......................................................... 366
WORKING SET SWAPPING .......................................... 374

CHAPTER 15 INTERPROCESS COMMUNICATION ............ 380
INTRODUCTION ......................................................... 381
COMMON EVENT FLAGS ............................................. 381
MAILBOXES ........................................................... 382
SHARED AREAS OF MEMORY ...................................... 383

CHAPTER 16 SPECIAL EVENT HANDLING .................. 386
INTRODUCTION ......................................................... 387
CONDITION HANDLERS ............................................. 387
SEARCHING FOR A CONDITION HANDLER ................. 388
EXIT HANDLERS ...................................................... 391
ASYNCRONOUS SYSTEM TRAPS .................................. 392

CHAPTER 17 PDP-11 COMPATIBILITY ....................... 398
OVERVIEW ............................................................ 399
IMPLEMENTATION CONSIDERATIONS ......................... 401

APPENDIX A COMMONLY USED MNEMONICS ............... 407
GLOSSARY OF SOFTWARE TERMS .............................. 411
INDEX ................................................................. 453
PREFACE
HOW TO USE THIS HANDBOOK

It is not necessary to read straight through this Handbook to get a view of the powerful new VAX software. The book has been written in small groups of chapters that allow specialized readers to see quickly how the VAX family incorporates power with simplicity and sophistication with speed.

The Table of Contents gives details of the chapters and should help you find material of interest. For example, if you are reading about computers for the first time, you will probably want to read Chapter 1, “What Is an Operating System?” and the subsequent introductory chapter before attempting to study the more user-oriented matter in later parts of the Handbook.

Readers interested in networking might like to skip to the sections on Communications and Internets to see, for example, how DIGITAL Network Architecture is implemented in DECnet-VAX.

Operators and users will want to go to the chapters that show login procedures, system services, the command language, and the wide variety of programming languages available. Or they may want to read about the Record Management Services (RMS) to discover the power of this software in complex record manipulation tasks.

System managers can find, in the chapter titled “The System Manager,” a brief description of the powers that they would have under the implementation of a VAX system, including granting privileges and management of accounting information. They will also see the ease and efficiency of setting up and verifying a VAX/VMS system.

The Handbook’s goal is the introduction of the system software, without delving into the degree of technicality one finds in the user documentation delivered with the system. It is equipped with a glossary of software terms, an alphabetic listing of most commonly used mnemonics, and a thorough index. In addition, there are two companion volumes intended to give the reader a complete overview of and introduction to the unusual capabilities of VAX. They are the VAX-11/780 Hardware Handbook and the VAX-11 Architecture Handbook.
CHAPTER OVERVIEW
For the reader without computer experience, this chapter introduces
the idea of an operating system. It focuses particularly on the
VAX/VMS Virtual Memory Operating System of the VAX family and
defines some fundamental terms necessary to understand software.
Topics include:
• Role of the Operating System
• Design of an Operating System
• Information in an Operating System
• A Typical Process
• Virtual Address Space
• Computer Numbers
• Command Language
CHAPTER 1
WHAT IS AN OPERATING SYSTEM?

INTRODUCTION
An operating system is a collection of control programs and routines designed to make computer hardware devices easily usable by people. Operating systems vary greatly in the kinds of hardware with which they are compatible, the range of complexity of tasks they handle, the degree of adaptability to special user purposes, and the programming languages which they support.

Operating systems not only provide a way by which the user’s specific program can run on the computer, but they can also have a set of utilities and routines which manage such resources as printers and terminals, detect errors in programs, keep user accounts, protect information, warn the operator of failures—and much more.

![Diagram](image)

Figure 1-1

Designing an operating system is a difficult task, involving an intimate knowledge of the hardware architecture and an understanding of the kinds of problems users are going to want to solve.

Some families of computers, such as DIGITAL’s PDP-11 family, operate under a variety of systems, depending on the particular tasks to be done. For example, there are systems that manage one user’s task at a time. There are others that accept many, and allocate the central processor according to a scheme of priorities, privileges, and time quotas. Some systems support real-time applications, that is, applications in which the computer is required to respond within given time limits to an external message, make a decision, and respond quickly (i.e., flight simulator control; power plant management; laboratory experiment supervision).

The VAX family hardware and the operating system—called VAX/VMS, for Virtual Memory System—were designed to comple-
What Is An Operating System?

VAX/VMS is the only DIGITAL operating system that is supported for VAX, and as the rest of this book shows, it can handle a tremendous range of tasks, languages, and hardware configurations. In addition, the operating system is always being improved: it is easy to incorporate these improvements into existing applications because of release-to-release compatibility.

Design of an Operating System

VAX/VMS is a disk resident operating system. Portions of it are loaded into the memory whenever the system is bootstrapped. After that, any portions of the operating system that are not in memory are moved ("paged") into the memory as needed.

An operating system performs many functions:

• Process management: allows the user to separate tasks into manageable pieces called processes.
• Memory management: allocates space in memory on a per-process basis.
• Processor scheduling: allocates central processor unit (CPU) time according to a defined algorithm. The goal is to provide balanced use of the CPU to the processes resident in memory, based upon priorities and events.
• Interprocess communication: allows two or more processes to convey information to each other.
• Input/output services: manages the flow of information among various parts of the computer, and between users and the computer; and manages the organization of data into records and files.
• Network: enables computers to link with other computers to facilitate distributed processing, geographical distribution, and project sharing.
• Language processor: translates programs written in a specific language into a binary form that can be executed by the computer.
• Error logging and recovery: reports errors and often enables transparent recovery in the event of hardware or software failure.
• Input/output drivers: actually control the interfaces to peripheral devices such as video terminals, lineprinters, and so on.

Other features include libraries of frequently used procedures (which the programmer merely CALLs, rather than rewriting for each application), debugging facilities for seeking potential program errors, a linker that prepares images for execution, and special functions useful to specific applications.

Each of these topics is considered in greater detail in later chapters in this Handbook.
The operating system is further divided into modules, which are units of related procedures. VAX/VMS keeps track of where its modules are located, and calls up the right one for each need. An advantage of VAX/VMS is that it supplies tools and conveniences which end the user's need to know either fixed addresses or complicated data structures. For example, the symbolic debugger eliminates the need to memorize fixed addresses.

**Information in an Operating System**

At the most fundamental level, computers deal in the binary digits 0 and 1, and all other information—numbers, words, characters, graphic symbols—must eventually be coded into patterns of binary digits if the computer is to process it. To simplify coding, there are certain standards in the computer industry that provide fixed ways of binary coding.

Some of the important definitions follow. A binary digit is called a **bit**. Eight binary digits are grouped together in a **byte**. For most VAX/VMS considerations, the byte is the smallest addressable unit of information.

The term **word** refers to a double byte (16 bits), and the **longword** is a double word (32 bits). In the VAX/VMS operating system, a longword is needed to contain an address of a datum; that is, it requires 32 binary digits in a particular pattern to specify any location of information. There is also the **quadword**, which is 64 bits long.

For VAX systems, a **page** is a 512-byte collection of related information.
The actual representation of numbers, words, or other data is a subtle process, determined by the possibilities provided by the system architecture. The programmer then may choose from among possibilities. By way of illustration, the same word pattern of bits could represent, depending upon programmer choices and environmental constraints, a positive integer, a negative integer, a fraction, a letter of the alphabet, part of an instruction to the computer, or a pair of decimal digits.
What Is An Operating System?

But usually programmers do not write programs directly at the binary level. It is tedious and highly susceptible to errors. Instead, they write programs in languages—either languages that use easy-to-remember code words called mnemonics, or ones that have English-like words and phrases. Then it is the job of the translators that run under the operating system to translate these accurately to binary code.

The programs that manage such translations are utilities called assemblers and compilers. Which one is used depends on the form of the source program supplied by the programmer. When the assembler is used, one mnemonic instruction usually corresponds to one hardware instruction. Thus a mnemonic such as MOVW (which tells the processor to move a 16-bit word to a specified location) corresponds to one instruction in the hardware. This mnemonic level assembler language is called VAX-11 MACRO.

Compilers and interpreters, on the other hand, are the translating programs of the operating system that work on higher level languages—in which one instruction may represent many hardware instructions. FORTRAN, COBOL, PASCAL, BASIC, PL/I, DATATRIEVE, and BLISS are some of the higher level languages for which compilers or interpreters are available in the VAX/VMS operating system. As described in the chapter on languages, each is generally designed for a category of program needs, and in all likelihood a VAX computer application would not actually use all the higher level language processors.

Compilers and interpreters can do more than translation. They allocate and optimize resources in the machine. They also usually perform syntax checks upon the source code and provide feedback on the type, location, and severity of any errors they find.

Other terms important in understanding how the VAX/VMS operating system manages information are "record," "file," "directory," and "field." A record is simply a coded collection of information. Perhaps a record might be the salary history of a particular employee or the formula for a certain medicine. For a scientific user, a record might be the location of a weather satellite at a particular moment.

Files are collections of related records. All employee salary histories, the formulas for all the different medicines a company manufactures, and a day's locations of the weather satellites are files. The manipulation of data files under instructions from the source program is managed by the operating system. New records may be inserted, old records updated, or the entire file scanned by a variety of different indices.
What Is An Operating System?

A directory is a list of the files. For example, a user may have 20 different files resident on a disk. That user's directory contains the names of files and pointers to where they are stored on the disk. Entering a directory name and a command produces a listing of the names of files in the directory. For ease of organization, directories may contain subdirectories down to many levels. Files can thus be organized by related uses in much the same way as files in a filing cabinet.

Fields are pre-defined parts of records. The first 10 bytes in the employee's salary history record could be the name field, the next 9 bytes could be the social security number field, and so on. In the satellite location record; bytes 21 to 50 could be the time-of-day field. When the operating system is instructed to scan an entire file by key, the key may correspond to some field. So that, for example, the operating system might be asked to list all employees whose salary field contained a number higher than $15,000 (primary key field) and whose job category field contained a 2 (secondary key field).

<table>
<thead>
<tr>
<th>RECORD: PLASTIC WIDGETS</th>
<th>PLASTIC WIDGETS</th>
<th>16.031</th>
<th>10/11/82</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NAME FIELD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INVENTORY FIELD</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>INVENTORY DATE FIELD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FILE : ALL WIDGETS</th>
<th>METAL WIDGETS</th>
<th>5.206</th>
<th>10/6/82</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PLASTIC WIDGETS</td>
<td>16.031</td>
<td>10/11/82</td>
</tr>
<tr>
<td></td>
<td>WOODEN WIDGETS</td>
<td>17.429</td>
<td>10/14/82</td>
</tr>
<tr>
<td></td>
<td>RUBBER WIDGETS</td>
<td>1.003</td>
<td>10/4/82</td>
</tr>
</tbody>
</table>

Figure 1-3

Records may be structured and accessed in a variety of ways, and in the VAX/VMS operating system, a set of routines collectively called the Record Management Services (RMS) handles this task.

Optionally, the sophisticated user may do his own file manipulations and structuring, just as he may create and manipulate his own directories and libraries, handle his own input/output procedures, or override many of the system's other built-in capabilities. Usually the reason for doing this would be to increase efficiency in an often-repeated, complicated task, and thereby decrease computer time and get quicker responses.
Virtual Address Space
Just as the parcels of land in an undeveloped tract can be assigned street addresses even before the streets are paved or any houses are built, so, too, bytes of binary code can have addresses even though those addresses have no physical correspondences in the physical memory. Virtual Address Space is the term used to describe the collection of all possible addresses. Since an address can have 32 binary digits, there are $2^{32}$ such possible addresses.

This number, almost 4.3 billion, is far greater than the number of locations in the physical memory of the VAX today; it also exceeds any current program needs. A large portion of the virtual space is available to user processes. The operating system handles movement from secondary storage addresses to physical memory addresses as needed. Hardware developments may in future make a larger part of Virtual Address Space actual, but for now the benefit to users is that they need not worry about compressing their programs into minimal space. Nor need they worry about overlaying or other length reduction techniques.

The management which the Virtual Memory System offers is transparent to the user, who need not consider, in writing applications, that only a small part of the process will probably be resident in the physical memory at any time.

![Table of Addresses](image)

**Figure 1-4**

Computer Numbers
It may seem that some of the numbers associated with computers are arbitrarily chosen. The numbers 8, 16, 32, 64, 512, $2^{32}$ show up again
and again in this Handbook. Even the symbol \( K \), which in ordinary usage stands for 1000, in computer terminology means 1024. This assignment is not a whim of system designers, but rather arises from the binary nature of the fundamental instructions and data representation used in computers. All of the important characteristics are powers of 2, or represent the minimum space necessary to translate a character to a binary code, a pattern of 0s and 1s. Binary forms are the representations a computer can work with, and all numbers, letters, and characters must eventually be coded into binary forms if they are to be useful to a computer. Most computers do the translation into binary themselves, so that users need not worry about these details.

Occasionally throughout this Handbook, two other number systems will be mentioned (besides the decimal system). They are the octal, or base 8, system, and the hexadecimal, or base 16, system.

A Typical Process
Consider a simple mathematical problem that one might use a computer to solve. Ten thousand pairs of numbers are read. The difference in each pair is computed, squared, and stored. All of these squared differences are added up, and their square root extracted. A short FORTRAN program is written to produce these instructions. For our example, the programmer types the program in at the terminal, but gets the 10,000 pairs of numbers from a real-time device. The user wants the result of this calculation typed out at a terminal.

To solve this problems, the user logs onto the system, calls the editor, and then types in the program. The file name (for the program) is then entered into a directory—in our case we'll call it SUMSQR.FOR.

Next, the programmer calls the FORTRAN compiler to translate the program into binary. The compiler might also optimize the program—improve its efficiency by finding better ways to do things. The compiler notes more subtle errors that have to be corrected. Once satisfied that the program is correct, the programmer invokes the operating system utility called the linker which binds the modules of his binary coded program with modules of system routines that it needs to run.

In this case, some of the system routines needed, such as the square root function, reside in the mathematical library. The FORTRAN program contains the mathematical function requests, but the library has the actual algorithms. The linker will assign addresses to all the information in the program and hold it ready for execution.

The operating system provides other services to the user process. It allocates the input and output devices (terminals and lineprinters)
What Is An Operating System?

when the process needs them. If the process is interrupted by other processes with higher privilege, or if the time allocation runs out before computation is complete, the system remembers where it was and picks up the calculation at that point—as often as necessary to completion.

Finally, if the process completes, but produces results that are wrong, the system provides facilities, called debugging facilities, that can help the programmer search out the error in his logic or his coding.

DIGITAL Command Language
A feature of the VAX/VMS operating system is its command language. Not exactly a programming language, the command language contains the instructions for establishing proper process environment and for other kinds of control. In our SUMSQQR program above, it would have been a command language command that brought the FORTRAN compiler from the system disk into physical memory, and another that invoked the linker. When we edited the file, command language commands would have called and run the editor. We might have allocated various resources to the process using command language. In our example, the commands were entered by the terminal operator at appropriate points in the typing of the program. But if we had submitted the entire program to a batch (non-interactive) execution stream, the same commands could also have been included in the file, along with the FORTRAN instructions.

Strings of commands can be put together into command procedures, and these procedures can be stored and called up as needed from a library, to save repetitive typing of frequently used command sequences. This quasi-programming technique is discussed in more detail in the chapter on the command language.

The rest of the Software Handbook elaborates these and other concepts. Your DIGITAL sales representatives can, if you require, supply you with companion Handbooks and information about DIGITAL Customer Training.
CHAPTER OVERVIEW
For the person familiar with computer concepts, 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, I/O Devices
- Interprocess Communication
- Communications and Internets
- Real-time Capabilities
- Languages and Language Processors
- Data Management Facilities
CHAPTER 2

VAX-11 SOFTWARE INTRODUCTION

SYSTEM INTRODUCTION
VAX is a family of high-performance multiprogramming 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. Real-time 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 extendible. 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. 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
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.

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.

Working Set
When a process executes, only a subset of its pages need be in physical memory. This subset 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
VAX/VMS 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. Each page contains 512 bytes, which is also the size of a physical page of memory and a disk block.

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 data base (the page frame number data base) describing the status of all physical pages of memory and the status and location of all virtual pages of processes in the
Introduction to the VAX Software

system. For example, a physical page could be part of a working set, or it could be available for a process virtual page to be loaded into it or on a free page list.

Memory management uses page tables as the data base 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 file.

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 globally available image 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 data base that is similar in structure to that used to describe processes. Global sections consist of a num-
ber 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 returns process working sets into the balance set and removes them from 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 real-time processes. The highest priority executable resident process is always selected for execution. Real-time 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 using the following mechanisms:
• 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 queueing 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 user 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 remain resident until 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 is performing the service on behalf of the user, functions requiring access to privileged data bases 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.

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
VAX/VMS 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 two variables: the level of explicit cooperation between the processes and the efficiency of communication.

Interprocess communication can occur using the following methods:
1. Implicit communication using a shared data base. This method is most efficient but requires explicit cooperation of the processes.
2. Generalized communication using mailboxes. Mailboxes are virtual devices to which processes can send and from which a process can read messages. This method, however, incurs 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.

In multiple-CPU systems, interprocess synchronization can also be achieved by shared global sections or by using interlock queue instructions.

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
Introduction to the VAX Software

such as OPEN, CLOSE, GET, and PUT, or, in MACRO, 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 physical 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 REAL-TIME ENVIRONMENT
The VAX hardware and VAX/VMS software have been developed together to insure a superior real-time multitasking computational
system. If real-time tasks are to be performed, the following inherent system attributes of VAX establish it as an extremely powerful system for the most demanding real-time applications:

- Highly efficient process scheduler providing 16 real-time process priorities
- Rapid process context switching
- Rapid hardware processing of interrupts
- Interrupts vectored to VAX/VMS device drivers
- VAX/VMS support of PDP-11 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 connecting to an interrupt

Since real-time 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 2-1 illustrates in layered form the VAX/VMS operating system.

![VAX/VMS Operating System Diagram](image)

**Figure 2-1** VAX/VMS Operating System

The outer layers of VAX/VMS are the more sophisticated general purpose features to ensure ease of use and functionality. These layers
constitute of command procedures, record management services, user programs, etc. The innermost layers constitute the real-time 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 the driver for the operating system procedure that maps and loads the driver and its device data base 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 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. Since 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 an application-specific device driver.

**COMMUNICATIONS SERVICES**
DECnet is the family of hardware and software network products that allow DIGITAL systems to participate in a cooperative multicomputer environment. Because of cooperation between VAX/VMS and DECnet—in such facilities as logical name services—the applications programmer may treat the networked computers as a common resource. Therefore, reconfiguration of the network does not necessarily require rewriting of the application programs. Hence, the network is transparent to the application programmer. Using DECnet, 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 products are built. DECnet is highly modular and flexible, and is designed to handle a broad range of application requirements.

To communicate with certain IBM and CDC computers, protocol emulators are available under VAX/VMS. 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 VAX/VMS, including FORTRAN, COBOL, BASIC, and PL/I. 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. Since languages can call one another, the programmer may incorporate more than one language in his application program easily. 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.

The complete list of native mode language currently available for VAX/VMS is:
- VAX-11 MACRO (assembly)
- VAX-11 FORTRAN
- VAX-11 COBOL
- VAX-11 BASIC
- VAX-11 PL/I
- VAX-11 PASCAL
- VAX-11 BLISS-32
- VAX-11 COBOL-74

Compatibility mode is a feature that allows a VAX computer to "look like" a PDP-11 computer for many types of applications. The programming environment under compatibility mode consists of language processors which produce compatibility made object code and the software tools supporting their development. The current compatibility languages are:
- MACRO-11 (assembler)
- BASIC-PLUS-2/VAX
- PDP-11 CORAL 66/VAX
- FORTRAN IV
- PDP-11 FORTRAN-IV-PLUS/VAX
VAX-11 PROGRAM DEVELOPMENT TOOLS
The VAX program development tools include text editors, a linker, a librarian, a common run time procedure library, a DIFFERENCES utility, and a symbolic debugger. These tools are available to the programmer through the VAX/VMS command language, as are the language compilers themselves.

The text editors can be used to create memos, documentation, and data files, as well as source program modules for any language processor. The linker, librarian, debugger, and run time procedure library described below are used only in conjunction with the language processors that produce native code.

For program development in compatibility mode, VAX/VMS provides compatibility mode debugger, librarian, task-builder, and object time libraries.

Editors
The programmer can use any or all of the three text editors: SOS, SLP, and EDT (the popular DEC standard editor). SOS is an interactive text editor that enables the programmer to create and modify text files using commands entered from either a hard-copy or video terminal. 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.

EDT, the DEC Standard Editor, is also an interactive editor. It allows character, word, line, and buffer editing efficiently and powerfully. In addition, EDT supports a keypad editor for users of VT100 and VT52 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.

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 symbols and procedure references between them, 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 subsequently be linked with other modules to produce an executable image. Furthermore, the linker accepts not only object modules to produce executable or shareable images, but also object module libraries, shareable images, 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 or macros or shareable images. 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 Procedure Library
The Run Time Procedure 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 run-time
- a single copy of the library to be shared by all processes
- installation of a new library without the need to relink existing programs

The run time library includes:

- A mathematical section (single and double precision trigonometric, logarithmic, and exponential functions)
- A resource allocation section (virtual memory and dynamic string functions)
- A general utility section (data type conversions)
- A condition handling section (signaling exception conditions and declaring condition handlers)
- A language-independent support section (error handling and record management services support functions)
- Several higher-language-specific support sections (file handling support functions)
Symbolic Debugger
The debugger can be linked with a native program image to control image execution. The debugger 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. Debugging 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.

VAX-11 UTILITIES
VAX-11 utilities include native mode VAX-11 SORT and PDP-11 DATATRIEVE/VAX.

DATATRIEVE
DATATRIEVE is user application software that provides direct, easy, and fast access to data contained in RMS (Record Management Services) files. While providing the user with an inquiry language and a report writing facility, DATATRIEVE also supports a user-specifiable data dictionary which defines RMS record formats. DATATRIEVE data management facilities include interactive data retrieval, sort, update and maintenance, and access to data dictionary entries that define RMS records.

VAX-11 SORT
The VAX-11 SORT utility allows the user to reorder data from any RMS input file. The MERGE facility of SORT is a callable stand-alone subroutine that allows the merging of files whether sorted recently or some time earlier. The format of the output file depends on key fields within the input data records. The sorting sequence is determined by user-specified control fields, also known as key fields, within the data itself. If the user does not wish to reorder the data base, SORT can be used to extract key information, sort that information, and store the sorted information as a permanent file. Later that file can be used to access the data base in the order of the key information in the sorted file. The contents of the sorted file may be entire records, key fields, or record file addresses which point to the position of each record within the file.
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 Virtual Memory Operating System gives very 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

26
CHAPTER 3
THE SYSTEM MANAGER

INTRODUCTION
In a VAX/VMS 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 VAX/VMS, 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, and easy adjustment of parameter files.

Practically speaking, the job of the system manager is best defined in terms of 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, VAX/VMS 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.

VAX/VMS comes pre-built. It is self-installing and autoconfiguring. That means that any VAX hardware configuration can be supported by VAX/VMS 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 work-
ing 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 down time 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, since VAX/VMS provides a “friendly” environment.

**User Environment Test Package (UETP)**

When a VAX/VMS system is first installed and brought up, an installation verification package can be useful 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 auto-configures to any buildable VAX configuration and assures the manager that hardware and the operating system are working properly together. It places VAX/VMS error messages in an error log to indicate system faults. This log will be useful to the system manager and DIGITAL Field Service in correcting any problems. 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 suspect the functional integrity of the system.

The test takes approximately 45 minutes to run. It satisfies three objectives:

1. Exercises major peripherals
2. Validates VAX/VMS system services
3. Tests major VAX/VMS software components (e.g., VAX RMS, VAX-11 SORT)

The UETP is fully automatic and requires no user interface once started. It will log start messages and stop messages at the terminal, along with messages that information has been placed in the error log, which may, if necessary, be printed later.

Errors will not kill the test; they will all be resolved within the UETP. There are no abnormal terminations upon exit. Tests do their own cleanup.

The UETP is not only a test mechanism, but it is also a mechanism for the control of other tests. Its modular design allows the sophisticated user to extend testing to special applications.
The System Manager

While the UETP is thorough, it is not exhaustive, and should not be construed as diagnostic or as replacing a diagnostic test. It does not, for example, test layered products (e.g., optional language compilers). Such products may have their own installation verification test packages in their distribution kits. The UETP itself, rather, is a functionality test that the 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 denote 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
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 management 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 on to the system, a
process is created on behalf of that user. The process acquires the characteristics of the user. These are the same that 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

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). (Through the User Authorization Program (AUTHORIZE), the manager may add, modify, delete, or display records in the User Authorization File (UAF).) The choice of UICs and their assignment involve the following two important questions:

1. Which users should be allowed to share data and file access, and which should not?
The System Manager

2. Which processes should be allowed to cooperate, and which should not?

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.

A User Identification Code (UIC)—which is explicitly assigned by the system manager—is an expression consisting of two octal numbers:
1. The first is a group number that can have a value in the range 0 through 377.
2. The second is a member number that can have a value in the range 0 through 377.

Protection and Owner, System, Group, World
For purposes of data protection, four different categories of users are defined. They are:
1. Owners—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 7 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, since 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 real-time processes, it is 16 through 31.

Processes with real-time priority are scheduled strictly according to base priority. But processes with normal priority 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, see 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 responsibilities 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 run time
- Installing frequently used shareable images as permanent global sections or images that may be shared at run time
- Establishing system-wide 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 run-time 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.

Just as preserving information on public volumes by backing them up is usually considered desirable, preserving files on private volumes is also considered desirable. However, responsibility for backing up the files on private volumes usually is left to the individual owners of those files and volumes.

There are two kinds of backups 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 permanent global sections.
There are two reasons for installing known images:
1. To permit system-wide 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 DCL 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 system-wide 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 data base, 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 line printers 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 high-speed storage device to buffer data passing between low-speed I/O devices and high-speed main memory. The low-speed devices, which can be either the ulti-
mate sources or the ultimate destinations of buffered I/O data, are called spooled devices; the high-speed mass storage devices are called intermediate devices.

Typically, the system manager chooses low-speed 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 line printer is set spooled when the system is started up. In a system with only one line printer, 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 $JOB card specifies otherwise, the name of his batch queue is SYS$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 line printer is associated with a physical queue (a queue that has the same name as the line printer) and unless that queue has been started (along with a generic print queue), no queued output can occur on that line printer.

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 Display Utility Program
The Display Utility Program can display system performance measurement statistics on a DECscope VT100 video display terminal, a DECscope VT105 video display terminal, or similar DIGITAL terminal. Such displays take the forms of graphs and tables.

The VAX/VMS operating system collects data on how the system is being used and how it responds to users' requests. The statistics are gathered 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:

- File system statistics
- I/O system activity
- Use of processor modes
- Page management statistics
- Nonpaged pool statistics
- Activity in the scheduler state queues
- Principal users of CPU time
- System process activity

Each time the system is booted, 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.

Figure 3-1
This particular graph measures percentage of time in the processor modes available.

The processor modes monitored are:
- **INTER**—percent of time processor was executing on interrupt stack
- **KERNEL**—percent of time processor was executing in kernel mode (does not include time on interrupt stack)
- **EXEC**—percent of time processor was executing in executive mode
- **SUPER**—percent of time processor was executing in supervisor mode
- **USER**—percent of time processor as executing in user mode (does not include time in compatibility mode)
- **COMPAT**—percent of time processor was executing compatibility mode user images
- **IDLE**—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. Corrective action can be taken before such failures then 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.

---

**Figure 3-2**
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 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 4
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 typed or displayed on the terminal.

The log-in sequence appears for a user named GING as follows:

<RET>
USER-NAME: GING <RET>
PASSWORD: <RET>

WELCOME TO VAX/VMS RELEASE Version 2.0
$

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 his default disk unless he 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 User

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

19-FEB-1980 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 actually 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.

- $<\uparrow U>$ The CTRL/U key deletes the current line and performs a carriage return, enabling the user to reenter an entire line.
- $<\uparrow R>$ 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<\uparrow R>$
$ PRINT MY$

- $<\uparrow C>$ 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 STOP command as follows:

$ TYPE MYFILE.LIS<RET>$
  .
  .

$<\uparrow C>$
$ STOP <RET>$

The HELP Command
The HELP command assists the user by displaying command-specific information. The HELP command is a useful interactive reference tool for the user not having convenient access to a reference manual. For example, to acquire information about the PRINT command, the user enters the following:

$ HELP PRINT

The system then responds:

PRINT
Queues one or more files for printing, either on a system-defined queued printer or on a specified device.
Format
PRINT file-spec,...
Additional information available:

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Qualifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>/AFTER-absolute-time</td>
<td>/DEVICE=device-name</td>
</tr>
<tr>
<td>/HOLD</td>
<td>/NOHOLD (D)</td>
</tr>
<tr>
<td>/NOIDENTIFY</td>
<td>/JOB-COUNT=n (D=1)</td>
</tr>
<tr>
<td>/NOLOWERCASE (D)</td>
<td>/NAME=job-name</td>
</tr>
<tr>
<td>/QUEUE=queue-name (D=1)</td>
<td>/BURST (D)</td>
</tr>
<tr>
<td>/DELETE</td>
<td>/NODELETE (D)</td>
</tr>
<tr>
<td>/NOFEED</td>
<td>/FLAG-PAGE (D)</td>
</tr>
<tr>
<td>/HEADER</td>
<td>/NOHEADER (D)</td>
</tr>
<tr>
<td>COUNT=n</td>
<td>/SPACE[=n] (D=1)</td>
</tr>
<tr>
<td></td>
<td>/PAGE</td>
</tr>
<tr>
<td></td>
<td>/FORMS=type</td>
</tr>
<tr>
<td></td>
<td>/IDENTIFY (D)</td>
</tr>
<tr>
<td></td>
<td>/LOWERCASE</td>
</tr>
<tr>
<td></td>
<td>/PRIORITY=n</td>
</tr>
<tr>
<td></td>
<td>/COPIES=n</td>
</tr>
<tr>
<td></td>
<td>/FEED (D)</td>
</tr>
<tr>
<td></td>
<td>/NO-FLAG-PAGE</td>
</tr>
</tbody>
</table>

The information displayed includes a synopsis of what the PRINT command does, the valid qualifiers and their default values, and the parameters required by the command.

**LOGOUT**

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

```
$ LOGOUT<RET>
```

The system responds:

```
VAX/VMS LOGOUT AT 11:30:50  19-FEB-1980
```

**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 (appropriated "file-spec").

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

Since 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 the files on that disk that belong to a particular user.

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 on to the system, the system assumes all of that user's files reside on a specific disk, allotted 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:

<table>
<thead>
<tr>
<th>File Type</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>.B2S</td>
<td>Input source file for the PDP-11 BASIC-PLUS-2/VAX (compatibility mode) compiler</td>
</tr>
<tr>
<td>.BAS</td>
<td>Input source file for the BASIC-PLUS and VAX-11 BASIC compilers</td>
</tr>
<tr>
<td>.B32 or .BLI</td>
<td>Input source file for the VAX-11 BLISS-32 compiler</td>
</tr>
<tr>
<td>.CBL</td>
<td>Input file containing source statements for the PDP-11 COBOL-74/VAX compiler</td>
</tr>
<tr>
<td>.CMD</td>
<td>Compatibility mode command procedure.</td>
</tr>
<tr>
<td>.COB</td>
<td>Input source file for the VAX-11 COBOL compiler</td>
</tr>
</tbody>
</table>
The System User

.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

.LIB Input file containing VAX-11 COBOL-74 source statements to be copied into another file during compilation

.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

If the user creates a FORTRAN program, for example, the file type .FOR would be used.
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 has three fields:
1. The device type. Each hardware device has a mnemonic. For example, an RP06 disk is DB and a TU77 magnetic tape is MT.
2. A controller designator. The controller designator identifies the hardware controller to which the device is attached. Controller A, for example, is designated by A.
3. The unit number. The unit number uniquely identifies a device on a particular controller.

Some examples of device names are:

<table>
<thead>
<tr>
<th>Name</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBA2</td>
<td>RP06 disk on controller A, unit 2</td>
</tr>
<tr>
<td>MTA0</td>
<td>TE16 magnetic tape on controller A, unit 0</td>
</tr>
<tr>
<td>TTB3</td>
<td>Terminal on controller B, unit 3</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th>Mnemonic</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CR</td>
<td>Card Reader</td>
</tr>
<tr>
<td>DB</td>
<td>RP04, RP05, RP06 Disk</td>
</tr>
<tr>
<td>DM</td>
<td>RK06, RK07 Disk</td>
</tr>
</tbody>
</table>
### Table 4-1  Device Names

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Device Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DR</td>
<td>RM03 Disk</td>
</tr>
<tr>
<td>DX</td>
<td>Floppy Disk</td>
</tr>
<tr>
<td>LAA</td>
<td>LPA11-K</td>
</tr>
<tr>
<td>LP</td>
<td>Lineprinter</td>
</tr>
<tr>
<td>MB</td>
<td>Mailbox</td>
</tr>
<tr>
<td>MT</td>
<td>TE16 Magnetic Tape/TU45/TU77</td>
</tr>
<tr>
<td>NET</td>
<td>Network communication device</td>
</tr>
<tr>
<td>TT</td>
<td>Interactive terminal</td>
</tr>
<tr>
<td>XM</td>
<td>DMC-11 Network Link Module</td>
</tr>
</tbody>
</table>

#### 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 specification.

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, an access violation error would be returned to the command.
Subdirectories, down to many levels, are possible in VAX/VMS. This useful feature allows a user to organize a tree structure of subdirectories and rapidly access files named far down the tree.

**LOGICAL NAMES**

VAX/VMS 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 4-1 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.
The System User

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

<table>
<thead>
<tr>
<th>Logical Name</th>
<th>Equivalence Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYS$INPUT</td>
<td>Default input stream for the process. For an interactive user, SYS$INPUT is equated to the terminal. In a command procedure or batch job, SYS$INPUT is equated to the input file or batch input stream.</td>
</tr>
<tr>
<td>SYS$OUTPUT</td>
<td>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.</td>
</tr>
<tr>
<td>Logical Name</td>
<td>Equivalence Name</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SYS$ERROR</td>
<td>Default device to which the system writes error and event messages. For an interac-</td>
</tr>
<tr>
<td></td>
<td>tive user, SYS$ERROR is equated to the terminal. In a batch job, SYS$ERROR is equat-</td>
</tr>
<tr>
<td></td>
<td>ed to the batch job log file.</td>
</tr>
<tr>
<td>SYS$COMMAND</td>
<td>Original SYS$INPUT device for an interactive user or batch job.</td>
</tr>
<tr>
<td>SYS$DISK</td>
<td>Default device established at login, or changed by the SET DEFAULT command.</td>
</tr>
</tbody>
</table>

**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 4-2 illustrates the necessary steps of program development.
The System User

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 4-2  Steps in Program Development**
Creating the Program
The user must create a file to contain the program source statements. The editor is used to create a file during either interactive or batch mode.

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 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.
CHAPTER OVERVIEW
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 and HELP commands.

Topics include:
• Language Name Command Conventions
• Command Procedures
• Terminal Function Keys
• Examples
CHAPTER 5
DIGITAL COMMAND LANGUAGE

INTRODUCTION
The VAX/VMS command language provides users with an extensive set of commands for:

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

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 ...) indicates 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.

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. VAX/VMS also includes commands to log in and log out, 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.

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 system, VAX/VMS provides an extensive HELP facility 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. Finally, logical names can be defined for complex file specifications so that repetitive typing can be avoided.

A single command language—called DIGITAL Command Language—serves both interactive and batch users, and is used in defining command procedures. Any command can be labeled, can be continued on subsequent lines, and can be commented.

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.

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.

**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.

COMMANDS
For the convenience of the user, commands are listed and described below in alphabetical order. Some include detailed examples.

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.

Examples:
1. $ ALLOCATE _DMB2:
   _DMB2: ALLOCATED (system response)
   The ALLOCATE command allocates a specific RK06 disk drive, unit 2 on controller B.
2. $ ALLOCATE MT: TAPE:
   _MTB2: ALLOCATED
   The ALLOCATE command allocates any available tape device and assigns it the logical name TAPE. The ALLOCATE command locates an available tape device and responds with the name of the device allocated. The underscore character (_) preceding the device names in both examples indicates that they are physical device names rather than virtual device names.
**ANALYZE/OBJECT**

**Format:**

```
ANALYZE/OBJECT file-spec
```

**Purpose:**

Provides a description of the contents of an object file or an executable image file. In describing the records, ANALYZE/OBJECT also identifies certain errors. The user is reminded, though, only to assume that he or she has detected and corrected the major errors in linking modules after they have been linked and run successfully. Some forms of linking errors will only be detected during these latter two operations.

The /OBJECT qualifier is required.

**Example:**

```
$ ANALYZE/OBJECT TAXES.EXE OUTPUT=SYS$OUTPUT
```

Analyzes all the records in the executable image file TAXES.EXE (including, by default, the debugger information, end of module, global symbol, module header, traceback and text information, and relocation records). Notice that this logical name for the output file causes the output to appear on the user's terminal. However, the display occurs in total, so there is no halting after each record display to request permission to continue, as there is in the interactive mode.

**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.

**Examples:**

1. `$ APPEND TEST.DAT NEWTEST.DAT`

   The APPEND command appends the contents of the file TEST.DAT from the default disk and directory into the file NEWTEST.DAT.

2. `$ APPEND/NEW/LOG *.TXT MEMO.SUM`

   The APPEND command appends all files with file types of TXT to a file named MEMO.SUM. The /LOG qualifier requests a display of the specifications of each input file appended.
NOTE
The symbol * is a wildcard. It means "take all of this category."

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.

Examples:
1. $ ASSIGN DBA2:[CHARLES] CHARLIE
   $ PRINT CHARLIE:TEST.DAT

   The ASSIGN command associates the logical name CHARLIE with the directory CHARLES on the disk DBA2. Subsequent references to the logical name CHARLIE result in the correspondence between the logical name CHARLIE and the disk and directory specified. Thus, the PRINT command queues a copy of the file DBA2:[CHARLES]TEST.DAT to the system printer.

2. $ ASSIGN DBA1: TEMP:
   $ SHOW LOGICAL TEMP
   TEMP = DBA1: (process)

   The ASSIGN command equates the logical name TEMP to the device DBA1. The SHOW LOGICAL command verifies that the logical name assignment was made.

BASIC
Format:

BASIC

Purpose:
The BASIC command invokes the VAX-11 BASIC compiler to begin a BASIC session. All subsequent command input is read by VAX-11 BASIC.

Example:
1. $ BASIC /LIST/OBJECT/MACHINE_CODE CALCAGE
   Compiles the VAX-11 BASIC program CALCAGE.BAS, producing a listing and an output object module. The listing file is named
CALCAGE.LIS and includes the machine code output. The object module is named CALCAGE.OBJ. By default, integer overflow and bounds checking will occur, line numbers will appear with error messages, traceback will be enabled, and floating point numbers and untyped integers will be 32 bits long.

2. $ BASIC RET
    READY
    The BASIC command without a file specification indicates a terminal session is desired. VAX-11 BASIC responds with a READY message.

BLISS
Format:
    BLISS  file-spec
Invokes the VAX-11 BLISS-32 compiler.
Example:
    BLISS/LIST=PROG 1  PROG
The command compiles a source file PROG.B32 and produces the object module named PROG.OBJ and a listing file PROG1.LIS. Numerous default options, including normal optimization, are selected.

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.

**COBOL/C74**

**Format:**

    COBOL/C74   file-spec

**Purpose:**
Invokes the VAX-11 COBOL-74 compiler to compile a COBOL source program. The /C74 qualifier is required.

**Example:**

    $ COBOL/C74 TRANSLATE/LIST
    $ LINK TRANSLATE, SYS$LIBRARY:/C74/LIBRARY

The COBOL compiler compiles the source program TRANSLATE.COB and creates an object file named TRANSLATE.OBJ and a listing file named TRANSLATE.LIS.

The LINK command specifies the object file, TRANSLATE.OBJ, and the COBOL-74 run-time library, C74LIB, that is located on the default system library device. This library is required to link all VAX-11 COBOL-74 images.

**COBOL/RSX11**

**Format:**

    COBOL/RSX11   file-spec

**Purpose:**
This command invokes the PDP-11 COBOL-74/VAX (compatibility) compiler to compile a COBOL source program. The /RSX11 qualifier is required.

**Examples:**

1. $ COBOL/RSX11 MYFILE

   The COBOL command compiles the source statements in the file MYFILE.CBL and produces an object file named MYFILE.OBJ.
2. **COBOL/RSX11/OBJECT=TEST2/LIST** TEST

   The COBOL command compiles the source statements in the file TEST.CBL and produces an object file named TEST2.OBJ and a listing file named TEST.LST.

**CONTINUE**

Format:

   CONTINUE

Purpose:

The CONTINUE command resumes execution of a 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

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.

**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

Examples:

1. $ COPY TEST.DAT NEWTEST.DAT

   - The COPY command copies the contents of the file TEST.DAT from the default disk and directory into a file named NEWTEST.DAT.
2. COPY *.COM [MALCOLM.TESTFILES]
   The COPY command copies the highest versions of files in the current default directory with a file type of COM to the subdirectory MALCOM.TESTFILES.

**CORAL**

Format:

    CORAL    file-spec

Purpose:

Invokes the PDP-11 CORAL-66/VAX compiler to compile one or more CORAL source programs.

Example:

    $ CORAL/TEST=8      FRED

Compiles all CORAL statements in FRED.COR that are preceded by "TEST X" where X must not exceed eight.

**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.

Examples:

1. $ CREATE A.DAT
   Input line one....
   Input line two....

   \[Z\]
   $

   After the CREATE command is issued from the terminal, the system reads input lines into the sequential file A.DAT until CTRL/Z terminates the input.

2. $ CREATE WEATHER.COM
   $ DECK
   $ FORTRAN WEATHER
   $ LINK WEATHER
   $ RUN WEATHER
$ EOD
$ @WEATHER

This batch job example illustrates using the CREATE command to create a command procedure from data in the input stream. The DECK command is required so that subsequent lines that begin with a dollar sign are not executed as commands, but are accepted as input records. Then, the procedure is executed with the @ (Execute Procedure) command.

3. $ CREATE/DIRECTORY DMA2:[MALCOLM]

The CREATE command creates a directory named MALCOLM on the device DMA2.

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.

Example:

$ DEALLOCATE_DMB1:

The DEALLOCATE command deallocates unit 1 of the RK06 device on controller B. The underscore character in the device name indicates that it is a physical device name; the DEALLOCATE command does not check to see if it is a logical name.

DEASSIGN

Format:

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

Purpose:

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

Example:

$ SHOW LOGICAL TEST_CASES
   TEST_CASES = DBA1:[HARVEY]FILES.DAT (process)
$ DEASSIGN TEST_CASES
$ SHOW LOGICAL TEST_CASES
   No translation for logical name TEST_CASES

The SHOW LOGICAL command displays the current equivalence name for the logical name TEST_CASES. The DEASSIGN command
Digital Command Language

deassigns the equivalence name; the next SHOW LOGICAL command indicates that the name is deassigned.

DEBUG

Format:

   DEBUG

Purpose:

The DEBUG command invokes the VAX-11 Symbolic Debugger 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.

Example:

$ FORTRAN/DEBUG/NOOPTIMIZE WIDGET
$ LINK/DEBUG WIDGET
$ RUN WIDGET

%DEBUG Version 1.0 24 Aug 1978
%DEBUG-I-INITIAL, language is FORTRAN, scope and module set to “WIDGET”
DBG>GO
ENTER NAME:
ENTER NAME:
    Uncontrollable loop
↑Y
$ DEBUG
DBG>

The FORTRAN and LINK commands both specify the /DEBUG qualifier, to compile the program WIDGET with debugger symbol table information and to include the debugger in the image file. The RUN command begins execution of the image WIDGET, which loops uncontrollably. CTRL/Y interrupts the program, and the DEBUG command gives control to the debugger; an interactive debugging session then begins.

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 $; the $ must be in the first character position (column 1) of each input record.

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.

Example:

$ DEFINE  PROCESS_NAME  LIBRA
$ RUN WAKE

The DEFINE command places the logical name PROCESS_NAME in the process logical name table with an equivalence name of LIBRA. The program WAKE can translate the logical name PROCESS_NAME to perform some special action on the process named LIBRA.
DELETE
Format:

    DELETE    file-spec,...

Purpose:

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

If /ENTRY is specified, the DELETE command deletes one or more entries from a printer or batch job queue. If /SYMBOL is specified, the DELETE command deletes a local or global symbol name assignment. The DELETE/ENTRY and DELETE/SYMBOL commands are described under their own headings.

Examples:
1. $ DELETE COMMON.SUM;2
   The DELETE command deletes the file COMMON.SUM;2 from the current default disk and directory.
2. $ DELETE *.OLD;*
   The DELETE command deletes all versions of files with file types of OLD from the default disk directory.

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.

Example:

$ PRINT/HOLD ALPHA.TXT
   Job 110 entered on queue SYS$PRINT

$ DELETE/ENTRY=110    SYS$PRINT

The PRINT command queues a copy of the file ALPHA.TXT in a HOLD status, to defer its printing until a later time. The system displays the job ID, 110, and the name of the queue in which the file was entered. Later, the DELETE/ENTRY command requests that the entry be deleted from the queue SYS$PRINT.
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.

Example:

$ DELETE/Symbol/ALL
The DELETE/Symbol command deletes all symbol definitions at the current command level.

$ DELETE/Symbol/GLOBAL PDEL
The DELETE/Symbol command deletes the symbol name PDEL from the global symbol table for the process.

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.

Example:

$ RUN MYPROG

↑Y
$ EXAMINE 2780
00002780: 1C50B344
DEPOSIT = 0
00002780: 00000000
$ CONTINUE

The RUN command executes the image MYPROG.EXE; subsequently, CTRL/Y interrupts the program. Assuming that the initial defaults of /HEXADECIMAL and /LONGWORD are in effect, the DEPOSIT com-
mand places a longword of 0s in virtual memory location 2780. Note that an eight-place hexadecimal number corresponds to a 32-bit binary number, or longword.

The CONTINUE command resumes execution of the image.

**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.

Examples:

1. `$DIFFERENCES COPYDOC.COM`
   The DIFFERENCES command compares the contents of the two most recent versions of the file COPYDOC.COM in the current default directory. DIFFERENCES compares every character in every record and displays the results on the terminal.

2. `$DIFFERENCES/IGNORE=(COMMENTS, SPACING) COPYDOC.COM`
   The DIFFERENCES commands compares the same files, but ignores all comment lines (that is, all lines beginning with exclamation points), and ignores multiple blanks or tabs in input lines.

**DIRECTORY**

Format:

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

Purpose:

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

Examples:

1. `$DIRECTORY`
   The DIRECTORY command lists all versions of all files in the current default disk and directory.

2. `$DIRECTORY LOGIN.COM;`
   The DIRECTORY command lists all versions of the file LOGIN.COM.
3. $DIRECTORY ALPHA.TXT,BETA,GAMMA

The DIRECTORY command lists the most recent versions of the files ALPHA.TXT, BETA.TXT, and GAMMA.TXT in the current default directory.

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, decimal, hexadecimal, or octal data format. The default format is hexadecimal.

Example:

$ DUMP ORION.EXE

The DUMP command displays the contents of the image ORION.EXE in hexadecimal format, beginning with the first block in the file.

EDIT

Format:

EDIT/editor file-spec

Purpose:

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

- SOS
- SLP
- EDT

Examples:

1. $ EDIT/OUTPUT=TEST.FOR ACCOUNT.FOR/PLINES=10

The EDIT command invokes the default editor, SOS, to edit the file ACCOUNT.FOR. The /PLINES qualifier sets the default number of lines to print with each print command during the editing session.
At the termination of the editing session the changes are written to a file called TEST.FOR.

2. $ EDIT/SLP AVERAGE.FOR

/ $

The command procedure illustrated uses the EDIT/SLP command; all input lines for the SLP editor follow the command in the input stream, terminated by the slash character (/).

**NOTE**
For more detailed information about the VAX/VMS editors SOS, SLP, and EDT, see Chapter 8 of the Handbook.

---

**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.
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

FORTRAN

Format:

    FORTRAN   file-spec,...

Purpose:

The FORTRAN command invokes the VAX-11 FORTRAN compiler to compile one or more source programs.

Example:

1. $ FORTRAN   SCANLINE
    
    The FORTRAN command compiles the program SCANLINE.FOR and produces an object module SCANLINE.OBJ. If this command is executed in a batch job, the compiler also creates a listing file named SCANLINE.LIS.

2. $ FORTRAN   A+B/LIST, C+D/LIST=ALL/OBJECT=ALL
    
    This FORTRAN command requests two separate compilations. For the first compilation, the FORTRAN command concatenates
the files A.FOR and B.FOR to produce an object module named A.OBJ and a listing file named B.LIS.

For the second compilation, the FORTRAN command concatenates the files C.FOR and D.FOR and compiles them to produce an object module name ALL.OBJ and a listing named ALL.LIS.

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.
Examples:

1. \$ HELP ASSIGN
   The HELP command displays an abstract of the ASSIGN command function, its format, and lists the keyword options the user can type to obtain more information.

2. \$ HELP ASSIGN PARAMETERS
   The HELP command displays a description of the parameters for the ASSIGN command.

3. \$ HELP SET TERMINAL
   The HELP command lists information about the TERMINAL keyword option of the SET 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.
If not specified, the INITIALIZE command uses the following default values:

<table>
<thead>
<tr>
<th>Device</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>RK06</td>
<td>2</td>
</tr>
<tr>
<td>RK07</td>
<td>4</td>
</tr>
<tr>
<td>RM03</td>
<td>6</td>
</tr>
<tr>
<td>RP04/5</td>
<td>6</td>
</tr>
<tr>
<td>RP06</td>
<td>11</td>
</tr>
</tbody>
</table>

The default value of structure level 1 disks is always 1.

Example:

```plaintext
$ ALLOCATE DMA2: TEMP
    DMA2: ALLOCATED
$ INITIALIZE TEMP BACK_UP_FILE
$ MOUNT TEMP BACK_UP_FILE
%MOUNT-I-MOUNTED, BACK_UP_FILE mounted on _DMA2
$ CREATE/DIRECTORY TEMP,[ARCHIE]
$ COPY *.* TEMP,[ARCHIE]
```

The above sequence of commands shows how to initialize an RK06/7 volume for backup. First, the device is allocated, to ensure that no one else can access it. Then after the volume is physically mounted on the device, the INITIALIZE command initializes it. When the volume is initialized, the MOUNT command makes the file structure available. Before any files can be placed on the volume, a directory must be created, as shown in the CREATE command example. Finally, the COPY command copies the highest existing versions of files on the default disk to the backup disk.

**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.
Digital Command Language

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
Digital Command Language

provides the compiler with a file name for the object and listing files. The output files are cataloged in the user HENRY's 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.

Examples:

1.  $ LIBRARY/CREATE TESTLIB ERRMSG,STARTUP

   The LIBRARY command creates an object module library named TESTLIB.OLB and places the modules ERRMSG.OBJ and STARTUP.OBJ in the library.

2.  $ LIBRARY/INSERT TESTLIB SCANLINE
    $ LINK TERMTEST,TESTLIB/LIBRARY

   The LIBRARY command adds the module SCANLINE.OBJ to the library TESTLIB.OLB. The library is specified as input to the linker by using the /LIBRARY qualifier on the LINK command. If the module TERMTEST.OBJ refers to any routines or global symbols not defined in TERMTEST, the linker will search the global symbol table of library TESTLIB.OLB to resolve the symbols.

3.  $ LIBRARY/LIST=MYMAC.LIS/FULL MYMAC.MLB

   The LIBRARY command requests a full listing of the macro library MYMAC; the output is written to a file named MYMAC.LIS.

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.
Examples:
1. $ LINK ORION
   The linker links the object module in the file ORION.OBJ and creates an executable image named ORION.EXE.
2. $ LINK/MAP/FULL DRACO,CYGNUS,LYRA
   The linker links the modules DRACO.OBJ, CYGNUS.OBJ, and LYRA.OBJ and creates an executable image named DRACO.EXE.
   The /MAP and /FULL qualifiers request a full map of the image, with descriptions of each program section, lists of global symbols by name and by value, and a summary of the image characteristics. The map file is name DRACO.MAP.

LINK/RSX11
Format:

    LINK/RSX11  file-spec,...

Purpose:

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

Example:

$ LINK/RSX11  AVERAGE
$ RUN  AVERAGE

The object module AVERAGE.OBJ is linked to create the task image named AVERAGE.EXE. The RUN command executes the task.

LOGIN PROCEDURE
Format:

    CTRL/C
    CTRL/Y
    <RET>

Purpose:

There is no LOGIN command. Rather, the user gets the attention of the system and signals an intention to access the system by pressing CTRL/C, CRTL/Y, or carriage return on a terminal not currently in use. The system then prompts for user name and password, and validates them.

Example:

<↑Y>
Username:   HUMPTY
Password:   

84
CTRL/Y gets the attention of the operating system, which immediately prompts for user name. After validating the user name, the system prompts for the password, but does not echo it.

LOGOUT
Format:

    LOGOUT

Purpose:
The LOGOUT command terminates an interactive terminal session.

Example:

$ LOGOUT
    HIGGINS logged out at 6-JUN-1981 17:48:56.73

MACRO
Format:

    MACRO file-spec,...

Purpose:
The MACRO command invokes the VAX-11 MACRO assembler to assemble one or more assembly language source programs.

If /RSX11 is specified, the MACRO command invokes the MACRO-11 assembler; all the other qualifiers apply to both the VAX-11 and the MACRO-11 assembler.

Example:

$ MACRO STAR

The MACRO assembler assembles the file STAR.MAR and creates an object file named STAR.OBJ. If this command is executed in a batch job, the assembler also creates a listing file named STAR.LIS.

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.

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.

Examples:

1. $ MOUNT MT: MATH06 STAT_TAPE
   % MOUNT-I-MOUNTED, MATH06 mounted on _MTA0:
   $ COPY ST061178.DAT STAT_TAPE:

   The MOUNT command requests that the magnetic tape whose volume label is MATH06 be mounted on the device MTA0 and assigns the logical name STAT_TAPE to the volume.

   Subsequently the COPY command copies the disk file ST061178.DAT to the tape.

2. $ MOUNT/FOREIGN MTA1:
   % MOUNT-I-MOUNTED, TESTER mounted on _MTA1:
   $MCR FLX

   The MOUNT command requests the mounting of a foreign volume (i.e., non-Files-11 structured) on the device MTA1. The MCR FLX command invokes the RSX-11M File Transfer Utility to process the volume.

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) encoun-
Digital Command Language

ters 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 CTRL_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, closing 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

87
$ 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.

**PASCAL**

Format:

```
PASCAL file-spec
```

Purpose:

The PASCAL command invokes the VAX-11 PASCAL compiler to compile one or more source programs.

Example:

$ PASCAL/LIST PROGA,PROGB,PROGC

The source files PROGA.PAS, PROGB.PAS, and PROGC.PAS are compiled as separate files. The compiler produces three object files named, by default, PROGA.OBJ, PROGB.OBJ, and PROGC.OBJ. In addition, three listing files are generated that include the warning-level messages but exclude both reference listings and machine object code. The listing files are named PROGA.LIS, PROGB.LIS, and PROGC.LIS, respectively. Furthermore, the other defaults specify that compilation will stop after 30 errors are detected, warning messages will be issued if nonstandard PASCAL statements are found, no runtime checks will be made for illegal subrange values, and only traceback records will be generated.

**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.

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.

Examples:
1. $ PRINT AVERAGE
   Job 236 entered on queue LPA1
   The PRINT command queues the file AVERAGE.LIS for printing on a system printer. The system displays the job ID and the name of the printer to which it queued the file.
2. $ ASSIGN LPA0: SYS$PRINT
   $ PRINT *.TXT
   Job 238 entered on queue LPA0:
   The ASSIGN command creates an entry in the process logical name table for the logical name SYS$PRINT to override the system logical name definition. Subsequently, a PRINT command translates the logical name SYS$PRINT and queues the job on the queue LPA0. This job consists of the highest versions of all files with the file type of TXT.

PURGE
Format:

PURGE file-spec,...

Purpose:
The PURGE command deletes all but the highest numbered version or versions of a specified file or files.
Digital Command Language

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.

Example:

$ RENAME   AVERAGE.OBJ   GINGER

The RENAME command changes the file name of the highest existing version of the file AVERAGE.OBJ to GINGER.OBJ. If no file named AVERAGE.OLD currently exists, the new file is given a version number of 1.

REQUEST
Format:

    REQUEST   message-text

90
Digital Command Language

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.

Example:
$ REQUEST/REPLY "ARE YOU THERE"
%OPCOM-S-OPRNOTIF, operator notified, waiting... 14:54:30.33
↑C
REQUEST-Enter message or cancel request with <↑Z>
REQUEST-Message?↑Z
%OPCOM-S-OPRNOTIF, operator notified, waiting... 14:59:01.38
%OPCOM-F-RQSTSCN, request was cancelled

The REQUEST command issues a message to see if there are any operators. When no response is received, CTRL/C interrupts the request and then CTRL/Z cancels it.

RUN (Image)
Format:

RUN file-spec

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

Example:
$ RUN LIBRA

The image LIBRA.EXE starts executing in the process.

SET
Format:

SET option

where the options are
CARD_READER
[NO]CONTROL_Y
DEFAULT
MAGTAPE
[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.

Example:

$ ALLOCATE CR:
   _CRA0:    ALLOCATED
$ SET CARD_READER CRA0:/029
$ COPY CRA0: [JUANITA.DATAMFILES]CARDS.DAT

The ALLOCATE command requests the allocation of a card reader by specifying the generic device name. When the ALLOCATE command displays the name of the device, the SET CARD_READER command sets the translation mode at 029. Then, the COPY command copies all the cards read into the card reader CRA0 into the file CARDS.DAT in the directory JUANITA.DATAMFILES.

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.

Example:

$ SET NOCONTROL_Y

After this command, the CTRL/Y function is disabled.
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.

Example:

```$ SET DEFAULT [CARPENTER]
$ COPY   A.* B.*
```

The SET DEFAULT command changes the default directory to CARPENTER. The default disk device does not change. The directory name CARPENTER is assumed to be the default directory for subsequent file searches, as in the COPY command shown.

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.

Example:

```$ MOUNT MTB1:/FOREIGN
$ SET MAGTAPE  MTB1:   /DENSITY=800
```

The MOUNT command mounts a foreign tape on the device MTB1. The SET MAGTAPE command defines the density for writing the tape at 800 bpi.

5) **SET ON**

FORMAT:

```
SET [NO]ON
```
Digital Command Language

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.

6) 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.

Example:
$ RUN/PROCESS_NAME=TESTER CALC
%RUN-S-PROC_ID, identification of created process is 00C0012F
$ SET PROCESS TESTER/PRIORITY = 10

The RUN command creates a subprocess and gives it the name TESTER. Subsequently, the SET PROCESS command assigns the subprocess a priority of 10.

7) 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.
Digital Command Language

Example:

$ SET PROTECTION=(GROUP=RWED,WORLD=R)/DEFAULT

This SET PROTECTION command sets the default protection applied to all files subsequently created to allow other users in the same group unlimited access, and all users read access. Default protection for system and owner is not changed. Note that R = read, W = write, E = execute, D = delete.

8) 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.

Example:

$ PRINT/HOLD MYFILE.DAT
   Job 112 entered on queue SYS$PRINT

$ SET QUEUE/ENTRY=112/RELEASE/JOB_COUNT=3

The PRINT command requests that the file MYFILE.DAT be queued to the system printer, but placed in a hold status. The SET QUEUE command releases the file for printing and changes the number of copies of the job to three.

9) SET RMS_DEFAULT

Format:

SET RMS_DEFAULT

Purpose:

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

Example:

$ SET RMS_DEFAULT/DISK/BLOCK_COUNT=16

The SET RMS_DEFAULT command defines the default multiblock count for disk file input/output operations as 16 blocks. This default is
defined only for the current process, and will be used for disk file operations in user programs that do not explicitly set the multiblock count.

10) SET TERMINAL

Format:

    SET TERMINAL [device-name]

Purpose:

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

Example:

$ SET TERMINAL/WIDTH=132/PAGE=66/NOBROADCAST
$ TYPE MEMO.DOC


$ SET TERMINAL/LA36

This SET TERMINAL command indicates that the width of terminal lines is 132 characters and that the size of each page is 66 lines. The /NOBROADCAST qualifier disables the reception of broadcast messages while the terminal is printing the file MEMO.DOC. The next SET TERMINAL command restores the terminal to its default state.

11) 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, includ-
ing command lines, as it reads them. At the end of the procedure, the
SET NOVERIFY command restores the system default.

12) 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.

Example:

$ SHOW WORKING_SET
   Working Set /Limit=100 /Quota=200 /Authorized Quota=200
$ SET WORKING_SET /QUOTA=100
   New Working Set /Limit=100 /Quota=100

The SHOW WORKING_SET command displays the current limit, quota,
and authorized quota. The SET WORKING_SET command sets a
quota limiting the maximum number of pages any image can request.

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.
Example:
$ SHOW DAYTIME
   27-NOV-1981 00:03:45
The SHOW DAYTIME command requests a display of the current date and time.

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.
Example:
$ SHOW DEFAULT
   DBA1:[ALPHA]
$ SET DEFAULT DBA2:[HIGGINS.SOURCES]
$ SHOW DEFAULT
   DBA2:[HIGGINS.SOURCES]
The SHOW DEFAULT command requests a display of the current default device and directory. The SET DEFAULT command changes these defaults, and the next SHOW DEFAULT command displays that the defaults have in fact been changed.
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.

Example:

$ SHOW DEVICES

This command displays, for each device in the system:
- Device name
- Device status (indicates whether the device is on line)
- 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.

Example:

$ SHOW LOGICAL/PROCESS

Contents of process logical name table:

SYS$INPUT = _TTB1:
SYS$OUTPUT = _TTB1:
SYS$ERROR = _TTB1:
SYS$DISK = _DBA3:
SYS$COMMAND = _TTAB1:
The SHOW LOGICAL command requests a display of the current process logical names. These are the default logical name assignments made by the command interpreter for an interactive process.

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.

Example:

$ SHOW MAGTAPE MTA0:

MTA0: UNKNOWN, DENSITY = 800, FORMAT = Normal-11 Odd Parity

The SHOW MAGTAPE command requests a display of the characteristics of the device MTA0. It displays the device type, density, and format (default or normal PDP-11).

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.

Example:

$ SHOW NETWORK

NETWORK STATUS AS OF 16-JUL-1980 12:42
LOCAL NODE NAME: VAX1
NUMBER: 3
STATE: ON

NODE LINE
MANILA XMA0
CHI XMC0

The SHOW NETWORK command displays the name, number, and status of the local node and lists available remote nodes.

If no remote nodes are available, the command displays:

NO REMOTES ACCESSIBLE
If the network is unavailable, the command displays:

NETWORK UNAVAILABLE

7) SHOW PRINTER

Format:

SHOW PRINTER [device-name[:]]

Purpose:
The SHOW PRINTER command displays the default characteristics currently defined for a system printer.

Example:

$ SHOW PRINTER LPA0:
LPA0: LP11, WIDTH=132, PAGE=64, NOCR, FF, LOWERCASE
Device spooled to DBB2:
The SHOW PRINTER command requests a display of the characteristics of the printer LPA0.

8) SHOW PROCESS

Format:

SHOW PROCESS

Purpose:
The SHOW PROCESS command displays information about the current process.

Example:

$ SHOW PROCESS
This command displays the following information about the current process:
• 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.

Example:

$ SHOW PROTECTION
  SYSTEM=RWED, OWNER=RWED, GROUP=RE, WORLD=NO ACCESS
$ SET PROTECTION=(GROUP:RWED, WORLD:RE)/DEFAULT
$ SHOW PROTECTION
  SYSTEM=RWED, OWNER=RWED, GROUP=RWED, WORLD=RE

The SHOW PROTECTION command requests a display of the current protection defaults; the SET PROTECTION command changes the file access allowed to other users in the same group and to miscellaneous system users. The next SHOW PROTECTION command shows the modified protection defaults.

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.

Example:

$ SHOW QUEUE/DEVICES

* DEVICE QUEUE "LPA0:" FORMS=0 GENPRT FLAG
* DEVICE QUEUE "LPB0:" FORMS=0 GENPRT FLAG

  CURRENT JOB 138 CRAMER
  ALPHA, PRI=4, COPIES=1 12-JAN-1978 14:16
  PENDING JOB 139 HIGGINS DOCMASTE, PRI=4, COPIES=1
  12-JAN-1978 15:12
  PENDING JOB 140 HIGGINS DOCRUN3, PRI=4, COPEIS=1 12-JAN-1978 15:13
* DEVICE QUEUE "SYS$PRINT" FORMS=0 GENDEV FLAG

    HOLDING JOB 105 HIGGINS DOCRUN2, PRI=4, COPIES=1, 
    12-JAN-1978 14:16

The SHOW QUEUE command displays the status of the printer 
queues. The first queue, LPA0, has no entries. The second queue, 
LPB0, is currently processing a job for the user CRAMER. Two jobs 
are pending for the user HIGGINS (who issued the command). The 
third queue, named SYS$PRINT, consists of jobs that are being held.

11)    SHOW QUOTA

Format:

    SHOW QUOTA

Purpose:
Displays the current disk quota that is authorized and used by a spe-
cific user on a specific disk.

Example:

$ SHOW QUOTA

This command will display the amount of disk space authorized and 
used on the current default disk for the present user.

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 uses for file 
operations.

Example:

$ SHOW RMS_DEFAULT

The SHOW RMS_DEFAULT command displays the current process 
and system default multiblock and multibuffer counts for all types of 
files.

13)    SHOW STATUS

Format:

    SHOW STATUS

Purpose:
The SHOW STATUS command displays the status of the image cur-
rently executing in the process, if any. The SHOW STATUS command
does not affect the image; execution of the image can be continued after displaying its status.

Example:

$ RUN MYPROG

$ SHOW STATUS

The RUN command executes the image MYPROG.EXE. While the program is running, CTRL/C interrupts it, and the SHOW STATUS command displays its current status:

* 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.

Example:

$ SHOW SYMBOL PRINT
PRINT = PRINT/HOLD

The SHOW SYMBOL command requests that the current value of the symbol name PRINT be displayed. The command interpreter searches the local symbol table for the current command level, then local symbol tables for preceding command levels, then the global symbol table.
15) SHOW SYSTEM

Format:

    SHOW SYSTEM

Purpose:

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

Example:

$ SHOW SYSTEM

The response displays:

- 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.

Example:

$ SHOW TERMINAL

The SHOW TERMINAL command displays the characteristics of the current terminal.
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.

Example:

$ ASSIGN   DBA1: DISK
$ ASSIGN/GROUP   DBB3: DISK
$ SHOW TRANSLATION DISK
   DISK = DBA1: (process)

ASSIGN commands place entries for the logical name DISK in both the process and group logical name tables. The SHOW TRANSLATION command shows the logical name for the first entry it finds: the equivalence name placed in the process logical name table.

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.

Example:

$ SHOW WORKING_SET
Working Set   /Limit=100/ Quota=200 /Authorized Quota=200

The response to this command indicates that the current process has a working set limit of 100 pages, a quota of 200 pages, and that the current quota is equal to the authorized limit (200 pages).

SORT

Format:

SORT   input-file-spec   output-file-spec

Purpose:

Invokes the VAX-11 SORT 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
Examples:
1. $ RUN MYPROG
   ...
   \n   ↑Y
   $ STOP
   The RUN command begins executing the image MYPROG. Sub-
sequently, CTRL/Y interrupts the execution and the STOP com-
mand terminates the image.
2. $ 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 lev-
els: if this ON command is executed in a command procedure that
is executed from within another procedure, control does not re-
turn 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.
Example:
$ SUBMIT AVERAGE
Job 112 entered on queue SYS$BATCH
The SUBMIT command enters the procedure AVERAGE.COM in the batch job queue. When the batch job completes, the log file AVERAGE.LOG is queued for printing.

**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.

Example:

```
TYPE COMMON.DAT
```

The TYPE command requests that the file COMMON.DAT be displayed at the terminal.

**UNLOCK**

Format:

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

Purpose:
The UNLOCK command makes accessible a file that became inaccessible as a result of being improperly closed.
Example:

$ TYPE TESTFILE.OUT
%TYPE-E-OPENIN, error opening DBA1:[MALCOLM]TESTFILE.OUT;
  3 as input
-SYSTEM-W-FILELOCKED, file is deaccess locked
$ UNLOCK TESTFILE.OUT
$ TYPE TESTFILE.OUT

The request to type the output file, TESTFILE.OUT, returns an error message that indicates the file is locked; the UNLOCK command unlocks it. The TYPE command is used to verify the contents of the file, which may be incomplete.

**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 STOP command or another DCL 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.

**TERMINAL FUNCTION KEYS**

- **<CR> or RETURN**  Carriage return. Transmits the current line to the system for processing.
- **CTRL/X**  Provides first part of 2-character sequence of functions.
- **CTRL/C**  Before terminal session, initiates log-in 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.
- **CTRT/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.
CHAPTER OVERVIEW
The large collection of language processors is described. 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:
• Assembly Language
• High Level Native Mode Languages
• Compatibility Mode Languages
CHAPTER 6
PROGRAMMING LANGUAGES

INTRODUCTION
VAX/VMS provides a complete program development environment. In addition to the native mode assembly language, MACRO, it offers the optional higher level languages commonly needed in engineering and scientific, commercial, instructional, and implementation applications—FORTRAN, COBOL, BASIC, PASCAL, BLISS, CORAL, and PL/I. VAX/VMS 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.

VAX-11 includes two programming environments:

- Native mode
- Compatibility mode

Native mode language processors produce native object code, and take advantage of the native instruction set and 32-bit architecture of the VAX hardware. VAX-11 MACRO, VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BLISS-32, VAX-11 PASCAL, VAX-11 PL/I, and VAX-11 BASIC are compilers and assemblers that produce native mode object code.

The compatibility mode programming environment consists of the compilers for PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN, MACRO-11, and PDP-11 CORAL 66/VAX. These produce compatibility mode object code.

VAX-11 MACRO
The VAX-11 MACRO assembler accepts one or more source modules written in MACRO assembly language and produces a relocatable object module and symbol table and optional assembly listing. VAX-11 MACRO is similar to PDP-11 MACRO, but its instruction mnemonics correspond to the VAX-11/780 native instructions. VAX-11 MACRO 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
Programming Languages

- User-defined macros
- Multiple macro libraries with fast access structure
- Program sectioning directives
- Conditional assembly directives
- Assembly and listing control functions
- Alphabetized, formatted symbol table listing
- Default error listing on command output device
- A 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-11 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-11 instruction mnemonic, or an assembler directive. MACRO includes directives for:
- Listing control
- Functional specification
• Data storage allocation
• 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

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 switch options included in the listing file specification in the command string issued to the MACRO assembler. The use of these switch options 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. MACRO 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, re-entrant programs.

VAX-11 FORTRAN
VAX-11 FORTRAN is an optional native mode language processing system whose 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-
Programming Languages

language level. At the same time, it provides optional 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-11 native object code
- Makes use of the VAX-11 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. VAX-11 FORTRAN is able to create, read, and write indexed (and relative) files. In addition, FORTRAN is able to reference a relative or indexed file already created by another language (for instance, COBOL), 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 11 I/O Services.

Simplified I/O Formats
List-directed input and output statements provide a method for obtaining simple sequential formatted input or output without the need for FORMAT statements. On 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.

Character Data Type
A program can create fixed-length CHARACTER variables and arrays to store ASCII character strings. The VAX-11 FORTRAN language pro-
vides 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. VAX/VMS provides a set of character manipulation instructions 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.

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 shared programs. FORTRAN subprograms can also be used to create shareable image libraries, which can be available to 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.

**Debugging FORTRAN Programs**
VAX-11 FORTRAN 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 call routines, to set break or trace points, to set default modes, to define symbols, and to deposit, examine, or evaluate virtual memory locations.

**Conditional Compilation of Statements**
During the development stages of a program, it is often useful to establish points in the program at which specified values can be examined to insure that the program is functioning correctly. For example, if the value of a variable is known after the execution of a specified statement, the variable can be printed to verify its contents. Therefore, by including a number of such source lines at strategic points throughout the program, debugging the program is greatly simplified.

**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.

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.
• 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
Programming Languages

comments. In the first category are assignment, control, I/O, format, and specification statements.

Following are three tables: Table 6-1 is a brief summary of FORTRAN-77; Table 6-2 is a summary of VAX-11 FORTRAN extensions to the ANSI standard. And Table 6-3 is a summary of traditional FORTRAN IV (industry-compatible) features supported by VAX-11 FORTRAN.

Table 6-1  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
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
### Table 6-2  VAX-11 FORTRAN Extensions

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thirty-one-character symbolic names</td>
<td>Permit interfacing to VAX/VMS system service procedures using the VAX-11 calling standards.</td>
</tr>
<tr>
<td>CALL extensions</td>
<td></td>
</tr>
<tr>
<td>Hexadecimal and octal constants and field descriptors</td>
<td>Structured looping control constructs</td>
</tr>
<tr>
<td>DO WHILE</td>
<td></td>
</tr>
<tr>
<td>END DO</td>
<td></td>
</tr>
<tr>
<td>Additional data types and type declaration statements</td>
<td>BYTE, LOGICAL<em>1, LOGICAL</em>2, LOGICAL, LOGICAL<em>4, INTEGER</em>2, INTEGER, INTEGER<em>4, REAL, REAL</em>4, DOUBLE PRECISION, REAL<em>8, COMPLEX, COMPLEX</em>8, DOUBLE COMPLEX, COMPLEX*</td>
</tr>
<tr>
<td>CHARACTER*n</td>
<td><strong>NOTE</strong> Names on the same line above are synonyms. Those in boldface are the ANSI standard ones.</td>
</tr>
</tbody>
</table>

Indexed File I/O  
keyed READ  
Key types: INTEGER*2, INTEGER*4, CHARACTER with generic, and approximate key match
Programming Languages

indexed file WRITE
REWRITE statement
DELETE statement
UNLOCK statement

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 6-3 Traditional FORTRAN IV (Industry-Compatible)

Features

FORTRAN IV compatible
direct access I/O:
   DEFINE FILE
   READ (u'r)
   WRITE (u'r)
   FIND (u'r)
   ENCODE Statement
   DECODE Statement
Hollerith processing of
character data
   (optional) one-trip DO
   loops instead of
   FORTRAN-77 zero-trip DO
   loops
Device-oriented I/O
Statements:
   TYPE
   ACCEPT
   PRINT

u = logical unit #
r = record #
VAX-11 COBOL-74
VAX-11 COBOL-74 is an optional language processing system for business data processing requirements. It conforms to ANSI-74 COBOL, the industry-wide, accepted standard for the COBOL language; it is also highly upwardly compatible with PDP-11 COBOL, for ease of program transport. VAX-11 COBOL-74 produces native-mode object code, taking advantage of the packed decimal and character string instruction set of the VAX-11 and the virtual memory system provided by VAX/VMS.

VAX-11 COBOL-74 includes the following language elements:
- Level 2 Nucleus module
- Level 2 Table Handling module
- Level 2 Sequential I/O module
- Level 2 Relative I/O module
- Level 2 Indexed I/O module
- Level 2 Segmentation module
- Level 1 Library module, with partial Level 2 REPLACING facility
- Level 1 Interprogram Communication module
- Cross Reference Compilation Listing
- DISPLAY verb and WITH NO ADVANCING clause
- Conditional variables—Data Division level 88
- Nested conditionals
- Debugging with the symbolic debugger

File Organizations
The Sequential I/O, Relative I/O, and Indexed I/O modules meet the full ANSI-74 Level 2 standards and include the following COBOL features:
- OPEN EXTEND—Add records to a previously created sequential file without recopying the file
- DYNAMIC ACCESS—Process a relative file or indexed file both randomly and sequentially in the same program
- START—Select positioning within a relative file for subsequent record retrieval
- REWRITE/WRITE—Logically replace a record in a mass storage file and generate a new record
- CLOSE LOCK—Protect a file from being opened by the current program a second time
- LINAGE—Specify logical page format
The Level 2 Indexed I/O module statements enable COBOL programs to use the multi-key indexed record management services to process indexed files. Indexed files can be accessed sequentially, randomly, or dynamically using one or more keys to select records. The Environment Division RESERVE clause enables the user to specify the number of buffer areas for fast multi-key processing.

**Data Types**
VAX-11 COBOL-74 supports a variety of data types, including:

- Numeric DISPLAY Data
  - Trailing overpunch sign
  - Leading overpunch sign
  - Trailing separate sign
  - Leading separate sign
  - Unsigned
  - Numeric-edited
- Numeric COMPUTATIONAL Data
  - 1-word fixed binary
  - 2-word fixed binary
  - 4-word fixed binary
- Alphanumeric DISPLAY Data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric-edited
- Packed Decimal Data

**String Manipulation**
COBOL has the capability to manipulate data strings. It offers the INSPECT, STRING, and UNSTRING verbs that search for embedded character strings, with tally and replace. In addition, it is possible to join together or break out separate strings with various delimiters.

**Interactive COBOL Programs**
The Procedure Division ACCEPT and DISPLAY statements allow terminal-oriented interaction between a COBOL program and a user. This is useful, for example, in an order entry application.

The ACCEPT statement allows the terminal user to enter input lines which the COBOL program can interpret. The DISPLAY statement transfers a message to a specified device, normally the user's terminal. The statement can be modified by a special WITH NO ADVANCING clause (without automatic appending of carriage return and line
Programming Languages

feed) that allows the COBOL program to control the format of the message sent. This is especially useful when typing prompting messages on the console.

While the ACCEPT and DISPLAY statements are intended primarily for use with keyboard devices, COBOL allows the ACCEPT statement to accept input from a card reader or the batch input stream, and the DISPLAY statement to display data on a line printer.

Sample Cobol-74 Code
This sample VAX-11 COBOL-74 code demonstrates some of the powerful language elements of VAX-11 COBOL-74. It illustrates an interactive COBOL program which will generate various types of reports depending upon user specified options. The program operates on an indexed information file via the dynamic access mode. Illustrated are three major COBOL verbs: ACCEPT, DISPLAY and INSPECT.

In Figure 6-1, the program describes the file organization and the access mode. Also described are the primary and alternate keys used for accessing the file randomly.

```
****** INPUT-OUTPUT SECTION *************
FILE=CONTROL,

SELECT CUSTOMER=FILE
  ASSIGN TO "CUSTOMER.DAT"
  ORGANIZATION IS INDEXED
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS CUST=CUST=NUMBER
  ALTERNATE RECORD KEY IS CUST=CUSTOMER=NAME
  FILE STATUS IS CUSTOMER=FILE=STATUS.

SELECT STATEMENT=REPORT
  ASSIGN TO "STATEMENT.REP"
  FILE STATUS IS STATEMENT=REPORT=STATUS.

****** File Description ****************************
```

Figure 6-2 describes an options area in the working storage section. The programmer uses "88" level numbers allowing reference to coded information via symbolic names.

127
Programming Languages

01 OPTIONS-AREA.
03 OPTIONS-AREA-CHAR OCCURS 30 PIC X(1).

01 A-COUNT PIC 9(2).

01 OPTION-STORAGE.
03 OPTION-ENTRY OCCURS 8 PIC 9(1).
01 OPTION-VALUES REDEFINES OPTION-STORAGE.
03 FILLER PIC 9(1).
   08 WANT-STATEMENTS VALUE 1 THRU 9.
03 FILLER PIC 9(1).
   08 WANT-INVOICES VALUE 1 THRU 9.
03 FILLER PIC 9(1).
   08 WANT-ALL-CATALOGS VALUE 1 THRU 9.
03 FILLER PIC 9(1).
   08 WANT-SOME-CATALOGS VALUE 1 THRU 9.
03 FILLER PIC 9(1).
   08 WANT-CREDIT-LIMIT-LETTERS VALUE 1 THRU 9.
03 FILLER PIC X(3).

01 RECORD-COUNT PIC 9(5) VALUE 0.
01 STATEMENT-COUNT PIC 9(5) VALUE 0.
01 INVOICE-COUNT PIC 9(5) VALUE 0.
01 CREDIT-LIMIT-COUNT PIC 9(5) VALUE 0.
01 CATALOG-COUNT PIC 9(5) VALUE 0.

*****************************************************

Figure 6-2  Working Storage Area
DISPLAY "ENTER OPTIONS:"
DISPLAY "S = Print statements",
DISPLAY "I = Print invoices",
DISPLAY "CA = Mail all catalogs",
DISPLAY "CO = Mail selective catalogs",
DISPLAY "CL = Credit limit letters",
ACCEPT OPTIONS-AREA.
MOVE ALL ZERO TO OPTION-STOREAGE.
IF OPTIONS-AREA = SPACES
   DISPLAY "Discrepancy Report Only"
   GO TO CONFIRM-OPTIONS.
   MOVE 1 TO A-COUNT.
INSPECT OPTIONS-AREA TALLYING
   OPTION-ENTRY (1) FOR ALL "S"
   OPTION-ENTRY (2) FOR ALL "I"
   OPTION-ENTRY (3) FOR ALL "CA"
   OPTION-ENTRY (4) FOR ALL "CO"
   OPTION-ENTRY (5) FOR ALL "CL".

IF OPTION-STOREAGE = ALL ZERO
   DISPLAY "No options recognized"
   STOP RUN.
DISPLAY "Selected options:"
IF WANT=STATEMENTS
   DISPLAY " Statements"
IF WANT=INVOICES
   DISPLAY " Invoices"
IF WANT=ALL-CATALOGS
   DISPLAY " All catalogs"
IF WANT=SOME-CATALOGS
   DISPLAY " Selected catalogs"
IF WANT=CREDIT-LIMIT-LETTERS
   DISPLAY " Credit limit letters".

Figure 6-3  Procedure Division Using Interactive COBOL Verbs

In Figure 6-3, using the DISPLAY verb, the interactive COBOL program requests the user to specify an options selection. The user response is then transmitted to the program via the ACCEPT verb. The program uses the INSPECT verb to check that a valid response has been received.
OPEN INPUT CUSTOMER=FILE.
MOVE "000000" TO CUST=CUST-NUMBER.
START CUSTOMER=FILE
    KEY IS CUST=CUST-NUMBER.
OPEN OUTPUT STATEMENT=REPORT.

*************************************************************

MAINLINE SECTION.
BEGIN.
READ CUSTOMER=FILE NEXT
    AT END
        GO TO END-PROCESS.
    ADD 1 TO RECORD-COUNT.
* *
Print statement if required.
*

Figure 6-4  Random to Sequential Access

Figure 6-4 illustrates the dynamic access method, i.e., shift from random to sequential access. The user moves zero to the primary record key, searches the file randomly, and commences sequential processing at the first non-zero number.

Source Library Facility
With COBOL, the user has a full ANSI-74 Level 1 Library facility, plus high-level extensions. 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 to reduce program preparation time and eliminate a common source of errors.

CALL Facility
VAX-11 COBOL-74 supports the CALL statement, allowing COBOL programs to invoke separately compiled subprograms and to pass arguments between them. These subprograms may be written in COBOL or in another VAX-11 supported language, and may include system service routines written in VAX-11 MACRO. To facilitate calls between programs written in different languages, VAX-11 COBOL-74 provides an extended call facility which allows arguments to be passed by value, reference, or descriptor; this is in contrast to standard COBOL in which arguments are passed by reference only. Figure 6-5 shows a sample program which utilizes all three types of argument passing mechanisms.
Programming Languages

CMD: CALLST2, CALLTST2/MAP/CHEF=CALLTST2
IDENTI 243147

IDENTIFICATION DIVISION.
PROGRAM-ID. CALLST2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE=COMPUTER. PDP=11.
OBJECT=COMPUTER. PDP=11.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TIMLEN PIC 9(4) USAGE IS COMP VALUE IS 0.
01 D-TIMLEN PIC 9(4) VALUE IS 9999.
01 TIMBUF PIC X(24) VALUE IS SPACES.
01 DUMMY PIC 9(5) USAGE IS COMP VALUE IS 0.
01 RETURN-VALUE PIC 9(9) USAGE IS COMP VALUE IS 999999999.
01 D-RETURN-VALUE PIC 9(9) VALUE IS 999999999.
PROCEDURE DIVISION.
00.
DISPLAY "CALL SYSS$ASCTIM".
CALL "SYSS$ASCTIM" USING BY REFERENCE TIMLEN
BY DESCRIPTOR TIMBUF
BY VALUE DUMMY
BY VALUE DUMMY
GIVING RETURN-VALUE.
DISPLAY "DATE/TIME = " TIMBUF.
MOVE TIMLEN TO D-TIMLEN.
DISPLAY "LENGTH OF RETURNED = " D-TIMLEN.
MOVE RETURN-VALUE TO D-RETURN-VALUE.
DISPLAY "RETURN VALUE = " D-RETURN-VALUE.
STOP RUN.

Figure 6-5  System Services Call

In this program the system service routine $ASCTIM is called, which converts binary time to an ASCII string representation. In this example, the buffer length as specified by “timbuf” plus the value of the item “dummy” determine the type of information which the service routine will return to the COBOL program (e.g., specifying a length of 24 plus values of 0 in the following two arguments will cause both current date and time to be returned; if a length of 11 had been specified, then only the date would be returned).

External Subprograms
The CALL statement enables a COBOL program to execute routines that are external to the object module in which the CALL statement appears. The VAX-11 COBOL-74 compiler produces an object module from a single source module supplied as input. The object module file can be linked with other COBOL object modules, or with object modules created by other VAX-11 compilers that produce native mode code to produce an executable image.
Debugging COBOL Programs
To make program debugging easier, the COBOL compiler produces
source language listings with embedded diagnostics. Fully descriptive
diagnostic messages are listed at the point of error. Over 500 different
error conditions are checked, varying from simple warnings to fatal
error detections.

When the compiler detects an error in the source program, the com-
piler attempts to recover from the error and to continue to compile the
program. The kind of error message, whether informational, warning,
or fatal, indicates the likelihood that the assumption made to recover
from the error will produce an object program that will run as the
programmer intended. The compiler will not produce object code if
fatal errors are detected at compile time.

Debugging large source programs is made still easier by using the
optional allocation maps for the Data and Procedure Divisions, exter-
nal subprogram references, and object library references. The comp-
iler can also produce a cross-reference listing.

When a fatal error occurs at runtime, an error message identifying the
cause of the error is displayed to the user. Additionally, a VAX/VMS
system facility called traceback prints the sequence of error. For each
call frame, traceback displays the module name, routine name, source
line number, and program-counter information. This information iden-
tifies in source language terms the module, routine, and source line on
which the fatal error occurs.

Figure 6-6 illustrates the printing of an error message and the
subsequent traceback for a COBOL module in which a subscript viola-
tion occurs at runtime. The “module name” and “routine name” fields
identify the entry point, SUBERRST, into the COBOL module. The
subscript violation occurs on line number 15 of the source module.
The “relative PC” field specifies that the subscript violation corre-
pondingly occurs at “3C” hexadecimal bytes into the object code rela-
tive to the entry point SUBERRST. The “absolute PC” field also speci-
fies that the violation occurs at absolute location “3074” in the
executable image containing SUBERRST. Thus, the issuance of a
specific, English-like error message coupled with the traceback facility
offers the user a powerful debugging tool in identifying programming
errors.
Programming Languages

%C74-F-SUBOUTTRAN, subscript out of range
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name  routine name  line  relative PC  absolute PC
SUBERRSTST  SUBERRSTST  15   0000003C   00003074

Figure 6-6 Example of Traceback Facility

Additionally, programs compiled with the VAX-11 COBOL-74 compiler may be debugged using the VAX/VMS symbolic debugging facility. Debugging capabilities include: setting breakpoints, examining and modifying the values of variables during dynamic execution of programs, and altering the flow of control of the executing COBOL program.

Source Program Formats
The COBOL compiler accepts source programs that are coded using either the conventional 80-column card reference format or a shorter, easy-to-enter terminal format. Terminal format is designed for use with the interactive text editors. It eliminates the line-number and identification fields and allows the user to enter horizontal tab characters and short lines.

The RFRMT (Reformat) utility program reads COBOL source programs that were coded using terminal format and converts the source statements to the 80-column source line format accepted by other COBOL compilers throughout the industry. The programmer can enter source programs in the simpler terminal format and then, if compatibility is ever required with other systems, run the RFRMT utility to convert the source to conventional format.

VAX-11 COBOL
Introduction
VAX-11 COBOL is a new, 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. Some features planned for the next COBOL (anticipated in 1981), are also included. VAX-11 COBOL expands and enhances VAX-11 COBOL-74 described earlier in this chapter, and includes features that appeal to a wider range of COBOL users because it allows more complex coding procedures to be accomplished more simply.

It is anticipated that the new ANSI standard will call for greater structured programming. This allows explicit delimiting of statements in the
Programming Languages

Procedure Division, a feature which can simplify COBOL coding that previously required additional GO TO statements and procedure names. In meeting the requirement for structured programming, the new VAX-11 COBOL includes—among other features—the in-line PERFORM statement, allowing a reduction of program complexity by putting all the logic of the PERFORM in line.

VAX-11 COBOL is considerably faster than its predecessor, with improvements in both compile and execute speeds. Another advantage of the VAX-11 COBOL compiler is its ability to support much larger programs.

Many features of VAX-11 COBOL make the programmer's job easier, either by simplifying coding procedures or by giving direct access to more VAX/VMS facilities. The COBOL SORT and MERGE verbs are now available in VAX-11 COBOL so that sorting and merging can be performed at the source language level rather than though direct calls to the VAX/VMS utilities. VAX-11 COBOL supports symbolic characters so that the programmer can define non-printable characters simply and can generate video display forms. Further, the REFORMAT utility allows bi-directional conversion of COBOL source programs—from easy-to-enter DIGITAL terminal format to ANSI standard format and vice versa.

VAX-11 COBOL is properly defined as an implementation of ANSI COBOL with full support of the following:

- Full Level 2 Nucleus Module without the RERUN option in the I-O-CONTROL paragraph
- Full Level 2 Table Handling Module
- Full Level 2 Sequential I/O Module
- Full Level 2 Relative I/O Module
- Full Level 2 Indexed I/O Module
- Full Level 2 Segmentation Module
- Full Level 2 SORT/MERGE Module
- Full Level 2 Library Module
- Full Level 2 Inter-program Communication Module

Besides the VAX-11 object module, the compiler is capable of producing a machine language listing, a cross reference listing in either alphabetic sequence or order of declaration, and maps of file names, data names, procedure names, and external program names.

General Characteristics
Most of the code in an object module is implemented with in-line, VAX-11 instructions. The object code produced by the compiler takes
advantage of such native mode features as:

- Direct calls to the operating system
- Transparent access to DECnet
- Direct calls to VAX-11 SORT
- Many of the VAX-11 string manipulation instructions
- Direct calls to the Common Run Time Library
- Direct calls to an external routine (written in a DIGITAL-supported language) that conforms to the VAX-11 Procedure Calling Standard

The object code produced by VAX-11 COBOL uses the VAX/VMS traceback facility for determining the source of run time errors. If a fatal error occurs at run time, an English error message is printed to identify the cause of the error. Additionally, the traceback pinpoints the source of the error to a specific line number in the COBOL source module producing the error. The English error message coupled with the traceback facility gives the user a powerful debugging tool for identifying fatal execution errors.

Object modules produced by the compiler can be linked with native mode object modules produced by other VAX-11 language processors including BASIC, FORTRAN, and MACRO.

Structured Programming

Structured programming adds some of the features of a block-structured language (such as ALGOL) to the new VAX-11 COBOL compiler. Thus, more complex programs can be written in-line without recourse to subroutines. This makes programs easier to write and to read.

The example below shows the READ and IF statements using structured programming. The statements after END-READ are executed regardless of whether the AT END condition occurs. Similarly, the MOVE after END-IF is executed regardless of the value of FILE-END.

```
IF ITEMA = ITEMB
    READ FILE-A AT END
    MOVE 1 TO FILE-END
    CLOSE FILE-A
    END-READ
MOVE ITEMB TO ITEMC
IF FILE-END = 1
    DISPLAY ITEMC
    END-IF
MOVE ITEMD TO ITEME.
```

Several COBOL verbs have structured programming delimiters. Among them are:
ADD
CALL
COMPUTE
DELETE
DIVIDE
IF
MULTIPLY
PERFORM
READ
RETURN
REWRITE
SEARCH
START
STRING
SUBTRACT
UNSTRING
WRITE

Particularly, the PERFORM verb has been enhanced. The resultant in-line PERFORM capability is similar to DO WHILE and DO UNTIL in other high level languages.

In this example, if the first occurrence of ITEMB is not equal to "X": (1) the in-line PERFORM statements are executed, moving an "X" to the first 10 occurrences of ITEMB; then, (2) the message is displayed.

IF ITEMB (1) NOT = "X"
PERFORM
   VARYING ITEMA FROM 1 BY 1
       UNTIL ITEMA > 10
       MOVE "X" TO ITEMB (ITEMA)
END-PERFORM
DISPLAY "ARRAY INITIALIZED"

Data Types
VAX-11 COBOL increases the number of data types available to the COBOL programmer, including floating point and double floating point. The complete variety of standard data types looks like this:

- Numeric DISPLAY Data
- Trailing overpunch sign
- Leading overpunch sign
- Trailing separate sign
- Leading separate sign
- Unsigned
- Numeric-edited
Programming Languages

- Numeric COMPUTATIONAL Data
  - 1-word fixed binary
  - 1-longword fixed binary
  - 1-quadword fixed binary

- Packed-Decimal Data (COMPUTATIONAL-3)
  - Unsigned packed decimal
  - Signed packed decimal

- Floating Point Data
  - Floating (COMPUTATIONAL-1)
  - Double Floating (COMPUTATIONAL-2)

- Alphanumeric DISPLAY Data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric-edited

Files and Records
VAX-11 COBOL’s 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 using one or more indexed keys to select records. The RESERVE AREAS clause enables the user to specify the number of I/O buffers for fast multikey processing. The APPLY clause allows the user to specify file processing optimization attributes for fast record access.

VAX-11 COBOL has full variable-length record capability. This is an improvement over VAX-11 COBOL-74, in which variable-length records were only partially supported.

Reference modification—the ability to refer to parts of defined fields without redefining them—has also been included in VAX-11 COBOL.

The language includes a facility to manipulate data strings. The INSPECT verb allows the user to search for embedded character strings, tallying and/or replacing the occurrences of such strings. Additionally, the STRING and UNSTRING verbs permit the user to join together and break out separate strings with various delimiters.
Programming Languages

The VAX-11 COBOL SORT/MERGE module meets the full ANSI standard and permits performing sort and merge operations at the COBOL 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 key(s) (in ascending or descending order) for each file, and the option to use either standard or user-specified input/output procedures.

External Subprograms
The CALL statement enables a COBOL programmer to execute routines that are external to the source module in which the CALL statement appears. The VAX-11 COBOL compiler produces an object module from a single source module. The object module file can be linked with other VAX-11 object modules, so as to produce an executable image. Thus, COBOL programs can call external routines written in other VAX-11 supported languages including BASIC, FORTRAN, and MACRO.

The CALL statement facility has been extended by allowing the user to pass arguments BY REFERENCE (the default in COBOL), BY DESCRIPTOR, and BY VALUE. These argument-passing mechanisms conform to the VAX-11 Procedure Calling Standard and allow COBOL programs to call VAX/VMS operating system service routines. Also, a COBOL program can receive a returned status value from the routine it calls via the GIVING clause associated with the extended CALL facility. Such an extended CALL facility gives the user access to operating system specific facilities and Common Run Time facilities.

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 (i.e., replacement) in the process. This capability reduces program preparation time and eliminates a common source of error during program development.

Shareable Programs
The COBOL language can be used to create shareable programs. VAX-11 COBOL subprograms can be placed in shareable image libraries created by the linker, which then can be made available to any program written in a native programming language.
Debugging COBOL Programs
The VAX-11 COBOL 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 listing, a file name map, a data name map, a procedure name map, an external program name map, and a cross reference listing.

When a fatal error occurs at run time, an error message identifying the cause of the error is displayed to the user. Additionally, the traceback system 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.

Additionally, VAX-11 COBOL debugging facilities provide access to the VAX/VMS DEBUG program. The DEBUG program lets the programmer set breakpoints, and examine and modify the contents of locations dynamically when the COBOL program is executing.

Source Translator Utility
The source 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 which could not be mechanically translated and which therefore require further investigation by the programmer.

Some of the difference between VAX-11 COBOL and PDP-11 COBOL or VAX-11 COBOL-74 that require such a translator are:
- Some changes in file status codes
- Different specification for the storage of intermediate results
- Different methods of specifying file optimization attributes.

Fortunately, most differences are transparent to the programmer, and moving programs from PDP-11 COBOL or VAX COBOL requires little (in some cases, no) programmer work.

Source Program Formats
The VAX-11 COBOL compiler accepts source programs that are coded using either the ANSI standard (conventional) format or a shorter, easy-to-enter DIGITAL terminal format. Terminal format is designed for use with the interactive text editors. It eliminates the line number
and identification 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 advantage of saving disk space and compile-time processing when a user is initially migrating from a non-DIGITAL COBOL system to VAX-11 COBOL.

Additional Features
Some additional features of the VAX-11 COBOL compiler are:

1. Subscripts can be arithmetic expressions.
2. Subscripting and indexing are interchangeable.
3. The CONTINUE statement is included. It transfers control to the next executable statement and can replace conditional or imperative statements.
4. The AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs are included.
5. INITIAL clause on the Program-ID is included.
6. User-defined alphabets are included.
7. PADDING CHARACTER is supported in the FILE-CONTROL paragraph.
8. VALUE OF clause is included.
9. Delimited scope statements are included (e.g., END-ADD, END-IF).
10. All arithmetic statements with overlapping operands function as if the operands did not overlap except for operands specified in LINKAGE SECTION or as EXTERNAL.
11. ALTER statement is included.
12. CALL data-name is included. Both ON OVERFLOW and EXCEPTION are supported.
13. CANCEL statement is fully implemented.
14. INITIALIZE statement is fully implemented.
15. INSPECT statement is fully implemented including combined TALLTING and REPLACING format.
16. SET statement supporting mnemonic-names and condition-names is included.
17. Independent segments (segments 50 and above) of the SEGMENTATION module are included.
Programming Languages

18. WRITE advancing mnemonic-name and associated SPECIAL NAMES C01 is included.
19. Use of source file libraries by the COPY statement is included.

This powerful, flexible, and easy-to-use compiler is layered with the VAX/VMS operating system and is available to those customers who require COBOL with VAX/VMS, V2.0.

VAX-11 BLISS-32
BLISS-32 is a high level systems implementation language for VAX-11. It is specifically designed for building software such as compilers, real-time processors, and operating systems modules. (For example, the VAX-11 FORTRAN compiler is coded in BLISS.) It contains many of the features of high-level languages (e.g., DO loops, IF-THEN-ELSE statements, automatic stack and register allocations, and mechanisms for defining and calling routines) but also provides the flexibility, efficiency, and access to hardware which one would expect from an assembly language. The BLISS compiler produces highly-optimized object code which is typically within 5% to 10% of the efficiency of code produced by an experienced assembly language programmer. The VAX-11 BLISS-32 compiler is written entirely in BLISS and runs in native mode under the VAX/VMS operating system.

Features of BLISS-32
VAX-11 BLISS-32 has several characteristics which set it apart from other high-level languages:

• Data—BLISS-32 is "type-free": all data are manipulated as values from 1 to 32 bits in length. The interpretation of any data item is provided by the operator applied to it. A value can be fetched from or assigned to any contiguous field of from 1 to 32 bits located anywhere in the VAX-11 virtual address space. The expression Y <4,4> = 0 deposits zero into bits 4 through 7 of location Y. This BLISS expression generates the single VAX-11 instruction: BICB2 # ↑XF0,Y.

• Value Assignments—all names in BLISS-32 represent addresses. Contents of storage locations are accessed by means of a fetch operator (.). Hence, the expression X = .Y+3 is interpreted as adding 3 to the contents of location Y, then assigning the result to the storage location beginning at X.

• Expressions—BLISS-32 permits construction of complex expressions in which several different kinds of operations can be performed in a single program statement. For example, the expression 2*(B=.C+1) calculates 2*(.C+1) and simultaneously assigns the value of .C+1 to B.
• Structures—BLISS-32 defines such data structures as VECTOR, BLOCK, BITVECTOR, and BLOCKVECTOR. In addition, the programmer can define arbitrary data structures specifically designed for a given application.

Other BLISS-32 features include:
• CASE, SELECT, SELECTONE, and IF-THEN-ELSE—providing for sequencing of instructions based on evaluation of expressions at run-time.
• DO, WHILE, and UNTIL—providing for looping until a particular condition is satisfied.
• INCR, DECR—providing for iterative looping, incrementing or decrementing a loop index.
• EXITLOOP and LEAVE—providing for early termination of loops and for exiting a BEGIN-END block. (There is no GO TO in BLISS.)
• GLOBAL and EXTERNAL data declarations—allowing code to be shared among several modules.
• LOCAL, STACKLOCAL, and REGISTER declarations—allowing dynamic stack-like allocation using either the execution stack or the general registers.
• REQUIRE and LIBRARY declarations—allowing source material from subsidiary files to be included in the module at compile time.
• LEXICAL functions—allowing a variety of compile-time operations such as concatenation of strings, construction of names, testing properties of macro parameters, testing compiler switch options, generating compiler diagnostic messages, and controlling macro expansion.

BLISS-32 also provides a number of machine-dependent features specifically designed to complement the VAX architecture and VAX/VMS. These include the following:
• LINKAGE declarations allow the programmer to make full use of the VAX-11 call facilities; in particular, he may specify either the CALLS/CALLG/RET or JSB/BSB/RSB call and return sequences, pass parameters in general registers or on the stack, and control the use of registers by a routine or across a set of routines.
• PSECT declarations provide information to the linker regarding storage requirements for various sections of a program. For example, a particular data segment may be designated as READ or NOREAD, SHARE or NOSHARE, LOCAL or GLOBAL, and so on.
• System Interfaces—STARLET.REQ provides keyword macros for all VAX-11 RMS and VAX/VMS system services, as well as symbolic definitions for system data structures and completion codes.
• BUILTIN declarations allow use of VAX-11 machine specific functions for access to VAX-11 features not otherwise provided in the BLISS-32 language. The compilation of a machine specific function results in the generation of inline code, often a single instruction, rather than a call to an external routine. Machine specific functions generally have the same name as their corresponding VAX-11 instructions (e.g., ADAWI, BISPSW, CRC, HALT, INDEX, MTPR, PROBER, REMQUE, MOVP, etc.). Over 50 such functions are provided. (The complete list is shown in Table 6-4).

Table 6-4  VAX-11 Machine Specific Functions

PROCESSOR REGISTER OPERATIONS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTPR</td>
<td>Move to a Processor Register</td>
</tr>
<tr>
<td>MFPR</td>
<td>Move from a Processor Register</td>
</tr>
</tbody>
</table>

PARAMETER VALIDATION OPERATIONS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROBER</td>
<td>Probe Read accessibility</td>
</tr>
<tr>
<td>PROBEW</td>
<td>Probe Write accessibility</td>
</tr>
</tbody>
</table>

PROGRAM STATUS OPERATIONS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVPSL</td>
<td>Move from PSL</td>
</tr>
<tr>
<td>BISPSW</td>
<td>Bit set PSW</td>
</tr>
<tr>
<td>BICPSW</td>
<td>Bit clear PSW</td>
</tr>
</tbody>
</table>

QUEUE OPERATIONS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSQUE</td>
<td>Insert entry in Queue</td>
</tr>
<tr>
<td>REMQUE</td>
<td>Remove entry from Queue</td>
</tr>
</tbody>
</table>

BIT OPERATIONS

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTBITSS</td>
<td>Test for Bit Set, then Set bit</td>
</tr>
<tr>
<td>TESTBITSC</td>
<td>Test for Bit Set, then Clear bit</td>
</tr>
<tr>
<td>TESTBITCS</td>
<td>Test for Bit Clear, then Set bit</td>
</tr>
<tr>
<td>TESTBITCC</td>
<td>Test for Bit Clear, then Clear bit</td>
</tr>
</tbody>
</table>
### BIT OPERATIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TESTBITSSI</td>
<td>Test for Bit Set, then Set bit Interlocked</td>
</tr>
<tr>
<td>TESTBITCCI</td>
<td>Test for Bit Clear, then Clear bit Interlocked</td>
</tr>
<tr>
<td>FFS</td>
<td>Find First Set bit</td>
</tr>
<tr>
<td>FFC</td>
<td>Find First Clear bit</td>
</tr>
</tbody>
</table>

### EXTENDED ARITHMETIC OPERATIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASHQ</td>
<td>Arithmetic Shift Quad</td>
</tr>
<tr>
<td>EDIV</td>
<td>Extended Divide</td>
</tr>
<tr>
<td>EMUL</td>
<td>Extended Multiply</td>
</tr>
<tr>
<td>INDEX</td>
<td>Index (Subscript) Calculation</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Calculation</td>
</tr>
</tbody>
</table>

### FLOATING POINT CONVERSION OPERATIONS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVTTLF</td>
<td>Convert Long to Floating</td>
</tr>
<tr>
<td>CVTLD</td>
<td>Convert Long to Double</td>
</tr>
<tr>
<td>CVTFL</td>
<td>Convert Floating to Long</td>
</tr>
<tr>
<td>CVTDL</td>
<td>Convert Double to Long</td>
</tr>
<tr>
<td>CVTFD</td>
<td>Convert Floating to Double</td>
</tr>
<tr>
<td>CVTDF</td>
<td>Convert Double to Floating</td>
</tr>
<tr>
<td>CVTRDL</td>
<td>Convert Rounded Double to Long</td>
</tr>
<tr>
<td>CVTRFL</td>
<td>Convert Rounded Floating to Long</td>
</tr>
<tr>
<td>CMPF</td>
<td>Compare Floating</td>
</tr>
<tr>
<td>CMPD</td>
<td>Compare Double</td>
</tr>
</tbody>
</table>
Programming Languages

Table 6-4  BIT OPERATIONS con’t

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVTTUC</td>
<td>Move Translated Until Character</td>
</tr>
<tr>
<td>SCANC</td>
<td>Scan Characters</td>
</tr>
<tr>
<td>SPANC</td>
<td>Span Characters</td>
</tr>
</tbody>
</table>

**STRING OPERATIONS**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVNP</td>
<td>Move Packed</td>
</tr>
<tr>
<td>CMPP</td>
<td>Compare Packed</td>
</tr>
<tr>
<td>CVTLP</td>
<td>Convert Long to Packed</td>
</tr>
<tr>
<td>CVTPL</td>
<td>Convert Packed to Long</td>
</tr>
<tr>
<td>CVTTP</td>
<td>Convert Packed to Trailing Numeric</td>
</tr>
<tr>
<td>CVTSS</td>
<td>Convert Trailing Numeric to Packed</td>
</tr>
<tr>
<td>CVTSPS</td>
<td>Convert Packed to Leading Separate Numeric</td>
</tr>
<tr>
<td>CVTSP</td>
<td>Convert Leading Separate Numeric to Packed</td>
</tr>
<tr>
<td>EDITPC</td>
<td>Edit Packed to Character</td>
</tr>
</tbody>
</table>

**DECIMAL STRING OPERATIONS**

**MISCELLANEOUS OPERATIONS**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>Halt Processor</td>
</tr>
<tr>
<td>ROT</td>
<td>Rotate</td>
</tr>
<tr>
<td>ADAWI</td>
<td>Add Aligned Word Interlocked</td>
</tr>
<tr>
<td>BPT</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>CHMx</td>
<td>Change Mode</td>
</tr>
<tr>
<td>CALLG</td>
<td>Call with General Argument List</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operating</td>
</tr>
</tbody>
</table>

**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 re-
placement, 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
BLISS-32 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
VAX-11 BLISS-32 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. BLISS-32 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-11 Symbolic Debugger 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. VAX-11 DEBUG understands BLISS syntax and permits the use of the user's symbolic names. (See the section on the Symbolic Debugger for a further description of VAX-11 debugging facilities.)

**Transportability Features**

The 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. BLISS compilers also exist for the DECSYSTEM-10 and DECSYSTEM-20 (BLISS-36), and for the PDP-11 (BLISS-16). BLISS-16 is a cross compiler and runs under VAX/VMS, TOPS-10, or TOPS-20. 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

BLISS'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 VAX-11 BLISS-32 can call VAX/VMS system services and the VAX-11 Common Run Time Procedure Library to print the current time on SYS$OUTPUT.
BLISS SAMPLE PROGRAM

```
MODULE showtime( IDENT='1', TITLE='SHOW TIME', MAIN=timeout);

BEGIN
LIBRARY 'SYSSLIBRARY;STARLET';
Defines System Services, etc.
MACRO
MACRO
M

\[
\begin{align*}
\text{desc[]} & \equiv \text{CHARCOUNT(REMAINING)}, \quad \text{A VAX-11 Style String descriptor} \\
\text{UPTR YTF( REMAINING ) } & \equiv 1
\end{align*}
\]

OWN

\[
\begin{align*}
\text{timebuf} & \equiv \text{VECTOR}(2), \quad \text{64 bit system time} \\
\text{msgbuf} & \equiv \text{VECTOR}(\text{RA,RYTE}), \quad \text{Output msg. buffer} \\
\text{msgdesc} & \equiv \text{PLOCK}(8, \text{RYTE}), \quad \text{String descriptor} \\
\text{PRES} & \equiv \text{(dasc\$length) = RA,} \quad \text{for output buffer} \\
& \phantom{=} \text{(dasc\$pointer) = msgbuf}); \\
\text{BIND}
\end{align*}
\]

\[
\begin{align*}
\text{fmtdesc} & \equiv \text{UPTR(DESC('At the tone, the time is ', %CHAR(7), '115XT'))}; \quad \text{At the tone, the time is }
\end{align*}
\]

EXTERNAL ROUTINE

\[
\begin{align*}
\text{bput \_output 1 ADDRESSING \_MODE(GENERAL)}; \quad \text{From VMS RTL}
\end{align*}
\]

ROUTINE timeouts

\[
\begin{align*}
\text{BEGIN}
\end{align*}
\]

\[
\begin{align*}
\text{LOCAL}
\end{align*}
\]

\[
\begin{align*}
\text{RSLT1 \_ \_WORD}; \quad \text{Resultant string length}
\end{align*}
\]

\[
\begin{align*}
\text{GETTIME \_TIMADR=timebuf}); \quad \text{Get time as 64 bit integer}
\end{align*}
\]

\[
\begin{align*}
\text{FA0L(C),timebuf} \equiv \text{fmtdesc,} \quad \text{Format control=string address}
\end{align*}
\]

\[
\begin{align*}
\text{OUTBUF} \equiv \text{msgdesc}, \quad \text{Output buffer descriptor address}
\end{align*}
\]

\[
\begin{align*}
\text{PRLST} \equiv \text{REF(timebuf))}; \quad \text{Address of pointer to time block}
\end{align*}
\]

\[
\begin{align*}
\text{MSGDESC(dasc\$length) = rslt1; \quad \text{modify output descriptor}}
\end{align*}
\]

\[
\begin{align*}
\text{bput \_output( msgdesc }) \quad \text{print the formatted time}
\end{align*}
\]

END;
```

"At the tone, the time is 115XT"
Programming Languages

Routine Size: 56 bytes, Routine Base: $CODES + 0000

1 0000 END ELUDOM
### PSECT SUMMARY

<table>
<thead>
<tr>
<th>Name</th>
<th>Bytes</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OWN$</td>
<td>96</td>
<td>WRT, RD, NOEXE, NOSHR, LCL, REL, CON, NOPIC, ALIGN(2)</td>
</tr>
<tr>
<td>$SPLITS$</td>
<td>48</td>
<td>NOWRT, RD, NOEXE, NOSHR, LCL, REL, CON, NOPIC, ALIGN(2)</td>
</tr>
<tr>
<td>$CODE$</td>
<td>56</td>
<td>NOWRT, RD, EXE, NOSHR, LCL, REL, CON, NOPIC, ALIGN(2)</td>
</tr>
</tbody>
</table>

### File

<table>
<thead>
<tr>
<th>File</th>
<th>Database</th>
<th>STARLET.L3219</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Total</th>
<th>Loaded</th>
<th>Percent</th>
<th>Blocks</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>2783</td>
<td>4</td>
<td>r</td>
<td>103</td>
<td></td>
</tr>
</tbody>
</table>

### COMMAND QUALIFIERS

BLISS /LIS/NOOB SHOWTIME

<table>
<thead>
<tr>
<th>Size</th>
<th>56 code + 136 data bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Time</td>
<td>00:01:09</td>
</tr>
<tr>
<td>Elapsed Time</td>
<td>00:03:07</td>
</tr>
<tr>
<td>Memory Used</td>
<td>117 pages</td>
</tr>
<tr>
<td>Compilation Complete</td>
<td></td>
</tr>
</tbody>
</table>
VAX-11 BASIC

VAX-11 BASIC is an optional native mode language processing system. An excellent instructional language, VAX-11 BASIC is also a general purpose programming language with a variety of industrial, technical, and commercial applications.

VAX-11 BASIC is a superset of PDP-11 BASIC-PLUS-2/VAX, which is itself a highly extended superset of standard BASIC language as developed at Dartmouth College. VAX-11 BASIC produces shareable native object code, makes full use of the VAX-11 floating point and character string instructions, and is itself shareable.

The compiler accepts a source program written in the VAX-11 BASIC language, and is available in several different ways to the programmer. The programmer may request the compiler to:

- load in a source program for editing
- compile a source program, produce an executable image in local storage, and execute it ("load and go")
Programming Languages

- compile a source program and produce an object module which can be linked with previously compiled object modules

In addition to producing good object code in compile mode, VAX-11 BASIC has a fast RUN command and immediate mode execution that delivers a high degree of interactivenss.

While under control of the BASIC system, a program can be stopped at any point. Once a program is stopped, the programmer can issue any valid BASIC statement, and it will be executed in the context of the stopped program. Program execution can be confirmed at any point, a feature that greatly speeds program development.

VAX-11 BASIC includes many programming facilities not found in most BASIC languages. These are:

- Program formatting and commenting facilities
- Long variable names
- PRINT USING statement
- COMMON statement
- A subprogram CALL statement
- VAX-11 RMS file handling
- Dynamic string handling
- Source program libraries
- Shareable re-entrant programs
- Extended debugging facilities
- EXTERNAL and DECLARE statements

Program Format
The BASIC source program unit is the line. In its simplest form, it consists of a line number, a keyword and statement, and a line terminator. In VAX-11 BASIC one or several spaces or tabs can be used to separate line numbers, keywords, and variable names. Since line number determines the order in which the program is processed, the programmer can write the lines in any order.

VAX-11 BASIC programs can be one or several lines long. The programmer can place one statement on each line, several statements on one line, or spread one statement over several lines. Program comments can be placed anywhere within a line using the REM (remark) statement or using comment field delimiters. These facilities enable the programmer to format a program freely, and thereby to make it more readable.

Long Variable and Function Names
Most BASIC languages limit the length of a variable or user-defined
function name to one character. VAX-11 BASIC recognizes variable names and function names as long as thirty characters. The programmer can thus fully identify variables and functions.

PRINT USING Output Formats
The PRINT USING statement allows the programmer to control the appearance and location of data on an output line to create complex lists, tables, reports, and forms. In addition to the numeric field definitions provided by BASIC-PLUS, which allow the programmer to generate floating dollar sign, aligned decimal point, trailing minus, asterisk fill, and exponential format fields, VAX-11 BASIC provides string field definitions which allow the programmer to generate left-justified, right-justified, centered, and extended string fields.

COMMON Statement
VAX-11 BASIC supports a COMMON statement that can be used to share data among subprograms in an image. The COMMON statement is compatible with both the VAX-11 FORTRAN COMMON and the .PSECT directive of VAX-11 MACRO. Furthermore, by using a global section, data in a COMMON can be shared among cooperating processes.

Calling External Subprograms and Functions
VAX-11 BASIC has both an external program CALL statement and an external function call, as well as highly flexible argument passing mechanisms to enable programmers to interface with:

- VAX-11 BASIC subprograms
- VAX-11 MACRO subroutines
- System services
- Utilities such as SORT
- Other VAX-11 native mode languages such as FORTRAN and BLISS

Arguments can be passed by descriptor, by reference, or by value in accordance with the VAX-11 calling standard. Also note that the VAX-11 BASIC subprograms can be invoked from any other native product.

File Handling
VAX-11 BASIC gives access to almost all the VAX-11 RMS file and record processing options. VAX-11 BASIC has an OPEN statement that allows specification of file organization, access modes, file sharing, record formats, size, and allocation. At the record level, a program can FIND, GET, PUT, UPDATE, DELETE, or RESTORE any record in a file either sequentially or randomly. An important feature is the ability to access files created by any other native mode product.
Programming Languages

Dynamic String Handling
The programmer may manipulate strings of alphanumeric characters easily. The VAX-11 BASIC language includes operators to compare and concatenate strings. Also included are string functions that:
- Create a string of a given length
- Convert string to numeric values and vice versa
- Search for the position of a set of characters within a string
- Insert spaces within a string
- Trim trailing blanks from a string
- Determine the length of a string
Unlike many BASIC languages, VAX-11 BASIC allows up to 65,535 characters in any string.

Source Program Libraries
VAX-11 BASIC has an APPEND command that 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 with reference to the COMMON block then merely APPEND this common file.

Shareable Re-entrant Programs
All code produced by VAX-11 BASIC is always shareable, position independent, and re-entrant. These features allow users to write subprograms and place them in shareable libraries for use by any native language. The linked images of VAX-11 BASIC programs can be shared among any number of users.

EXTERNAL Statement
VAX-11 BASIC provides an EXTERNAL statement for specifying functions or routines that are not included in the current source. This allows the use of EXTERNAL functions, as well as the ability to access symbolic system variables and constants.

DECLARE Statement
The DECLARE statement allows the programmer to declare various data types for program variables. This features allows users to have, for example, both 16-bit and 32-bit integers in a single program.

Debugging Facilities
Debugging facilities for VAX-11 BASIC include diagnostic messages and access to the VAX/VMS DEBUG program. The DEBUG program lets the programmer set breakpoints and tracepoints, and examine
and modify the contents of locations dynamically while executing the program.

**Compatibility**
A compatibility mode BASIC language called PDP-11 BASIC-PLUS-2/VAX is also supported by VAX/VMS. It is an optional language processor that includes its own compiler and object time system. PDP-11 BASIC-PLUS-2/VAX is the same BASIC-PLUS-2 that is available for the RSTS/E, RSX-11M, and IAS operating systems. It produces code that executes in compatibility mode.

**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).

Particularly suited to instructional use, PASCAL is also 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 programs written in PASCAL.

Block structuring and flexible data types make PASCAL a good language for commercial users. It is also suitable for systems programming and research applications.

VAX-11 PASCAL takes advantage of the VAX-11 hardware floating point, character instruction sets, and virtual memory capabilities of the VAX/VMS operating system. Many of the features common to other languages of VAX/VMS are available through VAX-11 PASCAL, including:

- 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:

- Run-time checks for illegal assignment to set and subrange variables, and illegal array subscripts
- Cross-reference listing of identifiers
- Source program listing
- Machine code listing
- Generation of some DEBUG and TRACEBACK records for the VAX-11 Symbolic Debugger
Though PASCAL has access to the Common Run Time Library routines of VAX/VMS, it also has PASCAL-specific Run Time Library routines installed in STARLET.OLB. Such routines primarily provide I/O interfaces to the Record Management Services (RMS).

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, VAX-11 PASCAL 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)`

156
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 VAX-11 PASCAL.

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, pro-
procedure, or function not written in PASCAL (for example, in VAX-11 FORTRAN or MACRO.)

**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). PL/I 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 full PL/I. It excludes features that were more error-prone, difficult to understand or use, and that tended to be implementation-dependent.

VAX-11 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 VAX-11 PL/I include selected features of full PL/I 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 full PL/I features and VAX-11 extensions in programming applications.

**Applications**

**Data processing** applications can take advantage of the extensive character-handling functions and data structuring capabilities of PL/I. 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, VAX-11 PL/I provides full 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.

VAX-11 PL/I supports all RMS file organizations, including sequential, relative, and indexed sequential. VAX-11 PL/I 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 VAX-11 PL/I, 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 trigonometric functions are defined within the language. VAX-11 PL/I supports all of the VAX-11 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-11 extensions to the language provide a simple means to refer to VAX/VMS system global symbols and data structures. Programs written in VAX-11 PL/I can take advantage of the VAX-11 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 PL/I 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 run time, the VAX-11 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-11 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 mes-
Programming Languages

sages are two or three lines long and provide information on how to correct the indicated error.

The VAX-11 Symbolic Debugger 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. The debugger understands all PL/I data types and all PL/I forms of references to variables.

Libraries
VAX-11 PL/I is fully compatible with the VAX-11 Run Time Procedure Library and provides additional run time 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 VAX/VMS configuration. It produces optimized, shareable, VAX/VMS object code that is run-time 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.

PDP-11 BASIC-PLUS-2/VAX
PDP-11 BASIC-PLUS-2/VAX is an optional language processing system that includes a compiler and an object time system. PDP-11 BASIC-PLUS-2/VAX is the same BASIC-PLUS-2 language processing system available as an optional language processor for the RSTS/E, RSX-11M, and IAS operating systems. The PDP-11 BASIC-PLUS-2/VAX compiler produces code that executes in PDP-11 compatibility mode.

BASIC-PLUS-2 is an extended BASIC language which features programming facilities not found in most BASIC languages. These include:

- Program formatting and commenting facilities
- Long variable names
- Virtual arrays
• PRINT USING statement
• COMMON statement
• A subprogram CALL statement
• Extended debugging facilities

The compiler accepts a source program written in the BASIC-PLUS-2 language.

The programmer can edit the source if necessary, and compile it to produce an object module which can be linked with previously compiled object modules.

The object time system is a collection of library modules used during program execution. The library routines include math and floating point functions, input/output operations, error handling, and dynamic string storage functions. Since the OTS is a library, the linker can select only those functions needed at run time to be included in a program. Unnecessary routines are omitted from the program and memory usage is reduced.

Program Format
The BASIC source program unit is a line. In its simplest form, it consists of a line number, a keyword and statement, and a line terminator. In BASIC-PLUS-2, one or several spaces or tabs can be used to separate line numbers, keywords, and variable names. Line number determines the order in which the program is processed; the programmer can write BASIC-PLUS-2 program lines in any order.

BASIC-PLUS-2 programs can be one or several lines long. The programmer can place one statement on each line, place several statements on any one line, or spread one statement over several lines. Program comments can be placed anywhere within a line using the REM (Remark) statement or using comment field delimiters. These facilities enable the programmer to freely format a program to make it more readable.

Long Variable and Function Names
Most BASIC languages limit the length of a variable or user-defined function name to one character. BASIC-PLUS-2 recognizes variable names and function names as long as 30 characters. The programmer can fully identify variables and functions.

Dynamic String Handling
The BASIC-PLUS-2 language enables the programmer to manipulate strings of alphanumeric characters easily. As in BASIC-PLUS, the BASIC-PLUS-2 relational operators enable programmers to con-
Programming Languages

catenate and compare strings, string operators enable the programmer to convert strings and numerics, and string functions add the ability to analyze the composition of strings. The BASIC-PLUS-2 language includes string functions that:

- Create a string of a given length
- Search for the position of a set of characters within a string
- Insert spaces within a string
- Trim trailing blanks from a string
- Determine the length of a string

Unlike many BASIC languages, BASIC-PLUS-2 imposes no limit on the size of string scalars or string elements of arrays manipulated in memory other than the amount of available memory. And BASIC-PLUS-2 handles string storage and manipulation with less overhead than BASIC-PLUS.

Virtual Arrays
Virtual arrays are random access disk-resident files. A program can create and access virtual arrays in the same way memory-resident arrays are accessed: using element names. Explicit read/write programming is not required. The last element in the array can be accessed as quickly as the first. Because the arrays are stored on disk, however, the programmer can manipulate large amounts of data without affecting program size.

PRINT USING Output Formats
The PRINT USING statement allows the programmer to control the appearance and location of data on an output line to create complex lists, tables, reports, and forms. In addition to the numeric field definitions provided by BASIC-PLUS, which allow the programmer to generate floating dollar sign, aligned decimal point, trailing minus, asterisk fill, and exponential format fields, BASIC-PLUS-2 provides string field definitions which allow the programmer to generate left-justified, right-justified, centered, and extended string fields.

Subprograms and the CALL Statement
The BASIC-PLUS-2 CALL statement enables a program to access external subprograms. A programmer can therefore write a program in several modular segments, each of which can be compiled separately to speed program development. BASIC-PLUS-2 provides a complete traceback on errors occurring in subroutines.

COMMON Statement
The COMMON statement enables a program to pass data to another program or subprogram. The BASIC-PLUS-2 COMMON statement
format is compatible with FORTRAN COMMON. Strings passed in COMMON are fixed length, which reduces string handling overhead.

Debugging Statements
BASIC-PLUS-2 provides an interactive debugging mode similar to the "immediate mode" facilities found in most BASIC interpreters. During program development, the programmer can use the compiler to create, save, edit, and test the source program. The compiler checks syntax immediately on input from a terminal so that many errors can be found prior to compilation. The debugging statements can be used when executing and testing the program. The BREAK, LET, PRINT, UNBREAK, CONTINUE, STEP, and STOP statements enable the programmer to control and observe program execution interactively.

To set breakpoints, the programmer uses the BREAK command just prior to running the program, or while it is stopped. As many as ten breakpoints can be set during the course of program execution. On reaching a breakpoint, the program halts to allow the programmer to examine or modify variables or set other breakpoints.

To examine variables while a program is stopped, the programmer uses the PRINT statement. The LET statement allows the programmer to modify the value stored in the variable.

Typing the CONTINUE command resumes execution until the next breakpoint is reached. Before typing CONTINUE, the programmer can issue an UNBREAK command to selectively disable one or all of the breakpoints set, and execution continues until a STOP statement is encountered in the program or until the program completes.

When a program halts because a STOP statement is included in the program, or because a BREAK command was issued interactively, the programmer can type the STEP command on the terminal to let program execution continue on a line-by-line basis. Typing a STOP command in interactive debugging mode terminates program execution, just as if an END statement was encountered in the program.

VAX-11 CORAL 66
The VAX-11 CORAL 66 compiler compiles in compatibility mode and generates native mode object code under VAX/VMS. The CORAL language, derived from JOVIAL and ALGOL-60 in 1966, is the standard language prescribed by the British government for military real-time applications and systems implementation. A government agency controls the language standard for CORAL, which was first widely used in military projects beginning in 1970. Her Majesty's Stationery Office publishes the "Official Definition of CORAL 66."
Programming Languages

The ČORAL 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.

VAX-11 CORAL 66 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 real-time needs, CORAL 66 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, switches, 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
- Switch-selectable option to optimize generated code
- Conditional compilation of defined parts of source code
- English text error messages at compile and (optionally) run-time
- Switch-selectable option to control listing output
- INCLUDE keyword to incorporate CORAL 66 source code from user-defined files
- Switch-selectable option to read card format

VAX-11 CORAL 66 is essentially a high-level block-structured language possessing certain facilities associated with low-level lan-
Programming Languages

guages, and is designed for use on small or medium-size dedicated computers. One of the main intentions is that programs written in CORAL should be fast to execute, taking up limited quantities of storage, while being easy to write.

The real-time 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.
CHAPTER OVERVIEW
Special applications sometimes require extensive special options. Customers who want broad data manipulation powers, screen formatting, and sorting facilities, but whose users are not highly technical, can make use of easy-to-learn utility languages.

Topics are:
• DATATRIEVE
• VAX-11 SORT
• Forms Management System (FMS)
CHAPTER 7
DATA MANAGEMENT FACILITIES

INTRODUCTION
VAX/VMS supports a number of data management facilities: DATATRIEVE, VAX-11 SORT, and the Forms Management System (FMS) utility package.

DATATRIEVE
DATATRIEVE is user application software that provides direct, easy, and fast access to data contained in VAX-11 RMS (Record Management Services) files. The system is designed for relatively unsophisticated computer users; everyday use of DATATRIEVE requires no programming skills. While providing the user with an inquiry language and a report writing facility, DATATRIEVE also supports a user-specifiable Data Dictionary which describes VAX-11 RMS record formats. DATATRIEVE data management facilities include interactive retrieval, sort, update, and display of data records, in addition to maintenance commands for the Data Dictionary.

DATATRIEVE Inquiry Facility
DATATRIEVE accepts English-like commands from the user, and reacts by modifying, updating, or extracting data from the specified VAX-11 RMS file. In those cases where certain sequences of commands need to be issued on a recurring basis, DATATRIEVE provides a feature that permits the definition and use of procedures. A procedure is a group of DATATRIEVE statements and commands identifiable (callable) by a unique procedure name. At any time during the interactive session, this group of DATATRIEVE statements and commands can be invoked simply by calling the procedure name.

DATATRIEVE Report Writer Facility
In addition to query commands, DATATRIEVE provides a report facility to generate reports from VAX-11 RMS files. The report facility allows the user to specify the following parameters:

- Spacing
- Titles
- Column headings
- Page headings and footnotes
- Report headings

Commands to the report facility are simply an extension of query facility commands. Although the report facility provides extensive for-
Data Management Facilities

matting capabilities, its default settings are suitable for many applications, further simplifying its use. Furthermore, errors in commands are discovered immediately (as in the query facility), so the user can correct the commands before printing wrong or incomplete reports.

Basic Commands
DATATRIEVE uses a simple English-like command language for data retrieval, modification, and display. Prompting is automatic for both command and data entry. The major commands are:

- **HELP** — provides a summary of each DATATRIEVE command.
- **READY** — identifies a domain for processing and controls the access mode to the appropriate file.
- **FIND** — establishes a collection (subset) of records contained in either a domain or a previously established collection based on a Boolean expression.
- **SORT** — re-orders a collection of records in either the ascending or descending sequence of the contents of one or more fields in the records.
- **PRINT** — prints one or more fields of one or more records. Output can optionally be directed to a lineprinter or disk file. Format control can be specified. A column header is generated automatically.
- **SELECT** — identifies a single record in a collection for subsequent individual processing.
- **MODIFY** — 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.
- **STORE** — creates a new record. The value for each field contained in the record is prompted for by name.
- **ERASE** — removes one or more records from the RMS file corresponding to the appropriate domain.
- **FOR** — executes a subsequent command once for each record in the record collection, providing a simple looping facility.
- **EXTRACT** — copies domains, records, procedures, and tables from the Data Dictionary to an external file.
- **SHOW FIELDS** — prints field names and data types for all fields in ready domains.
- **DEFINE DICTIONARY** — allows creation of private dictionaries.

In addition to the simple data manipulation commands, a number of more complex commands are available for the advanced user. These commands, such as REPEAT, BEGIN-END, and IF-THEN-ELSE, may be used to combine two or more DATATRIEVE commands into a sin-
Data Management Facilities

gle compound command. These, in turn, may be stored in the Data Dictionary as procedures for invocation by less experienced users.

DATATRIEVE provides a full set of arithmetic operators (addition, subtraction, multiplication, division, and negation), a set of statistical operators (total, average, maximum, minimum, and count), and provides automatic conversion between data types used in the FORTRAN, COBOL, DIBOL, and BASIC languages.

Terminology
Files, domains, collections, records, and fields are terms of fundamental importance to the file structure of DATATRIEVE.

Records are groups of related items of data that are treated as a unit. For example, all the pieces of data describing a model of a yacht in a marina could be grouped to constitute the record for that yacht.

Each of the individual pieces of data in a record is referred to as a field. The yacht’s model number, length, and price are all potential fields in its record.

The term files refers to the logically related groups of data that are kept by RMS. For example, we might put all of the yacht records for a current inventory of yachts into one file.

Domains are named groups of data containing records of a single type. A DATATRIEVE domain consists of all the records in a particular RMS file, in addition to a record definition of this file contained in the Data Dictionary. In this case, we could say that all the yacht records for the current inventory are kept in the YACHTS domain. The number of records in any domain may change as new records are stored or old records are erased.

A record collection is a subset of a domain. It may consist of no records, one record, or up to all the records in the domain. Using our previous example, we could say that all the yachts manufactured by Grampian could be made to form the Grampian collection, while those yachts manufactured by Islander could be used to form the Islander collection. To carry this example one step further, if the inventory is currently out of stock of yachts manufactured by Seaworthy, the Seaworthy collection will be empty, or null.

The Data Dictionary is a location where the definitions for procedures, records, and domains are kept in a standard fashion by DATATRIEVE. The data administrator will be concerned with the creation and maintenance of Data Dictionary information. Certain users will be able to display certain information from this dictionary, but only management will be concerned with defining it.
Processing a File as a Collection
Perhaps the most important systems concept to master is collection processing. DATATRIEVE operates on collections of records taken from the files. To get down to the level of record processing, the FIND and SELECT commands are employed to gather the collection and extract the records desired. The system provides a cursor facility to track the user's place in a collection. Figure 7-1 illustrates the cursor as a place marker, and shows how it can be manipulated through the collection, from the first, to the next, to the last record. It need not always move forward; directional movement within a collection is at the user's discretion.

In most DATATRIEVE operations, the files are never changed, but a great deal of manipulation occurs on the collections. Thus, the collections can be thought of as a sort of temporary storage, kept for immediate purposes, and then released.

The FIRST record of the collection is the first one encountered when it was established or collected. The record numbered with (1) in Figure 7-1 identifies the FIRST record in this example.
The NEXT record is always the one immediately after the CURRENT one. In the figure, the CURRENT record is the sixth one, so the NEXT record must be the seventh. The LAST record is the one located at the very end of the collection.

If there happen to be no records at all in a collection, the collection is null, therefore, the collection cursor will be null.

Generally, the user works on just one collection at a time, the CURRENT one. However, it is possible to name a collection and refer to it later in the same session.

Thus, it is possible to be working with a number of collections, one of which (the one most recently the object of a FIND command) is the CURRENT one. Each of the collections has its own collection cursor and may have a designated CURRENT record, as well. However, only one collection may be CURRENT at a time.

The lifetime of any collection is limited to a given session. Collections cannot be saved, nor can they be shared by several users. However, the user can always reproduce a collection of identical characteristics (although not necessarily identical records, since the files can change in the interim), simply by repeating the sequence of FIND commands used initially to establish and/or define the collection. If this is a frequent requirement, the steps can easily be concatenated as a procedure.

**Keywords**
DATATRIEVE utilizes language elements called keywords which have a specific denotation and associated function. If they are used in any other context, they may serve to confuse the system about user intentions. Thus, it is good policy to avoid the use of these words as names of domains, procedures, records, fields, and collections.

**Additional DATATRIEVE Features**
Among the many DATATRIEVE features supported by VAX/VMS are:

- Application Design Tool (ADT)
  ADT allows less experienced users to set up simple DATATRIEVE applications. Through a simple dialogue, ADT generates a command file containing the record, domain, and file definitions.

- Nested Procedures
  Procedures may contain references to other procedures (nested procedures) provided that no procedure invokes itself either directly or indirectly. The maximum depth of nesting varies from about 10 to about 30 depending on the amount of memory available, number and size of established collection, etc.
Data Management Facilities

- Data Hierarchies
  Use of hierarchies allows manipulation of complex data containing lists and sub-lists. A hierarchy may be defined as a single file with a repeating group or multiple domains automatically cross linked. Extensions related to hierarchies include the inner print list (to override default formatting of a sub-list) and the ANY Boolean expression, which allows DATATRIEVE to search a sub-list for the existence of a particular record.

- Views
  Views can be used to restrict the set of fields accessible, to apply an automatic selection criterion to a file, or to cross link a number of elementary domains to from an apparent hierarchy. Once defined, a view is indistinguishable from an RMS domain to the user.

- DATE Data Type
  This allows easy inclusion of dates in DATATRIEVE records, direct comparison of dates, computation of elapsed dates. Dates may be formatted for printing in virtually any form. Similarly, DATATRIEVE accepts the entry of dates in virtually any form. The DATATRIEVE date format is compatible with the VAX/VMS date standard.

- Tables
  Tables are generally used to translate encoded values into something that can be edited by the DATATRIEVE editor. Table lookups are performed by the VIA value expression; table searches (for table membership) are specified with the Boolean IN expression.

- TOTAL Statement
  The TOTAL statement allows very simple computation of totals and sub-totals.

- CONTAINING Relational Operator
  CONTAINING is used in a record selection expression to retrieve records with a field containing a particular substring. The substring may be anywhere in the field, and need not match the case (upper/lower case) of the search string. For example, the command:

  FIND BOOKS WITH TITLE CONTAINING "LASER"

  will find all records in BOOKS with the word "LASER" somewhere in the field TITLE.

- OCCURS Clause
  Use of the OCCURS clause permits definition of records containing a repeating group (sub-list). The sub-list may be of fixed or variable length.
Data Management Facilities

- Value Validation
  A Boolean validation expression may be included as part of a field description in a record definition. If specified on a field, the validation expression is automatically executed every time the field is modified, to insure that only legal values are stored in a data base. If a validation error is detected, the user is re-prompted for a new value if possible, or the DATATRIEVE statement is aborted.

- COMPUTED BY Fields
  A field in a record definition may be defined as a COMPUTED BY field by specifying a value expression to be computed for its value. A COMPUTED BY field takes no space in the actual RMS record, and is computed on reference. A COMPUTED BY field may be used in conjunction with a table to provide completely automatic table look-up.

- Tutorial Software (Guide Mode)
  A CRT-based tutorial is included in DATATRIEVE. The tutorial feature can be used only by VT52, VT52-compatible, and VT100 terminals. A tutorial session is entered by the DATATRIEVE command:

  SET GUIDE

  The software is self-documenting.

- Procedure Editor
  A DATATRIEVE procedure editor has been added. The editor is invoked by the command:

  EDIT procedure-name

  where "procedure-name" is the name of an existing procedure.

  The command EDIT procedure-name invokes an editor which can insert, replace, or delete text from procedures defined in the data dictionary.

Summary of DATATRIEVE Commands and Statements

Commands
DEFINe DOMAIIn domain-name-1 USING record-name-1
  ON rms-file-spec-1;
DEFINe FILE domain-name [option-1, option-2, ...]
DEFINe PROCEDURE procedure-name-1

DATATRIEVE statements and commands

END-PROCEDURE;
DEFINE RECORD record-name-1 USING [data-def-2...];
DEFINEP resource-name-1 [(password-str-1)] seg-number,
   lock-type-1,key-1, privilege-str-1
   domain-name-1
   record-name-1
   procedure-name-1
   } [(password-str-1)];
DELETE resource-name-1 [(password-str-1)] seq-number
DELETEP resource-name-1 [(password-str-1)] [OF rse]
EDIT procedure-name
ERASE [ALL [OF rse]]
EXIT
FIND domain-name-1 [WITH condition]
FIND CURRENT [WITH condition]
FIND record-selection-expression
FINISH [domain-name-1 [,domain-name-2...]]
HELP [ADVANCED] [command-name-1[,command-name-2...]]
MODIFY [ALL] [USING statement-1][VERIFY USING statement-2]
   [field-name-1[,field-name-2...]] [OF rse]
PRINT [ALL] [print-list-1] [OF rse] [ON file-spec-1]

READY domain-name-1 [(password-str-1)]
   [SHARED
   PROTECTED
   EXCLUSIVE
   ] [READ
   MODIFY
   WRITE
   EXTEND]

RELEASE collection-name-1 [,collection-name-2...]
SELECT
   FIRST
   NEXT
   LAST
   [collection-name-1]
   value-exp-1
SET { DICTIONARY rms-filespec
      } [NO] prompt
SHOW show-item-1 [,show-item-2...]

where the show items are chosen from the following list:

   PROCEDURES
   DOMAINS
   COLLECTIONS
   RECORDS
   ALL
CURRENT
READY
DICTIONARY
procedure-name-1[(password-str-1)]
domain-name-1[(password-str-2)]
record-name-1[(password-str-3)]
collection-name-1
SHOWP resource-1 [(password-str-1)]
SORT [collection-name-1] BY [sort-key-1][,sort-key-2...]
where the sort-keys assume the following form:
ASC[ENDING]
DESC[ENDING] field-name-1
INCREASING
DECREASING
STORE domain-name-1 [USING statement-1][VERIFY
USING statement-2]

Statements
field-name-1 = value-exp-1
field-name-1 = field-name-2

ABORT value-exp-1

BEGIN statement-1 [;statement-2...] END

DISPLAY value-exp-1

FOR rse-1 statement-1

IF condition THEN statement-1 [ELSE statement-2]

REPEAT value-exp-1 statement-1
statement-2 THEN statement-3

Report Writer Statements
field-name-1

AT \{ TOP\} OF \{ PAGE
REPORT \} PRINT summy-item-1
[,summy-item-2...]

PRINT detail-item-1 [,detail-item-2...]

REPORT [rse] [ON file-spec-1]

REPORT END

SET parameter-1 [,parameter-2...]
where the parameters are chosen from the following list:
REPORT-NAME = report-name
MAX-LINES = integer-1
MAX-PAGES = integer-2
NUMBER
NO-NUMBER
DATE = ["string-1"]
NO-DATE
LINES-PAGE = integer-3
COLUMNS-PAGE = integer-4

Subexpressions
Where the record selection expression (rse) assumes the following form:

\[
\begin{align*}
\{ & \text{ALL} \\
\{ & \text{FIRST} \ n \} \ [\text{collectn-name-2 IN}] \\
\{ & \text{CURRENT} \\
& \text{collectn-name-3} \} \ [\text{WITH} \ \text{conditn}] \\
& \text{domain-name-1} \\
\} \\
& \text{[SORTED BY} \ \text{key-1[,key-2...]]}
\end{align*}
\]

where each sort-key is in the form:

- ASC[ENDING]
- DESC[ENDING]  field-name-1
- INCREASING
- DECREASING

VAX-11 SORT
VAX-11 SORT is a native mode utility that may be utilized in a stand-alone mode or may be called from VAX-11 native mode user programs.

The SORT utility program allows the user to reorder data from any input file into a new file in a sequence based upon key fields within the input data records. The sorting sequence is determined by user specified control fields, also known as key fields, within the data itself. If the user does not wish to reorder the data base, SORT can still be used to extract key information, sort that information, and store the sorted information as a permanent file. Later that file can be used to access the data base in the order of the key information in the sorted file. The contents of the sorted file may be entire records, key fields, or record file addresses which point to the position of each record within the file.

VAX-11 SORT can perform the following functions:
- Reorder data files (records are sorted in ascending or descending order by up to ten keys which can be in any order)
Data Management Facilities

- Create reordered address files of RFAs and keys for software use
- Sort fixed, variable, and VFC records in ASCII collating sequence
- Sort sequential, relative, indexed-sequential files
- Sort character, decimal, and binary data types
- Determine its own work file requirements based on input file RMS information received
- Be controlled by a command string or specification file.
- Be tuned for maximum efficiency
- Provide four processing techniques: record, tag, address, index
- Sort input files from any VAX/VMS input device
- Output sorted data files to any VAX/VMS output device
- Automatically print out statistics upon completion
- Be invoked by a single command string, or can prompt the operator for input and then output file specification
- Respond with unique SORT error messages in VAX/VMS format

VAX-11 SORT supports the following data formats:

- Character data is ASCII
- Binary data is VAX representation
- Packed decimal data is VAX representation
- Zoned decimal data is VAX representation
- String decimal data format can be:
  - leading separate sign
  - leading overpunched sign
  - trailing separate sign
  - trailing overpunched sign

FUNCTIONAL PROGRAM DESCRIPTION

The SORT program consists of two basic parts: a control program and a callable subroutine package (see Figure 7-2). The control program directs the overall processing. The callable subroutine package serves as a collection of subroutines that the control program uses during its processing.

The user can write a program to take advantage of SORT’s callable subroutines.
There are three major phases of operation in the SORT control program. In the first phase, SORT reads the command string, decodes it, and stores the information. Then the specification file is read and decoded, if present. Any errors in the command string or specification file are reported at this point.

Second, SORT begins the pre-sort operation. The control program calls routines to open and read the input file and establish the keys. Then a routine is called to begin the initial sorting process. At this point, the amount of available internal storage space becomes important to the efficiency of the sort. If that space is not sufficient to hold all the records, SORT builds strings of sorted records and transfers them to work files on temporary storage devices (disk). The SORT program normally provides for a default of two work files. A qualifier in the command string can increase the number of work files used.

Third, SORT rebuilds the intermediate work files into a merged file. If the process is Tag Sort, another subroutine reads the records in the proper sequence. The records are then written in the output file. If there are no work files to merge because main memory was sufficient
to hold all the records, the sorted records are written directly into the output file. After the last record is written, the control program calls a routine to clean up the work files and exits; SORT is then ready to accept another job.

SORT provides four sorting techniques:

- **Record Sort** produces a reordered data file sorted by specified keys, moving the entire contents of each record during the sort. A record sort can be used on any acceptable VMS input device and can process any valid VAX-11 RMS (Record Management Services) format.

- **Tag Sort** produces a reordered data file by sorting specified keys, but moving only the record keys 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 conserves 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) a pointer to each record’s location in the file which can later be used as an index to read the data base in the desired sequence. Any number of address files may be created for the same data base. A customer master file, for instance, 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** 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.

VAX-11 SORT is designed to be used in conjunction with VAX-11 RMS and may be invoked via the VAX/VMS Digital Command Language command SORT.

Input files of sequential, relative, or indexed organization containing records of fixed, variable, or variable-with-fixed-length control (VFC) format can be input to SORT via disk, magtape, card reader or terminal. Input parameters to SORT are specified in the form of input file specification qualifiers.

Sorted output data files can be created as either sequential, relative, or indexed organization. These files may contain record types of either fixed, variable, or VFC format. Data files can be output to disk, magtape, printer, or terminal.

Sorted output address files of 6-byte record file addresses (RFAs) in binary coded records are output to disk only for sequential access by
programs. Output parameters to SORT are specified in the form of output file specification qualifiers.

COMMAND STRING AND SPECIFICATION FILE
The user can direct the SORT program by entering a command string. The command string has three functions:
1. To reference devices in the system for each file in the current sort.
2. To specify switches that define file parameters used in the sorting process.
3. To reference a specification file or to specify other switches to control the sort.

Command String
The command string consists of three parts: the command name, the input file specification parameter, and the output file specification parameter, in that order. Each part is separated by a space or tab, and is invoked by terminating with <RET>:
1. Command Name (SORT)
   SORT qualifiers specify the sort process, and describe the sorting key(s), specify the number of work files, indicate the specification file if a sort other than a standard sort is to be performed, and finally indicate whether the VAX-11 SORT utility or the RSX SORT-11 utility is to be invoked.
2. Input File Specification Parameter
   This parameter specifies the physical location of the input file. Input file qualifiers define the input file attributes such as record format, block size, and file size.
3. Output File Specification Parameters
   This parameter specifies the physical location of the sorted output file.
   Output file qualifiers define the output file attributes such as record format, record size, block size, file organization, space allocation, contiguous allocation, overlay existing file, and bucket size.

The general format of a command string is:

$SORT[Qualifiers]<SP>Input filespec[Qualifiers]<SP>-Output filespec[Qualifiers]<RET>

The VAX/VMS command interpreter will prompt the user for input and output filespecs if they are not entered in the first command string. The following is an example of prompted format:
Data Management Facilities

$SORT/KEY:(POS:1,SIZE:80)<RET> command with qualifiers
$-File: R100SQ<RET> input filespec
$-Output: TESTTMP<RET> output filespec

Specification File
As previously mentioned, one of the qualifiers in the SORT command string is the specification file qualifier. Use of the specification qualifier in the SORT command string allows the SORT utility to be controlled by SORT specification statements contained within the specification file.

SORT as a Callable Subroutine
SORT can be used as a set of callable subroutines from any high level language. This subroutine package provides two functional interfaces to choose from: a file I/O interface and a record I/O interface. Both interfaces share the same set of routines, and the same calls are used for all languages.

With this release of VAX/VMS, full MERGE capability has entered the list of SORT features callable as a subroutine. Files not recently processed by SORT may, nevertheless, be MERGEd. This allows much-increased file flexibility.

The sorting interface subroutines consist of six external function calls, each of which causes a phase of the SORT program to be performed, and returns a status (32-bit) value indicating either the success or failure type of the phase. Each call and associated parameters conforms to the VAX-11 standard calling interface. The calls are:

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOR$INIT_SORT</td>
<td>Initialize scratch files, work area, sorting parameters</td>
</tr>
<tr>
<td>SOR$PASS_FILES</td>
<td>Pass a filename to SORT</td>
</tr>
<tr>
<td>SOR$RELEASE_REC</td>
<td>Pass a record to SORT</td>
</tr>
<tr>
<td>SOR$SORT_MERGE</td>
<td>Initiate sorting and intermediate merging of records</td>
</tr>
<tr>
<td>SOR$RETURN_REC</td>
<td>Initiate final merge pass and receive output record from SORT</td>
</tr>
<tr>
<td>SOR$END_SORT</td>
<td>Allow clean-up of files and work area to complete the SORT operation</td>
</tr>
</tbody>
</table>
Data Management Facilities

File I/O Interface
The file I/O interface allows the user to specify an input file and an output file to SORT. SORT then reads the data from the input file and sorts it into the output file.

For the file I/O interface, use the following four calls in the order listed:

**Call**  
SOR$PASS_FILES  
SOR$INIT_SORT  
SOR$SORT MERGE  
SOR$END_SORT  

**Function**  
Pass file names  
Initialize work areas  
Sort records  
Clean-up work areas

Record I/O Interface
The record I/O interface allows the user to pass each individual data record to SORT, let SORT order them and then receive each record back in the correct order, individually, from SORT.

For the record I/O interface, use the following five calls in the order listed:

**Call**  
SOR$INIT SORT  
SOR$RELEASE REC  
SOR$SORT MERGE  
SOR$RETURN REC  
SOR$END_SORT  

**Function**  
Initialize work areas  
Pass a record  
Sort records  
Receive a record  
Clean-up work areas

**NOTE**
Calls 2 and 4 are each repeated for as many times as there are records to be sorted.

Programming Considerations
The file I/O interface provides a faster sort, however, any program can use either SORT subroutine package if the language it is written in supports the following features:

- 32-bit integers
- Longword addresses
- Call by string descriptors
- Call by reference
- Either CALLS or CALLG (that is, VAX/VMS standard calling sequence)
- External function calls (each sort routine returns a 32-bit status code)
VAX-11 SORT PERFORMANCE
Table 7-1 represents approximate performance figures for VAX-11 SORT. Several sample cases were executed in both 128-page and 1024-page working set memory environments. These SORT runs were stand-alone, i.e., they were not executed with other users active.

The following environment was used for all cases:
- 80-byte random ASCII records
- 80-byte random ASCII keys
- full record sort
- all files stored on the system disk, an RP06
- default number of work files used
- stand-alone system
- elapsed time is wall time in:
  MIN:SEC.1/100th SEC

<table>
<thead>
<tr>
<th>Case 1 — 128-Page Working Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># Records</strong></td>
</tr>
<tr>
<td>1,000</td>
</tr>
<tr>
<td>10,000</td>
</tr>
<tr>
<td>15,000</td>
</tr>
<tr>
<td>20,000</td>
</tr>
<tr>
<td>40,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case 2 — 1024-Page Working Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong># Records</strong></td>
</tr>
<tr>
<td>1,000</td>
</tr>
<tr>
<td>10,000</td>
</tr>
<tr>
<td>15,000</td>
</tr>
<tr>
<td>20,000</td>
</tr>
<tr>
<td>40,000</td>
</tr>
</tbody>
</table>

VAX-11 FORMS MANAGEMENT SYSTEM (FMS)
VAX-11 Forms Management System (FMS) is a utility package used to provide video form support for applications on the VT100 video terminal. VAX-11 FMS provides a flexible, easy-to-use interface between the form application program and the terminal user. FMS allows a terminal user to develop form applications using both native mode language processors (VAX-11 MACRO, VAX-11 FORTRAN, VAX-11 BASIC, and VAX-11 COBOL-74) and compatibility-mode processors (PDP-11 FORTRAN-IV-PLUS/VAX, FORTRAN IV/VAX TO RSX, PDP-11 BASIC-PLUS-2/VAX, AND MACRO-11).
Using Forms in an Application

VAX-11 FMS forms include a video screen image comprising data fields and constant background text, along with protection and validation attributes for individual data fields. The data fields and background text can be highlighted using any combination of the VT100 video attributes: reverse video, underline, blink, and bold characters. Split screen and scrolling capabilities allow the user to view more data than can be displayed on the screen at one time.

Individual data fields can be display-only, enter-only (no echo), or can be restricted to modification by privileged users. Data fields can be formatted with fill characters, default values, and other formatting characters—such as the dash in a phone number—which assist the terminal operator, but which are not visible to the application program. Fields may be right- or left-justified or may use a special fixed decimal data field type to normalize floating point decimal numbers into fixed point for easier use in computation.

Field validation includes checking each keystroke in a field for the proper data type (e.g., alphabetic, numeric, etc.). Fields may also be defined as “must enter” or “must complete.”

A line of HELP information may be associated with each field, and a chain of one or more HELP forms may be associated with each form. If people need additional instructions while using a form, they press the HELP key to display the HELP line for the current field. A second HELP keystroke displays the first HELP screen for the current form, so that from any point in the application form the user can get to an entire series of HELP forms. In this way the entire user manual for an application can be put on-line, automatically keyed to the appropriate user form.

Almost any class of application can benefit from using VAX-11 FMS. Source data entry and inquiry/response/update are the most obvious types of forms-oriented uses, but other types of programs can benefit equally well. For example, a simulation or numerical analysis program could use FMS forms to explain and accept input parameters, and then to format and scroll through the output of the run. Forms might constitute the front end of a student registration or a computer-aided instruction system. Alternatively, they could be used to review data acquired from laboratory or factory instrumentation, or to format the operator input of control parameters to such processes. Almost any application which uses alphanumeric video terminals can be enhanced by using FMS forms to talk to the terminal user.
Developing Applications with VAX-11 FMS

VAX-11 FMS forms are created and modified interactively on the screen using a special FMS utility called the Form Editor (FED). Because the video image is typed and manipulated directly on the screen, there is no need to lay out a form on a paper chart or to code complex specifications into a form definition program written in a difficult language. Rather, the form creator always sees the form on the screen exactly as it will appear to the application user. A set of 24 editing and data manipulation functions invoked through the function keypad of the VT100 terminal allow easy alteration of the form description.

Fields are defined interactively via the function keypad, and by typing the COBOL-like picture character for each position of the field directly on the screen. The remaining field attributes, such as field name, default value, and the contents of the HELP line, are described by interactively filling in a questionnaire form on the screen. Another form is used to define certain characteristics which apply to the form as a whole, such as the name of the form and of the first HELP form associated with it, and whether the screen is to be placed into reverse video or 132-column mode. A third form is used to specify application constants, called "named data," which can be stored with the form instead of hard-coded into the application program. This last feature allows application parameters such as small data tables, file names, names of subsequent forms, etc., to be stored with the form and edited almost interactively.

When the application developer is satisfied with the appearance and content of the form, the Form Editor writes the form out into a work file. The Form Utility (FUT) is then used to insert the form into a new or existing form library, from which it will be retrieved when the application program is executed. The Form Utility may be used to perform other maintenance functions on form libraries, as well as to generate hard-copy descriptions of forms suitable for inclusion in application documentation. FUT can also generate COBOL Data Division code to correspond to the form description.

Once the form has been stored in a form library, one or more application programs to use it must be coded. These application programs control the interaction between themselves, the form, and the operator by making calls to a library of VAX-11 FMS subroutines called the Form Driver (FDV). Under the direction of the calling application program, the Form Driver displays forms, performs all screen management an application requires, handles all terminal input and output, and validates each operator entry by checking it against the field description for the field. A broad selection of subroutine calls allows
Data Management Facilities

the program to communicate with the screen on either a full-screen or field-by-field basis. While the terminal operator is typing data, all data validation and formatting, error messages, and HELP requests occur completely transparently to the application program.

Maintaining VAX-11 FMS Applications
Several features of VAX-11 FMS make maintenance of applications using forms both rapid and reliable. The most obvious such feature is the interactive editing capability of the Form Editor. Capabilities such as Open Line for insert, Cut and Paste, etc., make modifications to existing forms quick and easy. Furthermore, when fields are moved around on the screen, their attributes are preserved, so that re-entry of the form is not required.

Perhaps the most important contribution to application maintainability comes from the fact that the application makes all references to screen data by field name. These field name references are not resolved until the program executes, so that the form description is actually independent of the application program. This means that it is easy to write programs that do not know or depend upon the specific order of the fields, or even know the names or each field. This capability, combined with the storage of forms in libraries, means that in many cases the form can be rearranged, or new fields added, without requiring that the application program be recompiled, or even relinked!

The last feature of VAX-11 FMS that promotes application maintainability is the ability to store application parameters with forms as named data. Parameters such as the names of related forms or programs, file specifications, small look-up tables, range check boundaries, and error messages specific to the particular form may be stored with the form and edited with the same rapidity and ease. For example, imagine an application that will be used by operators who speak a variety of different languages. The first thing the operator would do when logging into the application would be to select a language. The program would simply open the form library with all the forms translated into that language. Named data would be used to store all other language-dependent application parameters, such as program-generated error messages, abbreviations (Y=YES or O=OUI or S=SI), etc. The same program code would then be executed regardless of the language the application would “speak.” A new language could be added by simply modifying the initial language selection form and creating a form library with the forms and named data translated into the new language.
CHAPTER OVERVIEW

Special notice needs to be given to some of the important 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. The VAX/VMS DEBUG facility can help programmers step through their code to detect and correct logical and coding errors. The extensive Common Run Time Procedure Library, which holds coded algorithms ready for linking into user processes, is treated, as is the RUNOFF document formatting facility.

Topics include:

- Text Editors
- RUNOFF
- Linkers
- DEBUG Facility
- Common Run Time Library
INTRODUCTION
VAX/VMS 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 procedure 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 5, DIGITAL Command Language.

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

The user invokes the SOS and EDT text editors interactively with the computer system. That is, the user creates and processes files on-line. 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 SOS. Therefore, to invoke SOS, enter the command EDIT or EDIT/SOS; to invoke EDT, enter EDIT/EDIT; for SLP, use EDIT/SLP.

THE SOS EDITOR
SOS is a line-oriented, interactive text-editing program. SOS has features that allow examination and modification of text, character by character. SOS 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

189
SOS is line-oriented, so 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. SOS 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, SOS maintains line numbers in ascending order within each page of text.

In certain modes of operation, SOS 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.

**File Names and File Types**

By taking advantage of the default disk and directory, the user can identify a file uniquely by specifying its file name and file type, illustrated in the following format:

```
filename.typ
```

The file name can be from one to nine alphanumeric characters, and can assume any name that is meaningful to the user.

The file type is a 3-character identifier preceded by a period; it describes more specifically the kind of data in the file. Although file type can consist of any three alphanumeric characters meaningful to the user, the system automatically defaults to several file types for special purposes. Among these special file types are:

<table>
<thead>
<tr>
<th>File type</th>
<th>Default Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>.FOR</td>
<td>FORTRAN language source statements</td>
</tr>
<tr>
<td>.MAR</td>
<td>MACRO assembly source statements</td>
</tr>
<tr>
<td>.COB</td>
<td>COBOL language source statements</td>
</tr>
<tr>
<td>.BAS</td>
<td>BASIC language source statements</td>
</tr>
<tr>
<td>.DAT</td>
<td>A data file</td>
</tr>
<tr>
<td>.LIS</td>
<td>An output listing from a compiler</td>
</tr>
<tr>
<td>.EXE</td>
<td>An executable image</td>
</tr>
</tbody>
</table>

For example, a file containing FORTRAN source statements would possess the file type .FOR.
Support Facilities

Initiating and Terminating SOS
SOS is initiated by entering one of the following commands in response to the command language prompt:

$ EDIT file-spec <RET>

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

*E<RET>
[file-spec]
$

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

SOS Modes of Operation
SOS 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. SOS 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.

SOS 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 NEW.FOR <RET>
SOS VERSION V02.02D2
INPUT: SYS0:[TERRY]NEW.FOR.1
00100

SOS 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 5.

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. SOS 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 SOS. 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 8-1 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 <CR>
SOS VERSION V02.02D2
File:PROG2.COB<CR>
EDIT:SY0:[EMILY]PROG2.COB.4
```

<table>
<thead>
<tr>
<th>Form</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Copy</td>
<td>Copy a range of lines to another place within a file, or from another file</td>
</tr>
<tr>
<td>D</td>
<td>Delete</td>
<td>Delete a range of lines</td>
</tr>
<tr>
<td>E</td>
<td>End</td>
<td>Terminate SOS, return to command language monitor</td>
</tr>
</tbody>
</table>

Table 8-1   Common Edit Mode Commands
<table>
<thead>
<tr>
<th>Key</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Find</td>
<td>Search for the occurrence of one or more specified strings of text</td>
</tr>
<tr>
<td>H</td>
<td>Help</td>
<td>Print HELP facility on terminal</td>
</tr>
<tr>
<td>I</td>
<td>Input</td>
<td>Enter Input mode to insert lines of text</td>
</tr>
<tr>
<td>N</td>
<td>reNumber</td>
<td>Renumber a range of lines</td>
</tr>
<tr>
<td>P</td>
<td>Print</td>
<td>Print a range of lines on the terminal</td>
</tr>
<tr>
<td>R</td>
<td>Replace</td>
<td>Delete a range of lines and enter Input mode</td>
</tr>
<tr>
<td>S</td>
<td>Substitute</td>
<td>Replace one or more text strings with other string(s) in a range of lines</td>
</tr>
<tr>
<td>T</td>
<td>Transfer</td>
<td>Copy a range of lines to a new location and delete the original lines</td>
</tr>
<tr>
<td>W</td>
<td>Save World</td>
<td>Write a new file containing all the changes made so far and continue editing</td>
</tr>
</tbody>
</table>

**SOS Examples**

Copy command

1) C300,9000:95000
   Make a copy of lines numbered 9000-95000 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.
Support Facilities

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 num-
bered 500 through 800.

Transfer command
1) T300,9000:9500
Move all lines numbered 9000 through 9500 to a point follow-
ing line 300, deleting the lines in the old location.

THE EDT EDITOR
EDT, an interactive text editor, is included with VAX/VMS Version 2.0.
This editor lets users enter and manipulate text and programs. EDT is
used to view and modify a file directly.

EDT, with its extensive HELP facility, is designed to be learned easily
by novices. In addition, EDT 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 and keypad editing on the VT52 and VT100 video
terminals
• Ability to work on multiple files simultaneously
• A journaling facility, which protects against loss of edits due to sys-
tem crashes
• An extensive HELP facility
Support Facilities

- A default start-up command file, which allows a choice of editing options to be set automatically
- A window into a file (on video terminals only) that lets users view changes in file contents immediately
- Shareable installation for multiple users

EDT is also supported on hardcopy terminals and video terminals other than the VT52 and VT100.

SPECIAL FEATURES

Editing with a Window
“Window editing” is a valuable feature that lets users edit one 22-line window (screenful) at a time. This feature allows a user to see immediately how the edits made affect his file. The file moves through the window so that one always sees the cursor.

![Figure 8-1 Window Editing](image)

Start-up File
When the editor is started, it executes commands from a start-up file. In this file, one can insert editing options such as SET NOKEYPAD and DEFINE KEY. These options take effect automatically when an editing session begins.

HELP Facilities
The HELP facilities on EDT are extensive. Users can get help on general EDT operations by typing HELP. While in keypad mode, users can get help by pressing the help key, which displays information that is specific to keypad editing.

Redefining Keypad Keys
One can redefine any of the keypad keys, and most of the control (CTRL) keys, on VT52 and VT100 terminals. This feature lets the user assign a series of commands to a key; EDT performs these commands
when the keys are pressed. Therefore, one can adapt the functions of keypad and CTRL keys to meet special needs.

The SET and SHOW Commands
The SET command, with a variety of qualifiers, affects EDT’s editing capabilities. SET controls such screen parameters as line width. SET also lets a user determine the appearance of text, such as changing the window size to less than 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 the 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 may choose to retrieve and execute commands in this saved file with the /RECOVER option. In this way the user can recover edited files to the time of the crash.

The CAI Program
Also available with EDT is a Computer-Assisted Instruction (CAI) program. This interactive program presents the “Introduction to the EDT Editor” minicourse, which demonstrates how to use EDT. The CAI program runs on VT100 terminals and takes about three hours.

The minicourse teaches the novice user the basics of both line and character editing on EDT. The CAI program uses sophisticated video displays that make the course interesting and effective.

EDT Modes of Operation
A “mode” in EDT is a state in which the editor lets a user perform a specific set of functions. EDT has two basic modes of operation: line mode and change mode.

Line mode allows users to establish editing parameters and to display and edit text by range specification. (One can specify a range with such entities as line numbers and character strings.)

One can modify the text with line editing commands such as COPY, SUBSTITUTE, and REPLACE. Or one can move about in the text by using the FIND and TYPE commands, for example, or by pressing the RETURN key.

Change mode lets users operate on such entities as characters, words, sentences, paragraphs, and lines. One can also work with strings of text or delete and move whole pages. EDT lets a user rede-
fine these entities to tailor them to specific applications, which can be as diverse as documentation and programming.

Change mode consists of a set of NOKEYPAD commands. Typing any of these commands lets a user perform useful functions. By typing FNDNXT, for example, one can find the next occurrence of a string of characters.

With VT52 and VT100 terminals, one can also use KEYPAD commands. The set of keypad keys, as well as several CTRL keys, lets the user enter any of the NOKEYPAD commands simply by pressing a key. Users can also redefine the function of these keys.

**The Keypad**

The keypad is a special set of keys to the right of the main keyboard. The following diagram represents the functions of the VT100 keypad; the VT52 keypad is nearly identical.

![VT100 Keypad Function Diagram](image)

**Figure 8-2 VT100 Keypad Functions**

Keypad functions allow the user to perform a variety of operations. Furthermore, he can change the function of any keypad key to meet his needs with the DEFINE command.

The commands in the keypad submode let users alter text or change the cursor position in the file. Some of the keypad functions let them advance or back up the cursor or move the cursor to the top or bottom of the text. One can also move the cursor any number of characters, words, lines, or pages at a time.
Support Facilities

Keypad keys let a user select a string of text and move it elsewhere in any of his files. One can even find the next occurrence of some text and delete or replace it. There is also a key to press for help messages.

A Sample Session with EDT
To begin an editing session with EDT, log in and type EDIT/EDT. A prompt appears to let you start the editor:

$ __ File:

Creating a File — If you want to create a file named TEST, type the following after the command prompt (the underlining indicates what you type):

$ __ File: TEST<CR>

EDT notifies you that no file with that name exists by responding:

Input file not found
[EOB]
*

[EOB] means that you are at the “End Of the Buffer” in the new file. (A “buffer” is the location in memory of the file you are working on.) The asterisk prompt indicates that you are now in line mode. When EDT is in this mode, you can edit your files by individual lines.

Entering Text in Line Mode — The first entry in your new file is an insertion. When you type “i” to insert, any text that you enter is indented two tab spaces:

1. *l<CR>

This is Line 1.<CR>
This is the second line.<CR>
Here is some text for<CR>
the new file named TEST.<CR>

↑Z

You need to type the CTRL/Z (↑Z) to save your insertions.

Range Specifications in Line Mode — There are various ways of expressing range specifications in line mode. Some examples follow:

1. Type the whole file.

     *tw<CR>

     1
     2
     3
     4

     [EOB]
     *

     This is Line 1.
     This is the second line.
     Here is some text for
     the new file named TEST.

     198
2. Type the second line.
   *2<CR>
   2 This is the second line.
   *

3. Type the rest of the file.
   *tr<CR>
   2 This is the second line.
   3 Here is some text for
   4 the new file named TEST.
   [EOB]
   *

4. Type every line in the file that contains the word “the.”
   *t all ‘the’<CR>
   2 This is the second line.
   4 the new file named TEST.

Deleting and Replacing Text — Range specifications are useful not only for displaying lines but also for manipulating text. The following examples show how you delete and replace text in your file.

Notice that you can use the /QUERY option (/Q) 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. You want to delete either line 2 or 3. Use /QUERY to read them first, since you are not sure which of the lines to delete.
   *D2:3/Q<CR>
   2 This is the second line.
   ?<CR>
   Please answer Y(es), N(o), Q uit, or A(ll)
   ?N
   3 Here is some text for
   ?Y<CR>
   1 line deleted
   4 the new file named TEST.
   *

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

1. *tw<CR>
   1 This is Line 1.
   2 This is the second line.
   4 the new file name TEST.
   [EOB]
   *
You can resequence the contents of the file with the following command:

1.  *res 1:4<CR>

EDT checks lines 1 through 4 and renumbers them in increments of 1.

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

   *re 3<CR>
   1 line deleted
   
   Here are two lines to replace Line 3.

   ↑Z

Notice that the REPLACE command deletes the line you specify and puts EDT in the insert level of line mode. Exiting with CTRL/Z saves this version of file TEST. The file now looks like this:

1.  *t w
    
    1
    This is Line 1.
    
    2
    This is the second line.
    
    3
    Here are two lines to replace Line 3.
    
    [EOB]

**Entering Change Mode** — If you are using a VT52 or VT100, the easiest way to edit a file is with keypad functions. You can reset the default mode for video terminals with the SET KEYPAD command. When you type an abbreviation for change mode (C, CH, or CHA, for example), you automatically enter the keypad submode of change mode:

   *CH<CR>

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

**Using the Keypad** — To move the cursor to the bottom of the buffer, press the SHIFT key and then the BOTTOM key. The file appears as shown:

This is Line 1.
This is the second line.
Here are two lines to replace Line 3.
[EOB]

200
Support Facilities

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

I am typing these lines<CR>
while EDT is in keypad mode.<CR>

Press the up-arrow key twice to move the cursor up two lines. The screen looks like this:

This is Line 1.
This is the second line.
Here are two lines
to replace Line 3.
I am typing these lines
while EDT is in keypad mode.

[EOB]

You can move a section of text about in the file with the CUT and PASTE commands. The following shows how to move the last two lines to the beginning of the file:

1. Mark the start of the lines by pressing the SELECT key.
2. Move the cursor to the end of the lines ([EOB]) by pressing the down arrow key twice.
3. Press the SHIFT and then the CUT key to insert the two lines into a paste buffer. The line will disappear from the screen.
4. Move the cursor to the top of the file by pressing the SHIFT key and then the TOP key.
5. Press the SHIFT key and then the PASTE key to place the contents of the paste buffer at the start of the file.

The file now looks like this:

I am typing these lines
while EDT is in keypad mode.
This is Line 1.
This is the second line.
Here are two lines
to replace Line 3.

[EOB]

You can continue pressing the SHIFT and PASTE keys as many times as you like to duplicate these two lines.
Returning to Command Level — To save the file and exit change mode, enter a CTRL/Z. This returns you to the asterisk prompt in line mode. Now type “EX” (for EXIT) after the prompt:

*EX<CR>

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

DBB1:[SMITH]TEST.;1 11 lines
$

If you prefer not to save this practice file, type “QUIT” instead:

*QUIT<CR>

QUIT returns you directly to the command level prompt.

SLP EDITOR
SLP is the batch-oriented editing program used for source file maintenance. SLP allows updating (deletion, replacement, addition) of lines in an existing file. Furthermore, SLP 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 SLP 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 SLP 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 5) can be used to create a SLP correction file that will change the old file into the new one.

**SLP Input and Output Files**

SLP 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**

SLP 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.
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**
SLP 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 8-2 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 8-2 (including itself).

**Table 8-2  SLP Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>First character of an SLP edit command</td>
</tr>
<tr>
<td>\</td>
<td>Suppress audit-trail processing</td>
</tr>
<tr>
<td>%</td>
<td>Re-enable audit-trail processing</td>
</tr>
<tr>
<td>@</td>
<td>Invoke an indirect file for SLP processing</td>
</tr>
<tr>
<td>/</td>
<td>Terminate the edit session and return to SLP command level</td>
</tr>
<tr>
<td>&lt;</td>
<td>Escape character</td>
</tr>
</tbody>
</table>

**General Form of the Edit Command**
The general form of the SLP command is as follows:

−locator1[,locator2]/audittrail/[;comment] inputline
  .
  .
  .

204
Support Facilities

where:

-(minus) specifies that this is an SLP edit command line

locator1 is 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 is 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/ is 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 is 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 is 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]

( . )

where:

string is 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 (type 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.

**DOCUMENT FORMATTING UTILITY**

**VAX-11 RUNOFF**

VAX-11 RUNOFF is a document formatter. A RUNOFF-processed document can be updated without extensive retyping because textual changes—made with the text editors—do not affect the basic design. The input to RUNOFF is a file containing the text of the document and the RUNOFF 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, some of which are illustrated by examples later in this section. 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 call flags which control such enhancements as underlining or **bolding** a word or letter.

**Filling and Justifying**

RUNOFF commands set left and right margins, so that the user may enter text without concern for line width or variable spacing between words. The RUNOFF 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. RUNOFF justifies the line by expanding the spaces between words to produce an even right margin.

**RUNOFF Default Modes**

When an input file is processed by RUNOFF, certain default actions are performed that do not depend upon command or flag entries for their execution. These actions are closely analogous to those performed during the preparation of a manually typed document.

RUNOFF default modes provide:
Support Facilities

- A standard typewriter page size of $8^{1/2} \times 11$
- Sequential page numbering for every page but the first
- Page width of 60 characters
- Single spacing
- Automatic tab settings for every eighth print position, starting with the ninth column (9, 17, 25, etc.)
- 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 establish the style and location of chapter headings and subheads. Other page formatting commands engage or disengage 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 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; printing any subtitles of designated words; determining the number of header levels (up to six) that the document will have are all 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, as with ragged right-hand margin, indenting a paragraph, skipping a number of lines, centering the text, underlining, hyphenation, and overstriking. Of course, different parts of the text may be formatted differently, and commands may be combined. To illustrate, a user has the option to have lists justified or to
have them with ragged margins.

For example:

.LM 5 .RM 58  Sets 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 output file.

.NJ  Disables justifying: lines are right ragged.

.BR  Causes a break: current line is output without being filled or justified.

.SK 2  Skips two blank lines.

.PG  Enables a .BREAK, then forces 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 text on the 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 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 and endnotes also have to be prepared in such a way that they 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, possibly 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 a element.
.DLE "(" ,LL ,")" Establishes a user-specified display format for lists: in this case, sequential, lower-case 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
14.3 King Lear
14.3.2 Tragic flaw — The definition of tragic flaw...

Miscellaneous Formatting
A number of useful RUNOFF commands helps the user to re-establish all default values, to add nonprinted comments to the source file, to gather externally located files into the input, to exert conditional control, and to set time and date.

For example:

.PROCESS the lines following only if /VARIANT:COMPLETE was given on the command line.

!appendix C is 200 pages

RUNOFF ignores comments.

.REQ "APNDXC.RNO"

Process 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.SKIO; Contact the author...

Allows commands and text in one line.

.ENDIF complete

Marks the end of a group of conditionally processed lines.

Flags
A flag is a special character (e.g., an ampersand) that is used to perform a specific operation (e.g., underlining). The specified operation is invoked when the character is recognized as a flag by RUNOFF. Certain special characters initially are recognized by default.

For example:

mis&spell

The UNDERLINE flag (&) underlines the next character, so that the word appears: misspell.
Support Facilities

fix#some#space  The SPACE flag (#) fixes one nonexpandable space wherever it occurs.

R-&D    The ACCEPT flag (-) prevents RUNOFF from misinterpreting the ampersand in R&D as an underline flag.

INDEX AND TABLE OF CONTENTS
RUNOFF has powerful facilities for creating indexes and tables of contents easily. There is a command to generate a one-column index. In addition, the TCX program generates two-column indexes, while the TOC program generates tables of contents. Both TCX and TOC create files that can be edited or can be processed by RUNOFF; this adds great flexibility to the preparation of indices and tables of contents.

For example:

.X Satire Creates an index entry for Satire. RUNOFF gives it the current page number.

.ENTRY Parody-see Satire Provides a cross reference to the index.

.DX Places the sorted and formatted one-column index in the output.

RUNNING THE RUNOFF PROGRAM
RUNOFF is initiated by entering the following command:

RUNOFF filespec <CR>

After processing the file, RUNOFF 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; 3-160, 4-1; 4-16" Prints only pages 3-11 through 3-16 and 4-1 through 4-16.

/DEBUG:echo Traces the operation of any RUNOFF commands defined by the parameter by echoing each execution in the output file.

/INDEX:drama.bix Creates an index file called drama.bix
Support Facilities

/CONTENTS: poems.btc
Creates an index file called poems.btc.

/OUTPUT:TT:
Puts the output on the terminal.

LINKER
The VAX/VMS linker accepts as input one or more native code object modules produced by an assembler or compiler. To write an application in modules, it is necessary to be able to link together the separately compiled modules. Linking consists of three basic operations:
1. Allocation of virtual memory addresses
2. Resolution of intermodule symbolic references
3. Initialization of the contents of a memory image

At the end of a linking operation, the program is assigned virtual memory addresses, has intermodule references resolved, and exists as an executable initialized entity in a disk-resident image file.

Virtual Memory Allocation
Language translators do not compute any addresses in the program. At the time of translation, the allocation of virtual address space is undecided. Each object module is relocatable in virtual memory. The reason that language translators cannot allocate virtual memory addresses is that a translator can see only one module at a time: it cannot know how modules interrelate. As a result, it is the linker's function to perform the memory allocation, reference resolution, and image initialization required to form one executable program from a number of object modules.

VAX/VMS language translators use the object language to describe a module to the linker. The output from a translator is an object module consisting of the binary code to be executed and object language records describing the module to the linker. The language translators define each object module as a number of separate areas called program sections. Some program sections contain data, others contain instructions. Some can be modified during execution, others cannot. Some are accessible to procedures in other modules, others are local to a module. When determining the virtual memory allocation of a program, the linker must consider the attributes of each program section. The linker groups program sections with the same attributes together in virtual memory.

Resolution Of Symbolic References
VAX language translators provide the ability to call external procedures by name. They permit the use of other external items such as literals and variables by name. External references have values that
are available only to the linker when all the input, e.g., modules and library procedures, is gathered together. The object language provides the ability for a language translator to describe to the linker the external items required by a module. The linker maintains a description of the items of each module that are available to other separately translated modules. In the object language, all of these external items are either references to global symbols or definitions of global symbols.

**Image Initialization**
VAX/VMS translators also permit the inclusion in an expression of a term that is defined externally. In many cases, the value of such an expression is required in the instruction stream of the procedure. The linker must place the value of an expression in line with the machine code generated by the language translator. The object language provides the capability for the linker to evaluate such an expression under the direction of the language translator and to store the result at the appropriate location in the code.

**Memory Allocation Algorithm**
Each program section of an object module has a set of attributes that the linker uses in allocating memory for the program section. The first attribute considered is whether the program section is concatenated or overlayed. Modular contributions to concatenated program sections are made in the order in which the linker processes the modules. Each module adds to its own contribution to the program section.

All contributions to overlayed program sections, on the other hand, start at the same address. The size of the final program section is the length of the longest contribution. The result of the handling of concatenated and overlayed attributes is a group of program sections, each having contributions from a number of modules.

The program sections are then sorted into image sections based on the remaining attributes. For example, all program sections that are read-only data are collected into one image section. They are allocated base addresses in alphabetic order by program section name. The result of grouping read-only program sections is a number of pages that are protected against modification when mapped into an executable image. The linker performs the same actions for other combinations of program section attributes to produce an image section for each combination of attributes.

Once the program sections are grouped into image sections, the linker assigns the virtual address space to each image section.
Overview of Linker Interface to Memory Management
The linker describes the virtual address space required for an image in such a way that the image activator function of memory management can place the image in a process virtual address space. When a user requests execution of an image, the image activator obtains a description of the image's virtual address requirements from the image file produced by the linker. The linker description of an image also provides memory management with the information needed to make shareable images globally available in physical memory, that is, to allow sharing of pages of physical memory.

The mechanism used to describe images to memory management is an image section descriptor. The linker creates an image section description (ISD) for each image section of a shareable or executable image. The header of an image contains the ISDs for the image. With the ISD, memory management can determine the following information about an image section:
1. The starting block number of the image section in the image file
2. The starting page in the process's virtual address space to which to map the image section
3. Characteristics of the image section, e.g., read-only, read/write
4. Additional control information

Using the information in the ISD, memory management sets the page table and other data structures used to bring process pages into physical memory and to allow sharing in physical memory.

File Input
The linker accepts the following types of files as input to a binding operation:
1. Object module files
2. Libraries of object modules
3. Shareable images files
4. Symbol tables from shareable images
5. Libraries of shareable images

Object Module Files
The linker requires as a minimum one object file as input to a binding operation. An object module contains four types of information:
1. Compiler program code and data
2. Descriptions of program code and data used by the linker in performing relocation and link-time computations
3. Identification of the object module and its history for use by the librarian and patch utilities
4. Description of the memory allocation requirements of the module

The input files containing object modules can be any of the following:
1. A file containing one object module
2. A file containing a concatenation of individual object modules
3. A file containing a library of cataloged object modules

The language processors produce both individual object modules and concatenations of modules. In addition, a file system utility can concatenate object modules into one file. The linker accesses object module files only by file name; there is no implicit extraction by module name from single-module or concatenated-module files.

Object Module Libraries
The librarian produces object module library files. Each library file contains a catalog of the object modules within it. The linker can access modules in such libraries either explicitly or implicitly.

Explicit extraction is performed on the basis of the name of a particular module in the file or by naming the library file and letting the linker extract any modules required to resolve undefined symbols.

Implicit access to object module libraries occurs after all explicitly named input modules have been extracted.

Shareable Image Files
A shareable image is an image that has been partially bound to the linker. It cannot execute, however, until it is bound with another object module. When the linker receives an object module that is to become a shareable image, it resolves references that are internal to the module; it cannot resolve the remaining references until the shareable image is bound with an object module. The result of that binding operation is an executable image. To handle shareable images, the linker requires the description of both the partially linked image and the symbols defined in the modules that form the image.

Shareable Image Symbol Tables
When the linker produces an image file, it appends the symbol table to the file. The symbol table produced by the linker has the same form as an object module. That is, it defines those symbols available to object modules that are outside the set of object modules that produced the shareable image. Such symbols are called universal.
Shareable Image Libraries
A shareable image library is a cataloged collection of shareable images and their associated symbol tables. Shareable images are in all other respects analogous to object module libraries. Inclusion of a shareable image as input to the linker can be either explicit or implicit. Like object modules, shareable images have names that are separate from the name of the file that contains them.

Linker Output
The linker produces three different types of images.
1. Executable images
2. Shareable images
3. System images

Executable images are the most common. As the name suggests, an executable image is the type run in response to a command to the command interpreter. The second type, shareable images, is intended for use at link time and, potentially, at run time. At link time, a shareable image can be linked with object modules to produce an executable image. The same shareable image can be shared when executable images bound to it run. System images are intended for stand-alone operations on the hardware. It is assumed that they do their own memory management, if any, and are not run under the VAX/VMS operating system.

The major differences between system images and other images produced by the linker are that system images usually do not have any image header and have no image sections. Memory is allocated to the program sections in alphabetic order without any consideration of their attributes.

Examples of system images are the VAX/VMS operating system and a stand-alone memory diagnostic.

Executable images and shareable images are quite similar. The image file format is almost identical. The virtual memory allocation algorithm is almost the same. The DCL command to produce either is the same with the exception of one or two qualifiers. However, the following differences do exist.

1. Shareable images are not runnable by themselves. They are intended for use by many executable images.
2. Shareable images contain global symbol tables, whereas executable images normally do not unless the /DEBUG qualifier is used at link time.
3. Shareable images cannot be linked with the debugger.
4. Shareable images can be linked together; executable images cannot.

**The LINK Command**
The DCL LINK command provides the interface between the user and the linker. When the user requests the linking of object modules, the command interpreter receives the command and activates the linker.

The general format of the LINK command is:

\[
\text{LINK} \{\text{qualifiers}\} \text{infilen}\{\text{qualifiers}\} \\
\quad \{\ldots.\text{infilen}\{\text{qualifiers}\}\} \\
\]

The LINK command is also described, with examples, in Chapter 5, DIGITAL Command Language.

**VAX/VMS DEBUG FACILITY**
The VAX-11 DEBUG program is a language-independent, interactive, symbolic debugger that works with programs written in most of the native mode languages supported by VAX/VMS. Current languages for which the debugger works are: VAX-11 FORTRAN, VAX-11 BASIC, VAX-11 COBOL, VAX-11 BLISS-32, and the VAX-11 MACRO assembly language. After linking with the user program, the DEBUG facility is operative in the language of the first module of the image file. If it is necessary to alter the language for a later module, the SET LANGUAGE command may be used.

DEBUG enables dynamic examination and modification of the contents of memory locations, which is useful in fixing programs with errors. 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:

\[
\text{\$LINK}/\text{DEBUG PROG1} \\
\]

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:

\[
\text{\$ RUN PROG1} \\
\]

216
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

DEBUG (Identification Message): VAX/VMS DEBUG V2.0 (date)

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.

(For native mode languages such as COBOL, which are supported in several different standards by VAX/VMS, the debugger will automatically recognize and adapt to the version in which the process is written.)

**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 or the STEP command.
5. Determine the location of breakpoints, tracepoints, and watchpoints by means of the commands SHOW BREAK, SHOW TRACE, SHOW WATCH, respectively.
6. Erase breakpoints, tracepoints, and watchpoints in the program, through use of the CANCEL command.
7. Display the contents of memory locations, by using the EXAMINE command.
8. Change the value of the contents of memory locations, by using the DEPOSIT command.
9. Obtain the value of an expression or the current address of a symbol, or express a numeric value in a different radix, by using the EVALUATE command.
10. Call a subroutine at DEBUG time, by means of the CALL command.
11. Change values of parameters for LANGUAGE, SCOPE, MODE, TYPE.
12. Specify an arbitrary file name for the DEBUG log file by means of the SET LOG command.
13. Control DEBUG I/O at debug time, via the SET OUTPUT command. This includes normal terminal output, log file output, and command file verification.
14. 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.
15. Instruct DEBUG to take commands from a specified file by means of @Filespec.

**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 8-3 summarizes the value that may be used for keyword and parameter.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Parameter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANGUAGE</td>
<td>Language-name</td>
<td>Specifies the language characteristics to be used by DEBUG</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBG&gt; SET LANGUAGE FORTRAN</td>
</tr>
<tr>
<td>BREAK</td>
<td>address</td>
<td>Sets a breakpoint at a location in the program; optionally specifies commands to be performed at that point</td>
</tr>
<tr>
<td></td>
<td>[Do(DERBug. commands)]</td>
<td>DBG&gt; SET BREAK 500 DO(EXAMINE K)</td>
</tr>
</tbody>
</table>

218
### Table 8-3 SET Command Summary

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Parameter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRACE</td>
<td>address</td>
<td>Lets the user follow the program’s execution sequence, to ensure that instructions are being executed in proper order</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBG&gt; SET TRACE % LINE 25 (see note below)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBG&gt; SET TRACE % LABEL 99 (see note below)</td>
</tr>
<tr>
<td>WATCH</td>
<td>address</td>
<td>Sets a watchpoint at the specified address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBG&gt; SET WATCH ANYSYMB</td>
</tr>
<tr>
<td>SCOPE</td>
<td>A list whose elements may be: pathname nonnegative integer &quot;&quot;</td>
<td>Establishes an ordered list which the DEBUGGER traverses in order to find the definition of a specified symbol</td>
</tr>
<tr>
<td>MODE</td>
<td>Radix: DECIMAL HEXADECIMAL OCTAL</td>
<td>Alters the defaults used by DEBUG for radix and symbolic representation of addresses</td>
</tr>
<tr>
<td>TYPE</td>
<td>BYTE WORD LONG ASCII:length INSTRUCTION</td>
<td>Establishes a data type to be used to interpret those addresses for which DEBUG cannot infer a type from the data definition</td>
</tr>
</tbody>
</table>

**NOTE**

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

Table 8-3 SET Command Summary con’t

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Parameter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOG</td>
<td>file name</td>
<td>Specifies that the DEBUG log can be called something other than the default name, “DEBUG.LOG”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBG&gt; SET LOG NEW.LOG</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>[NO]LOG,</td>
<td>Tailors output modes of DEBUG to suit particular applications</td>
</tr>
<tr>
<td></td>
<td>[NO]TERMINAL,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[NO]VERIFY</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DBG&gt; SET OUTPUT NOLOG VERIFY TERMINAL</td>
</tr>
</tbody>
</table>

EVALUATE Command

The EVALUATE command allows the user to check the value of an expression or the definition of a symbol, or express a numerical value in a different radix. This command has the form:

EVALUATE element

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:

\[ \text{CALL s(a,\ldots,a)} \]

where

\[ s \quad \text{subroutine name} \]
\[ a,\ldots,a \quad \text{actual arguments} \]

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

\[ \text{SHOW keyword} \]

---

**Table 8-4  SHOW Command Summary**

<table>
<thead>
<tr>
<th><strong>Keyword</strong></th>
<th><strong>Function</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAK</td>
<td>Displays, in symbolic form, the location of each breakpoint in the program</td>
</tr>
<tr>
<td>TRACE</td>
<td>Displays, in symbolic form, the location of each tracepoint in the program</td>
</tr>
<tr>
<td>WATCH</td>
<td>Displays, in symbolic form, the location of each watchpoint in the program</td>
</tr>
<tr>
<td>MODE</td>
<td>Displays the current modes</td>
</tr>
<tr>
<td>SCOPE</td>
<td>Displays the current ordered list of scopes</td>
</tr>
<tr>
<td>TYPE</td>
<td>Displays current type</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>Displays output attributes</td>
</tr>
</tbody>
</table>

**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:

\[ \text{CANCEL keyword [parameter]} \]

Table 8-5 lists the keywords and parameters that can be specified with CANCEL.
Support Facilities

Table 8-5  CANCEL Command Summary

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Parameter</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAK</td>
<td>address</td>
<td>Eliminates the break-point at the specified location</td>
</tr>
<tr>
<td>TRACE</td>
<td>address</td>
<td>Eliminates the trace-point at the specified location</td>
</tr>
<tr>
<td>WATCH</td>
<td>address</td>
<td>Cancels the watch-point at the specified location</td>
</tr>
<tr>
<td>MODE</td>
<td>none</td>
<td>Cancels the current mode for radix, length, or data type and restores the default value</td>
</tr>
<tr>
<td>SCOPE</td>
<td>none</td>
<td>Restores the default value of SCOPE</td>
</tr>
<tr>
<td>ALL</td>
<td>none</td>
<td>Cancels all parameters previously set for DEBUG</td>
</tr>
</tbody>
</table>

**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 500
```

DEBUG resumes program execution at virtual address 500.

**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 how many instructions or statements are to be executed in the user program. The user may specify instruction or statement (for languages that support 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 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.

EXIT Command
To terminate the DEGUG session and return to DCL level, use the EXIT command.

COMMON RUN TIME PROCEDURE LIBRARY
The VAX-11 Common Run Time Procedure Library (RTL) is composed of a set of general and language-specific VAX procedures which establish a common run-time environment for all user programs written in any native mode language. Because all of the language support procedures follow the same programming standards and make non-conflicting assumptions about the execution environment, 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-11 Run Time Library is constructed as a separate shareable image which interacts with the rest of the operating system via entry point vectors. 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 small portion of the Run Time Library is not shareable.)

The VAX-11 Run Time Library consists of one or more shareable image sections. Each procedure entry point in the shareable image section 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 Run Time binding also permits a new release of the library to be installed without requiring that user images be relinked.
Support Facilities

The VAX-11 Run Time Library comprises several sections which are grouped by function or calling sequence. They are:

- a resource allocation section (virtual memory, logical unit number, and event flags)
- a condition handling section (signaling exception conditions and declaring condition handlers)
- a general utility section (data type conversions)
- a mathematical section (single and double precision trigonometric, logarithmic, and exponential functions)
- a language-independent support section (error handling and record management services support functions)
- language-specific support sections (file handling support functions)
- a string handling section (static and dynamic string functions)

The Run Time library is designed as a set of modular re-entrant procedures.

Resource Allocation Section (LIB$)
The Resource Allocation Section includes all procedures which 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-11 condition handling facility is a collection of library procedures and system services which 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.
Support Facilities

General Utility (LIB$)
General utility procedures are not mandatory in order to use the rest of the library successfully. They are provided for the convenience of the user only. General utility procedures include outputting a record to a logical device (SYS$OUTPUT).

Mathematical Functions (MTH$)
The mathematical library consists of standard procedures to perform common mathematical functions, such as taking the sine of an angle. The standard entry points have one or two call-by-reference input parameters and a single function value. Some frequently used procedures also have call-by-value Jump to Subroutine (JSB) entry points.

Language-Independent Support (OTS$)
The language support libraries support the code generated in-line by compilers. As such, most of the procedures are called implicitly as a consequence of a language construct specified by the user, rather than being called explicitly by the user with a CALL statement. Those language support procedures which are independent of higher level language use the facility prefix OTS$.

Language-Specific Support (FOR$, BAS$)
Each of the language support libraries is composed of five principal types of procedures:
1. I/O processing procedures
2. Language-independent initialization and termination
3. System procedures
4. Compiled-code support procedures
5. Error and exception-condition processing procedures

String Processing (STR$)
The string processing procedures allocate and deallocate dynamic strings and perform a number of useful string functions on any class of VAX strings.

System Procedures
VAX-11 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.
Compiled-Code Support Procedures
These routines complement the compiled code by performing operations too complicated or too cumbersome to perform directly with in-line code. Thus, the language support libraries support the code generated by the compiler. For example, division of complex numbers is performed by a library procedure.

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$). However, the math library and the language support libraries indicate most errors by CALLing the VAX-11 LIB$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.
CHAPTER OVERVIEW

Communications among computers is made quite easy through the facilities of DIGITAL's DECnet architecture—which enables the linking of DIGITAL operating systems and hardware—and through the range of protocol emulators—which permit DIGITAL computers to work with computers from other vendors. The current phase of DECnet-VAX is demonstrated through sample communications; description of emulators complete the chapter.

Topics include:

- Description of a DECnet Network
- Command Language, FORTRAN, MACRO
- Task Messages
- Programming Procedures
- Protocol Emulators
CHAPTER 9

COMMUNICATIONS AND INTERNETS

INTRODUCTION
DIGITAL computers can communicate with others, both near and remote, over a network. By using protocol emulators (Internets), they can communicate with computers from other suppliers.

In order to avoid mixing terminology when discussing networks—which may include several different operating systems—we will use the word task to mean job, image, or process.

DECnet is the family of products that allow DIGITAL systems to participate in a cooperative multiprocessing environment known as a network. A network is a configuration of two or more independent computer systems, called nodes, linked together to facilitate remote communications, share resources, and perform distributed processing. Network nodes are not all required to run on the same type of operating system. 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 cable wires laid from one node to another, or they can be temporary connections that change with each use, such as dialed-up telephone calls.

In a network of DECnet nodes, several tasks can use the same physical link to exchange data. That is, more than one data path can be handled simultaneously by a physical link. This data path is known as a logical link.

With DECnet, a variety of computer networks can be implemented. 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 data bases on one or two large host systems in the network. Figure 9-1 illustrates such a network.
• Resource-Sharing Networks. These networks exist to permit sharing expensive computer resources among several computer systems. Shared resources not only include peripherals such as mass storage devices, but they can also include logical entities, such as a centralized data base which is made available to other systems in the network. Such networks are often characterized by the concentration of high-performance peripherals, extensive data bases, and large programs on one or two host systems in the network, while the satellite systems have less expensive peripherals and smaller programs. Figure 9-2 illustrates a resource-sharing network.

Figure 9-2 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 may have specific geometries (star, ring, hierarchy), but often have no regular arrangement of links and nodes. Such 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 data bases distributed throughout the network. Figures 9-3 and 9-4 illustrate two such networks.

Figure 9-3  Typical Manufacturing Network

Figure 9-4  Computational Network
When DECnet is used to connect heterogeneous systems, each node of the network has both common DECnet attributes and system-specific attributes. Programs executing in native mode can access 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 programs to the correct target node in situations where the source node and target node do not share a link.
- **Network Command Terminal**: A terminal on one VAX system can appear to be connected to another VAX system in the network.
- **Down-Line System Loading**: Initial load images for RSX-11S systems in the network can be stored on the host VAX system, and be loaded into adjacent PDP-11 systems configured for the RSX-11S operating system.
- **Down-Line Task Loading**: Program images for RSX-11S systems in the network can be stored on the host VAX-11 system, and loaded on request into PDP-11 systems configured for the RSX-11S operating system.
- **Down-Line 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.

*The goal of DECnet-VAX is to provide a network capability that is extremely easy to use. Task-to-task communication and file access between systems is transparent; these intersystem facilities appear to be no different from the intrasystem interprocess communication and file access facilities.*

**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.
Its functional components 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 a level of abstraction and 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. For DECnet-VAX, DDCMP is incorporated into the microprocessor of the communications interface.

**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. DECnet
components can be replaced if better communications hardware becomes available or if technical innovations in networking occur. A DECnet network can accommodate the change of a function from software into hardware.

**DECnet-VAX FEATURES**
The capabilities offered the DECnet-VAX programmer and terminal user extend through a wide range of network functions.

**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/VMS 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 as part of its regular 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.

**Remote Terminal**
The remote terminal facility allows a local terminal to operate as if it were physically connected to a remote computer.

**Intertask Communications**
Any native-mode language programmer can write programs that perform intertask (interprocess) communication. Intertask 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.

Intertask communication routines can be coded using one of two methods, transparent or nontransparent.

**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. 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 a little 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 unsolicited messages to or from another process or task. Unsolicited message traffic is essentially no different from solicited 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. In DECnet-VAX, an interrupt message is written to a mailbox that a process supplies for that purpose.

In DECnet-VAX, 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 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 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"loginstring"::filename.filetype;version
```

where:

- **nodename**
  - Node name is a 1- to 6-character name (numeric or upper case alphabets) identifying the remote network node. This can be followed by an optional quoted string used for logging in at the remote node.

- **loginstring**
  - If the “loginstring” is omitted, default login information comes from an entry (for the remote node) in the local configuration data base. Thus, by using the loginstring, the user overrides the default login information.

  Use one of the following formats for the loginstring:

  - “username password”
  - “username password accountname”

  The double colon (::) following the nodename separates the nodename from the file specifier.

- **filename**
- **filetype**
- **version**

See the chapter on the VAX 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.

Use the following format 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 supports a limited set of VAX/VMS (DCL) commands. They are:

APPEND
ASSIGN
COPY
DEASSIGN
DEFINE
DELETE
DIRECTORY
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 is supported by DECnet-VAX, including the management of sequential, relative, and indexed file organizations. 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.
Communications and Internets

.TITLE DEM0I - 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::LP0:
$DEMO1

.SBTTL CONTROL BLOCK AND BUFFER STORAGE

IMPURE NOEXE.LONG

Define the source file FAB and RAB control blocks.

SRC_FAB:
$FAB FAC=GET,-1; GET ACCESS
FOP=SQO,-1; SEQUENTIAL ONLY
FNA=SRC_NAM,-1; ADDRESS OF FILENAME STRING
FNS=SRC_NAM_SIZ; SIZE OF FILENAME STRING

SRC_RAB:
$RAB FAB=SRC_FAB,-1; ADDRESS OF FAB
RAC=SEQ,-1; SEQUENTIAL RECORD ACCESS
UBF=BUFFER,-1; 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>-1; PUT (WRITE) ACCESS
FOP=SQO,-1; SEQUENTIAL ONLY
FNA=DST_NAM,-1
FNS=DST_NAM_SIZ,
ORG=SEQ,-1; SEQUENTIAL FILE (DEFAULT)
RFM=VAR,-1; VARIABLE LENGTH RECORDS
RAT=CR; PRECEDE LINE BY LF, FOLLOWED BY CR

DST_RAB:
$RAB FAB=DST_FAB,-1
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

.MAINLINE

.CODE NOWRT

.ENTRY DEM01, FM<>
Communications and Internets

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>&1FFFF>
       BEQL  20$
       MOVL RAB$L_UBF(R7),RAB$L_RBF(R9)
       MOVW RAB$W_RSZ(R7),RAB$W_RSZ(R9)
       BLBS     RAB=(R9)
       BRB    R0,10$

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
       END      ; display on error condition
       DEMO1

239
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 cooperate 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:

\[
\text{nodename}::\text{loginstring}::\text{"TASK=}\text{taskid}"\]

where:
- `nodename` (Refer to DIGITAL Command Language File Handling section of this chapter.)
- `loginstring` Use one of the following formats for loginstring:
  - "username password"
  - "username password accountname"
- `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.

**VAX-11 FORTRAN Intertask Communication Example**

The following programs illustrate VAX-11 FORTRAN intertask communications.

**Source Task Code**

```
PROGRAM DEMO2.FOR

This program prompts the user for a request, communicates
with a remote task to obtain the requested data, and displays
the answer for the user. The remote task is referenced by
the logical name TASK. If the remote task is named DEMO3.EXE
at node TULSA, the following procedure is used to run the
two programs:

$ DEFINE TASK TULSA::"""TASK=DEMO3"""

$ RUN DEMO2

LOGICAL*1 CCDE(4),BUFFER(20)

100 FORMAT ("$nter request code: ',4A1)

200 FORMAT (4A1)

300 FORMAT (Q,20A1)

400 FORMAT ('0Stock number for code ',4A1 is:,20A1)

C C

Request the remote task to be run and establish a logical
link into it.

OPEN (UNIT=1,NAME='TASK,ACCESS='SEQUENTIAL',FORM='FORMATTED')

C C

Prompt the user for a request code, send the code to the
remote task, read the reply from the remote task, and display it to the user.
Repeat the cycle until the user enters 'Exit' as his request code.
```
TARGET TASK CODE

PROGRAM DEMO3.FOR

This is the companion task for DEMO2. For each request it receives from the remote task it replies with a 1- to 20-character response. This program does not know the name of the requesting task. To complete the logical link with its initiator, it opens the 'file' specified by the logical name SYS$NET.

LOGICAL*1 CODE(4),BUFFER(20)

100 FORMAT (4A1)
200 FORMAT (20A1)

Establish a communication path with the remote task.

OPEN (UNIT=1,NAME='SYS$NET,ACCESS='SEQUENTIAL',FORM='FORMATTED')

Process requests until end-of-file encountered.

10 READ 100,END=20,CODE

Perform appropriate processing to obtain result to transmit back to the requesting task.

WRITE (1,200) (BUFFER(K),K=1,NCHAR)
GOTO 10

C C C
C C C
C C C
C C C
20 CLOSE (UNIT=1)
END

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 ($AS-
SIGN). 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 JOS$ 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 be connected to. 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)

$INPUT—Read a Message

$OUTPUT—Write a Message

$DASSGN—Disconnect the Logical Link

**MACRO NONTRANSIENT INTERTASK COMMUNICATION**
As does 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 completes.

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: solicited messages and unsolicited messages.

Solicited Messages
A solicited message is a data message sent by one task, and expected by the cooperating task. This 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 solicited message in nontransparent intertask communications is the same as a data message sent in transparent communications.

Unsolicited Messages
All other messages received by a task employing a nontransparent interface are classified as unsolicited 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 places a received unsolicited message into the mailbox associated with the channel representing the logical link to which the unsolicited 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 unsolicited message from the mailbox using the $QIO(IO$READ$BLK) 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 places any unsolicited 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.

The IOSB (if specified) will also contain the connect request result:

SS$ _NORMAL — Connect accept

SS$ _REJECT — Connect reject

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$!O$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 ($QIO(\!O$\_WRITEVBLK) or $OUTPUT$), the receiving task must issue a corresponding receive ($QIO(\!O$\_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 $QIO(\!O$\_WRITEBVLK$!O$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 notification.

**Disconnecting or Aborting the Logical Link**

Either task can disconnect or abort the logical link by:

- Calling $QIO(\!O$\_DEACCESS$!O$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 $QIO(\!O$\_DEACCESS$!O$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—Accept a Logical Link Connection Request

$QIO—Reject a Logical Link Connection Request

$QIO—Send a Message to a Remote Task

$QIO—Receive a Message from a Remote Task

$INPUT—Read a Message

$QIO—Send an Interrupt Message to a Remote Task

$QIO—Synchronously Disconnect the Logical Link

$QIO—Abort a Logical Link

$QIO—Declare a Network Name

$DASSGN—Disconnect the Logical Link

PROTOCOL EMULATORS (INTERNETS)
VAX/VMS supports a number of software emulators that enable and promote coexistence between the VAX family and products supplied by other vendors. In this way, VAX computers become even more flexible, particularly in extending existing mainframe facilities to include powerful minicomputer data processing.

VAX-11 2780/3780 Protocol Emulator
This product provides the VAX/VMS user with a mechanism for transferring files between the VAX system and another system equipped to handle IBM 2780 or 3780 communications protocols. It
does this by emulating the synchronous line protocol used by a 2780 or 3780 Remote Batch Terminal.

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 forms to control translation schemes (records can be padded with spaces to card images before transmission), translation to and from EBCDIC, and BSC 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
• BSC Transparency
• 3780 Space Compression

All of the above features are supported on a simultaneous, multiligne 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 baud per line.

**MUX200/VAX Multiterminal Emulator**

MUX200/VAX 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.

MUX200/VAX 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.

MUX200/VAX 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 baud.

MUX200/VAX enables several users to 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 9-5 for a schematic of the MUX200 use.
CHAPTER OVERVIEW

A wide range of system services is incorporated into VAX/VMS 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 Names
- I/O Services
- Timer and Time Services
- Exception Condition Services
- Process Control Services
- Memory Management Services
- Change Mode Services
CHAPTER 10
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 native mode 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 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 on to 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 subprocess.

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 on to 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 any user to control his subprocesses, allow a user with group privilege to affect processes in his group, and give a user 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

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 VAX/VMS 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
System Services

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 can be used to coordinate processes on various 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 explicitly "associate" with the cluster.

The range of event flag numbers and the clusters to which they belong are summarized in Table 10-1.

### Table 10-1 Summary of Event Flag and Cluster Numbers

<table>
<thead>
<tr>
<th>Cluster Number</th>
<th>Event Flag Numbers</th>
<th>Description</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0-31</td>
<td>Process-local event flag clusters for general use</td>
<td>Event flags 24 through 31 are reserved for system use.</td>
</tr>
<tr>
<td>1</td>
<td>32-63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>64-95</td>
<td>Assignable common event flag cluster</td>
<td>Must be associated before use</td>
</tr>
<tr>
<td>3</td>
<td>96-127</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

257
Listed below are the event flag system services.

**Associate Common Event Flag Cluster—$ASCEFC**

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.

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.

The following example shows how a process might create a common event flag cluster named COMMON-CLUSTER.

```
CLUSTER ASCID /COMMON-CLUSTER/ ; CLUSTER NAME

$ASCEFC_S EFN=’#65’, NAME=CLUSTER ; CREATE ; CLUSTER
```

**Disassociate Common Event Flag Cluster—$DACEFC**

The Disassociate Common Event 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: ASCID /COMMON-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 99 is equivalent to bit 3.

SHARE: .ASCID /COMMON CLUSTER/;CLUSTER NAME

```
$ASCEFC_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
```
System Services

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 them is 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; otherwise, 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.

```
$QIO_S
BSBW
$QIO_S
BSBW
$WFLAND_S
BSBW

; ISSUE FIRST QUEUE I/O REQUEST
; CHECK FOR ERROR
; ISSUE SECOND I/O REQUEST
; CHECK FOR ERROR
; WAIT UNTIL BOTH COMPLETE
; CHECK FOR ERROR

; CONTINUE EXECUTION
```

The MASK argument specifies which flags in the cluster are to be awaited: the first and second. The EFN argument specifies any flag number in the cluster containing flags for which you are waiting.

ASYNCHRONOUS SYSTEM TRAP (AST) SERVICES
Various system services allow a process to request that it be interrupt-ed 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 option-al 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)
System Services

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 register (AP) points to an argument list that has the format:

<table>
<thead>
<tr>
<th>31</th>
<th>8 7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>5</td>
</tr>
</tbody>
</table>

AST PARAMETER

- R0
- R1
- PC
- PSL
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.
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.

- 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.)
System Services

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, let us examine in some detail the nature and use of logical names themselves, and of some software structures known as logical name tables.

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>
<thead>
<tr>
<th>Table</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>2</td>
</tr>
<tr>
<td>Group</td>
<td>1</td>
</tr>
<tr>
<td>System</td>
<td>0</td>
</tr>
</tbody>
</table>

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 10-2 illustrates a user process logical name table.

This process logical name table equates the logical name TERMINAL to the specific terminal TTA2:.. INFFILE and OUTFILE are equated to
disk file specifications: these logical names were created from supervisor mode.

**Table 10-2  Sample Process Logical Name Table (Group=200)**

<table>
<thead>
<tr>
<th>Logical Name</th>
<th>Equivalence Name</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMINAL</td>
<td>TTA2:</td>
<td>User</td>
</tr>
<tr>
<td>INFIL</td>
<td>DM1:[HIGGINS]TEST.DAT</td>
<td>Supervisor</td>
</tr>
<tr>
<td>OUTFILE</td>
<td>DM1:[HIGGINS]TEST.OUT</td>
<td>Supervisor</td>
</tr>
</tbody>
</table>

The group logical name table contains names that cooperating processes in the same group can use, and the user must have special privilege to place a name in the group logical name table. Table 10-3 illustrates a sample group logical name table.

**Table 10-3  Sample Group Logical Name Table**

<table>
<thead>
<tr>
<th>Logical Name</th>
<th>Equivalence Name</th>
<th>Group Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>TERMINAL</td>
<td>TTA1:</td>
<td>100</td>
</tr>
<tr>
<td>MAILBOX</td>
<td>MB3:</td>
<td>200</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>TERMINAL</td>
<td>200</td>
</tr>
<tr>
<td>TERMINAL</td>
<td>TTA3:</td>
<td>300</td>
</tr>
</tbody>
</table>

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 10-4 illustrates a system logical name table.

**Table 10-4  Sample System Logical Name Table**

<table>
<thead>
<tr>
<th>Logical Name</th>
<th>Equivalence Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSSLIBRARY</td>
<td>DBA0:[SYSLIB]</td>
</tr>
<tr>
<td>SYSSYSTEM</td>
<td>DBA0:[SYSEXEC]</td>
</tr>
</tbody>
</table>

The system logical name table contains the default physical device names for all processes in the system. SYSSLIBRARY and SYSSYSTEM provide logical names for all users to refer to the device(s) containing system files.

And now we can consider the VAX/VMS operating system logical name services, 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:. 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
VAX/VMS provides the user 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

$ASSIGN_S DEVMAN=TTNAME,CHAN=TTCHAN
```
Deassign I/O Channel—$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.
System Services

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.
System Services

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/
DEVDESC:
.LONG 64
.LONG DEVDESC+8
.BLK8 64
TAPECHAN:
.BLKW 1

$ALLOC_S_DEVNAME=LOGDEV. PHYLEN=DEVDESC=DEVDESC,-
PHYBUF=DEVDESC

ERROR
$ASSIGN_S BSBW DEVNAME=DEVDESC,CHAN=TAPECHAN ;ASSIGN CHANNEL
$ASSIGN_S BSBW ERROR ;CONTINUE WITH I/O

$DASSIGN_S BSBW ERROR ;DEASSIGN CHANNEL
$DALLOC_S DEVNAME=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.
System Services

- 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.

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.

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.
System Services

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.
System Services

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 on to 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 log-on.

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. 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 EXECUTE 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=ACE
```

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:
## System Services

### Process Hibernation and Suspension

**Hibernation**

- Can only hibernate self
- Reversed by $WAKE$ system service
- Interruptible; can receive ASTs
- Can wake self
- Can schedule wakeup at an absolute time or at a fixed time interval
- Hibernate/wake complete quickly; require little system overhead

**Suspension**

- Can suspend self or another process, depending on privilege
- Reversed by $RESUME$ system service
- Noninterruptible; cannot receive ASTs
- Cannot cause self to resume
- Cannot schedule resumption
- 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 PROCNAME=ORIONDESC
BBBW ERROR

: WAKE ORION
```

### 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.
System Services

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)

For an example of canceling a wakeup, refer to “Timer and Time Conversion Services.”

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—$FORC鑫
The Force Exit system service causes an Exit ($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)
System Services

In the following example, a call to $FORCEX causes the image executing in the process named SMITH to exit.

PROGNAME: .ASCI/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:

.LONG 0 ;EXIT CONTROL BLOCK
.LONG EXITRTN ;SYSTEM USES THIS FOR pointer
.LONG 1 ;ADDRESS OF EXIT HANDLER
.LONG STATUS ;NUMBER OF ARGS FOR HANDLER
.BLKL 1 ;ADDRESS TO RECEIVE STATUS CODE
.BLKL 1 ;STATUS CODE FROM $EXIT

PEGASUS:

.WORD $DCLEXH_S
.WORD $EXITRTN
.BBLK=EXITBLOCK
.ERROR

.RET ;ENTRY MASK FOR PEGASUS

ENTRY MASK FOR PEGASUS

EXITRTN:

.WORD $DCLEXH_S
.WORD $EXITRTN
.RET ;END OF MAIN ROUTINE
.CMPL STATUS,#SS$_NORMAL
.BEQL 10$ ;ENTRY MASK
.RET ;NORMAL EXIT?
.RET ;YES, FINISH
.RET ;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.
System Services

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:

```
SPACE:
    .BLKL 2 ;RETURN START AND END OF NEW PAGES

$EXPREG=S PAGCNT=#4,RETADR=SPACE,REGION=#0 ;GET 4 PAGES
```

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
System Services

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.
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:

DIPSY: DESCRIPTOR <DIPSY> :NAME DESCRIPTOR

$SETPRN $PRCNAM=DIPSY

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:

TIME: .BLKQ 1 :BUFFER FOR TIME

$GETTIME_S TIMADR=TIME :GET TIME

The string returned by the service in the buffer ATIMENOW has the format:

dd-mmm-yyyy hh:mm:ss.cc

where 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
System Services

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.

```
ATIMENOW:
.LONG 23
.LONG ATIME NOW+8
.BLKB
$ASCTIM_S TIMBUF=ATIMENOW
;GET CURRENT 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.

```
ANOON: .ASCID /-- 12:00:00.00/
BNOON: .BLKQ 1

$BINTIM_S TIMBUF=ANOON,TIMEADR=BNOON ;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—$SCANTIM
The Cancel Timer Request system service cancels all or a selected subset of the Set Timer requests previously issued by the current
image executing in a process. Cancellation is based on the request identification specified in the Set Timer ($SETIMR) system service. If more than one timer request was given with the same request identification, they are all canceled.

**Schedule Wakeup—$SCHDWK**
The Schedule Wakeup system service schedules the awakening of a process that has placed itself in a state of hibernation with the Hibernate ($SHBER) 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.
System 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—$DCLCMH
Declare Change Mode or Compatibility Mode Handler ($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:

```
ATENMIN: DESCRIPTOR <000:10:00.00> ;ASCII TEN MINUTES
BTENMIN: .BLKQ 1 ;BUFFER FOR BINARY TEN MINUTES

$BINTIM_S TIMBUF=ATENMIN,TIMADR=BTENMIN ;CONVERT TIME
```

- 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.

**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 page frames in memory or I/O space.

Creating a disk file section 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—$CMKRN
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.
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-11 Virtual Memory System, complete I/O services are provided for handling, controlling, and queueing I/O needs or requests. A powerful set of routines of VAX/VMS, called the Record Management Services (RMS), 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 11
INPUT/OUTPUT SERVICES

INTRODUCTION
VAX/VMS supports a wide variety of input and output devices, including disks, magnetic tapes, lineprinters, and card readers, and 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 as issued and are subsequently processed according to the relative priority of the processes which issued them.

RMS and QIO
I/O requests can be handled indirectly through the use of an established set of procedures, such as VAX-11 RMS, or they can be interfaced directly to an I/O driver by means of a QIO request. The principal feature of VAX-11 RMS is its ease of use and device independence. Generally VAX-11 RMS is used for I/O requests to mass storage devices, while the more direct—and complicated—QIO request is for terminals, special devices (e.g., graphics equipment), and highly specialized formatting.

Figure 11-1 represents an overview of the major I/O processing system components and user relationships.
Figure 11-1 User Interfaces To I/O Services
PROGRAMMING INTERFACES
The I/O programming tools are: the record management system (RMS)—for general purpose file and record processing—and the I/O system services—for direct I/O processing. Table 11-1 summarizes the programming interfaces.

<table>
<thead>
<tr>
<th>Method</th>
<th>Program Interface</th>
<th>I/O Components</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record I/O</td>
<td>RMS requests</td>
<td>RMS, ACP and Driver</td>
<td>Use Files-11 disk or ANSI magtape file structure, device-independent I/O, use RMS record access methods</td>
</tr>
<tr>
<td>File I/O</td>
<td>RMS OPEN and $QIO requests</td>
<td>RMS for OPEN, ACP and Driver</td>
<td>Use Files-11 disk or ANSI magtape file structure, implement own record access methods</td>
</tr>
<tr>
<td>Device I/O</td>
<td>$QIO requests</td>
<td>Driver</td>
<td>Fast dumps to disk or magnetic tape, foreign file structure</td>
</tr>
</tbody>
</table>

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 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 and 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 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. 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 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, VAX/VMS 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 condi-
tions 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 first long-
word of the I/O status block (IOSB).
Input/Output Services

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 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, the 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 built by RMS—sequential, relative, and indexed—are variously available to three different access modes—sequential, keyed, and Record’s File Address. In some cases, RMS 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, DIGITAL’s networking architecture. For details, see Chapter 9 of this Handbook.

After a description of each organization and access mode, below, is an examination in some detail of operations in RMS-built files and of the RMS run time environment.

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 11-2 File Organizations—Advantages and Disadvantages

<table>
<thead>
<tr>
<th>File Organization</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>Uses disk and memory efficiently: minimum disk overhead, 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</td>
<td>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 at user’s risk</td>
</tr>
<tr>
<td>Relative</td>
<td>Allows both sequential and random access for all languages Provides random record deletion and insertion Allows records to be write-shared</td>
<td>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</td>
</tr>
</tbody>
</table>
Table 11-2  File Organizations—Advantages and Disadvantages

<table>
<thead>
<tr>
<th>File Organization</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexed</td>
<td>Allows sequential and random access by key value for all languages</td>
<td>Allows data to be stored on disk only</td>
</tr>
<tr>
<td></td>
<td>Allows random record deletion and insertion</td>
<td>Requires more disk space</td>
</tr>
<tr>
<td></td>
<td>Allows records to be read- and write-shared</td>
<td>Uses more of the central processing unit to process records</td>
</tr>
<tr>
<td></td>
<td>Allows variable-length records to change length on update</td>
<td>Generally requires multiple disk accesses to process a record</td>
</tr>
<tr>
<td></td>
<td>Allows easy file extension</td>
<td></td>
</tr>
</tbody>
</table>

**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 11-2 illustrates sequential file organization.

![Sequential File Organization](image)

**Relative File Organization**

When relative organization is selected, RMS 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.
Input/Output Services

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 11-3 illustrates a relative file organization.

![Relative File Organization](image)

Figure 11-3 Relative File Organization

Since 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. RMS 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 numerically in a manner that is synonymous with 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, 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 11-4 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.

![Indexed File Organization Diagram](image)

**Figure 11-4** 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 re-
trieved 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 immedi-
ately following the last record. Then each sequential write operation
the program issues causes a record to be written following the previ-
ous 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 repre-
sents a different logical ordering of the records in the file.

When reading records in sequential record access mode from an in-
dexed file, a program initially specifies a key (primary key, first alter-
nate 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
Input/Output Services

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

302
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, 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 format and maximum size specifications for the records the file will contain. The specified format establishes how each record appears in the file. The size specification allows RMS to verify that records written into the file do not exceed the length specified when the file was created.

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-control (VFC) records consist of two distinct parts, the fixed 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 control area is identical for all records of the file. The contents of the fixed control area are completely under the control of the program and can be used for any purpose. For example, fixed control areas can be used to store the
identifier (relative record number or RFA) of related records. Indexed file organizations do not support VFC record format.

**Key Definitions for Indexed Files**

To define a key for an indexed file, the user specifies the position and length of character data 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 indexes require storage space and RMS updates indexes 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
- Signed 15-bit integer
- Unsigned 16-bit binary
- Signed 31-bit integer
- Unsigned 32-bit binary
- Packed decimal

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.
Input/Output Services

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 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.

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 run time. The user controls the sharing of a file through information the program provides 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 uses a record locking facility to control operations to a relative or indexed file that is being accessed by one or more writers. The purpose of this facility is to ensure that a program can add, delete, or modify a record in a file without another program's 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.

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, RMS permits a program to perform the following record operations:

● Get a record — RMS returns an existing record within the file to the program.
Input/Output Services

- 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 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
The 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, 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:
Input/Output Services

- 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. 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 physical 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 Run Time 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 record management services procedures are complemented by a number of utilities designed especially for RMS file creation and maintenance. The terminal user has access to the following utilities through the VAX/VMS command interpreter:

- DEFINE creates an RMS file and defines the attributes of the file
- DISPLAY lists the attributes of a single file or a group of files
- CONVERT permits insertion of records into an indexed file, creation of a sequential file from the records of an indexed file, and appending records to a sequential file
- BACKUP creates copies of a single file or group of files in a compact format for later restoration by RESTORE
- RESTORE creates standard RMS files from the compact versions of these files created by BACKUP
- INDEXED FILE LOAD builds an indexed file using records from another RMS file of any organization; this is a very fast, direct, and efficient way to populate an empty RMS indexed file.

**USING VAX-11 RMS**

VAX-11 RMS 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>
<thead>
<tr>
<th>Category</th>
<th>Macro Name</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>FILE PROCESSING</td>
<td>$CREATE</td>
<td>Creates and opens a new file</td>
</tr>
<tr>
<td>FAB = address</td>
<td>$OPEN</td>
<td>Opens an existing file and initiates file processing</td>
</tr>
<tr>
<td></td>
<td>$CLOSE</td>
<td>Terminates file processing and closes the file</td>
</tr>
<tr>
<td>RECORD PROCESSING</td>
<td>$CONNECT</td>
<td>Associates and connects an RAB to the file</td>
</tr>
<tr>
<td>RAB = address</td>
<td>$GET</td>
<td>Retrieves a record from a file</td>
</tr>
<tr>
<td></td>
<td>$PUT</td>
<td>Writes a new record to a file</td>
</tr>
<tr>
<td></td>
<td>$UPDATE</td>
<td>Rewrites an existing record in a file</td>
</tr>
</tbody>
</table>
Input/Output Services

Listed below is a brief program, with comments, that will demonstrate the ease and simplicity of using VAX-11 RMS 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.
Figure 11-5  A Sample RMS Use Program
Notice in the running of the sample program, below, that merely by use of the ASSIGN command the programmer was able to apply SIMPLE 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
$ I demonstrate how easy RMS is to use
$ set nonerror
$ I use TYPE to type out the sample text file used in
$ I the following tests.
$ I
$ TYPE DEMO.TXT
this is a test program
containing ascii text records
this will be printed out by
the test program SIMPLE
$ I
$ I now assign the text file DEMO.TXT
$ I to the logical name INFILE which the
$ I test program SIMPLE will open for input
$ I
$ 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
$ I
$ I copy the text file to another vax node on the network
$ I
$ COPY/LOG DEMO.TXT GALAXY1:DEMO.TXT
COPY=S=COPYED, DBA01[SAETH][DEMO.TXT]3 copied to GALAXY1:DEMO.TXT! (4 records)
COPY=S=NEWFILES, 1 file created
$ I
$ I now use the same test program SIMPLE to access
$ I the file across the network
$ I
$ ASSIGN GALAXY1: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 11-6  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 12
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-11 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 data base 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-11 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 data base. The data base 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, initialize a device or controller by setting hardware registers and initializing fields in the I/O database.

I/O setup
Prepare an I/O request for a device by formatting data, allocating system buffers, locking pages in memory, validating the request, etc.

Start I/O
Set up device registers and the I/O database to start a device; complete I/O request.

Interrupt handling
Respond to hardware interrupts; read and reset device registers; return status.

Error recovery
Set up device registers for retry of an I/O operation; apply Error Correcting Code (ECC) corrections to disk data; return error status.

Error logging
Write the contents of device registers and other data into an error buffer.

Cancel I/O
Set 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.

**Function decision table**
Lists all valid function codes and buffered function codes for the device. Also lists the addresses of function setup routines. 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, and 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 data base 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.

![Driver Organization Diagram](image)

**Figure 12-2  Driver Organization**

**The I/O Data Base**
The operating system and the device drivers refer to an I/O data base 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)
Input/Output Drivers

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 data base 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 data base. In addition, the packet contains fields into which the driver subroutine can write, such as status fields.

The I/O data base 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.

<table>
<thead>
<tr>
<th>PRIORITY</th>
<th>HARDWARE EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex 1F</td>
<td>Machine Check, Kernel Stack Not Valid</td>
</tr>
<tr>
<td>1E 30</td>
<td>Power Fail</td>
</tr>
<tr>
<td>1D 29</td>
<td>Processor, Memory, or Bus Error</td>
</tr>
<tr>
<td>1C 28</td>
<td>Clock</td>
</tr>
<tr>
<td>1B 27</td>
<td>UNIBUS BR7</td>
</tr>
<tr>
<td>1A 26</td>
<td>UNIBUS BR6</td>
</tr>
<tr>
<td>19 25</td>
<td>UNIBUS BR4</td>
</tr>
<tr>
<td>18 24</td>
<td>Device Interrupt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PRIORITY</th>
<th>SOFTWARE EVENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 15</td>
<td>Reserved for DIGITAL</td>
</tr>
<tr>
<td>0E 14</td>
<td>Device Drivers</td>
</tr>
<tr>
<td>0D 13</td>
<td>Timer Process</td>
</tr>
<tr>
<td>0C 12</td>
<td>Queue Asynchronous System Trap (AST)</td>
</tr>
<tr>
<td>0B 11</td>
<td>Reserved for DIGITAL</td>
</tr>
<tr>
<td>0A 10</td>
<td>I/O Post</td>
</tr>
<tr>
<td>09 09</td>
<td>Process Scheduler</td>
</tr>
<tr>
<td>08 08</td>
<td>AST Delivery</td>
</tr>
<tr>
<td>07 07</td>
<td>User Process Level</td>
</tr>
<tr>
<td>06 06</td>
<td>User Process Level</td>
</tr>
</tbody>
</table>

Figure 12-3  Interrupt Priority Level

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 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 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 IPL level.

**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$: .ASCII \\
20$:

$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

RMS 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 run-time system. The run-time 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. Causes a call to the run time system.
3. Run time system checks to see if this is the first I/O request on that file.
4. If it is, the run time system performs an "OPEN." (When the first I/O request is queued, a function of the run time 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:

```fortran
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 RMS, 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 data base descrip-
tions 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 data base and stores it in the 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.

![Flowchart](image)

**Figure 12-4  Function Decision Table**

The 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
Input/Output Drivers

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 data base, 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 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 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 high speed 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 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:

$QIO_S  
CHAN = channel_number,-  
FUNC = #$S_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 12-5  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 line printer driver's function decision table (FDT). Figure 12-6 illustrates the sequence of pointers from the channel index number to the FDT address.

![Diagram](image)

**Figure 12-6  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 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 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 IPL to the device fork level stored in the UCB. Raising priority to fork level synchronizes driver access to the I/O data base.

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 data base to the driver dispatch table (DDT), which contains the start I/O entry point. This process is illustrated in Figure 12-7.
If the device is busy with another I/O transfer, the operating system inserts the 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. Locate 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.

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 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 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 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 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 Unit Control Block)
• Inserting the UCB in the appropriate 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:
Input/Output Drivers

- 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 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 IPL drops to IPL$_IPOST, 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 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.
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 /X85/

; 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

; define crb offsets
; define dbf offsets
; define dpt offsets
; define idb offsets
; define i/o function codes
; define irp offsets
; define line printer characteristics
; define system message types
; define pcb offsets
; define ucb offsets
; define vec offsets

; LOCAL SYMBOLS

; ARGUMENT LIST OFFSET DEFINITIONS

p1=#0
p2=#0
p3=#0
p4=#12
p5=#16
p6=#20

; CHARACTER CODE DEFINITIONS
```
Input/Output Drivers

\*CR=13  \*FF=12  \*VT=11  \*LF=10  \*TAB=9
\*CARRIAGE RETURN  \*FORM FEED  \*VERTICAL TAB  \*LINE FEED  \*TABULATION

\*CRPEND=1  \*CRPEND=0  \*CARRIAGE RETURN PENDING

\*LP11/LS11/LV11 DEVICE REGISTER OFFSET DEFINITIONS

\*DEFINE LP
\*DEFS LP
\*DEFS LP
\*CONTROL STATUS REGISTER
\*DATA BUFFER REGISTER

\*DEFINE LP

\*DEFINE DEVICE DEPENDENT UNIT CONTROL BLOCK OFFSETS

\*DEFINE UCSB

\*UCBSK_LENGTH
\*DEFS UCSB
\*LINE PRINTER UCB MUX
\*CURRENT HORIZONTAL POSITION
\*CURRENT LINE COUNT ON PAGE
\*OFFLINE TIME COUNTER
\*SPARE UNUSED BYTE

\*DEFINE UCSB

\*LOCAL DATA

\*DRIVER PROLOGUE TABLE

\*DPTAB
\*END=LP_END,= END OF DRIVER
\*ADAPTER=MBA,= ADAPTER TYPE
\*UCBSIZE=128,= UCB SIZE
\*NAME=LPDRIVER,= DRIVER NAME
\*DPT_STORE INIT,= CONTROL BLOCK INIT VALUES
\*DPT_STORE UCB,UCSB,FIPL,B,= FORK IPL
\*DPT_STORE UCB,UCBSL,DEVCHAR,L,= DEVICE CHARACTERISTICS
\*  \<DEVSH_REC>=
\*  \<DEVSH_AVL>=
\*  \<DEVSH_CCL>=
\*  \<DEVSH_CDO>=
\*  \<CARRIAGE CONTROL DEVICE>
\*  \<OUTPUT DEVICE>
\*DPT_STORE UCB,UCBS,DREVCLASS,B,=DEVICE CLASS
\*DPT_STORE UCB,UCBS,DEVTYPE,B,=LP11=DEVICE TYPE

\*DPT_STORE UCB,UCBS,DEVBUFSIZE,W,=132 DEFAULT BUFFER SIZE
\*DPT_STORE UCB,UCBSL,DEVPRECIP,E,=LPSH,M_ECHFORM= PRINTER PARAMETERS
\*DPT_STORE UCB,UCBSL,DIPL,B,=DEVICE IPL
\*DPT_STORE UCB,UCBSL,LP_MUX,W,M,=INITIALIZ MUX
\*DPT_STORE REINIT,= CONTROL BLOCK RE-INIT VALUES
\*DPT_STORE CRS,CRS,INTD,A,D,=LSPINT INTERRUPT SERVICE ROUTINE ADDRESS
\*DPT_STORE CRS,CRSL,INTD+VESC,INITIAL,D,=LP11_INIT CONTROLLER INIT
\*DPT_STORE CRS,CRSL+VESC,UNITINIT,D,=LP11_UNIT UNIT INIT
\*DPT_STORE DDB,DOB,D,=LP11 DDB ADDRESS
\*DPT_STORE END

336
FUNCTION DECISION TABLE (FDT) ROUTINE

Set Mode FDT Routine

Set Mode FDT Routine

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.
Write FDT Routine

; PAGE 338
; GBTTL 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 CCS,
; 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:
; CLRL R11
; CLR L R10
;
; FORMAT:
; MOVLE FP,SP
; PUSH 4+R3,R4,R5,R6,R7,AP>
; MOVLE P(AP),R6
; MOVZNL P2(AP),R9
; CMPL #105,WRITEPBLK,R7
; BEQL 109
; MOVLE P4(AP),IPRSSB,CCARCON(R3)
; MOVLE GTS1,SCARRIAGE
; MOVZL B IPRSSB,CCARCON(R3),R8
; MOVZL B IPRSSB,CCARCON+2(R3),R1
; ADDL R6,R0,R1
; MOVAB 32(R1)[R10],R10
; 1051 TSTL R0
; BEQL 208
; MOVQ R6,R0
; JSB G*EXESWRITECMK
; MOVAB 12(R0)[R10],R1
; JSB G*EXESBUFFREQ
; BLC R0,.4S
; JSB G*EXESALLOCBUF

; SET MODE FUNCTION PROCESSING
; GET ADDRESS OF CHARACTERISTICS
; CAN CHARACTERISTICS QUADWORD BE READ?
; SAVE PACKET ADDRESS
; GET ADDRESS OF UCB MUX
; LOCK UCB FOR WRITE ACCESS
; SET MODE FUNCTION
; IF EQL YES
; SET DEVICE CLASS AND TYPE
; SET DEFAULT BUFFER SIZE
; SET DEVICE CHARACTERISTICS
; UNLOCK UCB
; RESTORE PACKET
; SET NORMAL COMPLETION STATUS
; SET ACCESS VIOLATION STATUS

; WRITE FUNCTION PROCESSING
; CLEAR TOTAL NUMBER OF OVERHEAD BYTES
; ASSUME WRITE PASS ALL FUNCTION
; REMOVE ALL TEMPORARIES FROM STACK
; SAVE REGISTERS
; GET STARTING ADDRESS OF USER BUFFER
; GET LENGTH OF USER BUFFER
; WRITE PHYSICAL BLOCK?
; IF EQL YES
; INSERT CARRIAGE CONTROL INFORMATION
; TRANSLATE CARRIAGE CONTROL INFORMATION
; GET NUMBER OF PREFIX CONTROL BYTES
; GET NUMBER OF SUFFIX CONTROL BYTES
; CALCULATE NUMBER OF CARRIAGE CONTROL BYTES
; CALCULATE TOTAL NUMBER OF OVERHEAD BYTES
; ANY BUFFER SPECIFIED?
; IF EQL NO
; RETRIEVE BUFFER PARAMETERS
; CHECK ACCESSIBILITY OF USER BUFFER
; CALCULATE LENGTH OF BUFFER REQUIRED
; CHECK IF PROCESS HAS SUFFICIENT QUOTA
; IF LBC QUOTA CHECK FAILURES
; ALLOCATE BUFFER FOR LINE PRINTER OUTPUT
@IBLC R0,453  ; IF LBC ALLOCATION FAILURE
MVIL R3,(SP)+3  ; RETRIEVE ADDRESS OF I/O PACKET
MVIL R3,IPSH_SWAPTE(R3)  ; SAVE ADDRESS OF BUFFERED I/O PACKET
SUBW R1,PCSWBYTESCNT(R4)  ; ADJUST BUFFERED I/O QUOTA
MOVW R1,IPSH_BOFF(R3)  ; SET NUMBER OF BYTES CHARGED TO QUOTA
CRLR IPSH_MEDIA(R3)  ; CLEAR LINE FEED COUNT IN PACKET
MOVW R9,IPSH_BCNT(R3)  ; INSERT SIZE OF USER BUFFER
MOVAB 12(R2),R2  ; GET ADDRESS OF BUFFER DATA AREA
MOVAB UCSBL_LP_MUTEX(R5),R8  ; GET ADDRESS OF UCSB_MUTEX
JSB *GSCSCHLOCK  ; LOCK UCSB FOR WRITE ACCESS
BS 595  ; IF EQL YES
SUBW #12,R1  ; CALCULATE ACTUAL LENGTH OF DATA AREA
MOVZBL UCSBB_LP_CURSOR(R5),R8  ; GET CURRENT HORIZONTAL CARRIAGE POSITION
MOVZBL UCSBB_DEVSTS(R5),R9  ; GET CURRENT CARRIAGE RETURN PENDING FLAG
MOVZBL UCSBL_LP_LINCNT(R5),R7  ; GET CURRENT LINE ON PAGE
MOVZBL UCSBB_DEVBUFSIZE(R5),R10  ; GET WIDTH OF PRINTER CARRIAGE
MOV #"X20",AP  ; ASSUME PRINTER DOES NOT HAVE LOWER CASE
BAC #LPV_LOWER,UCSSBL_DEVPEND(R5),555  ; IF CLR, NO LOWER CASE
CRLR AP  ; SET FOR PRINTER WITH LOWER CASE
3551 BL 705  ; INSERT PREFIX CARRIAGE CONTROL
3051 DECL R9  ; ANY MORE BYTES TO TRANSFER TO SYSTEM BUFFER?
BLS 405  ; IF LESS NO
MOVZBL (R8)+,R0  ; GET NEXT BYTE FROM USER BUFFER
BSSB WRITE_BYTE  ; WRITE BYTE IN SYSTEM BUFFER
BRR 305  ;
4051 BSBS 805  ; INSERT SUFFIX CARRIAGE CONTROL IN BUFFER
SUBL IPSH_SWAPTE(R3),R2  ; CALCULATE LENGTH OF OUTPUT PLUS HEADER
SUBH #12,R2,IPSH_MEDIA(R3)  ; CALCULATE ACTUAL LENGTH OF OUTPUT BUFFER
MOV R4,UCSSBL_LP_CURSOR(R5)  ; SAVE CURRENT HORIZONTAL CARRIAGE POSITION
INSV R7,UCSSBL_LP_LINCNT(R5)  ; SAVE CURRENT LINE ON PAGE
BRR 605  ; RESTORE REGISTERS
JMP *GEXEBSABORTID  ;
5051 MOVW R9,IPSH_MEDIA(R3)+2(R3)  ; INSERT NUMBER OF BYTES TO PRINT
MOVW R9,(R8),(R2)  ; MOVE CHARACTERS TO SYSTEM BUFFER
JMP *GEXEBSABORTID  ; RESTORE REGISTERS
MOVL #R0,R3  ; SAVE ADDRESS OF I/O PACKET
MOVL UCSBB_LP_MUTEX(R5),R8  ; GET ADDRESS OF UCSB_MUTEX
JSB *GSCSCHUCK  ; UNLOCK UCSB_MUTEX
BSPL R3  ; RESTORE ADDRESS OF I/O PACKET
JMP *GEXEBS10DRVPKT  ; QUEUE I/O PACKET TO DRIVER

7051 MOVZBL IPRB_B_CARCON(R3),=(SP)  ; GET NUMBER OF CHARACTERS TO OUTPUT
BEUL 1008  ; IF EQL NONE
MOVZBL IPRB_B_CARCON+1(R3),R8  ; GET CHARACTER TO OUTPUT
BRR 0599  ;
8051 MOVZBL IPRB_B_CARCON+2(R3),=(SP)  ; GET NUMBER OF CHARACTERS TO OUTPUT
BEUL 1008  ; IF EQL NONE
MOVZBL IPRB_B_CARCON+3(R3),R8  ; GET CHARACTER TO OUTPUT
8591 BNEQ 903  ; IF NEQ CHARACTER SPECIFIED
MOVZBL #C_CR,R0  ; GET CARRIAGE RETURN
BSBSS WRITE_BYTE  ; WRITE BYTE IN SYSTEM BUFFER
MOVZBL #C_LF,R0  ; GET LINE FEED
BSBSS WRITE_BYTE  ; WRITE BYTE IN SYSTEM BUFFER
9081 SOCTR (SP),R9  ; ANY MORE LEFT TO INSERT?
10081 TSTL (SP)+  ; REMOVE COUNT FROM STACK
R59  ; PAGE
JMP *GEXEBS10WRITE  ; WRITE BYTE INTO SYSTEM BUFFER

7051 MOVZBL IPRB_B_CARCON(R3),=(SP)  ; GET NUMBER OF CHARACTERS TO OUTPUT
BEUL 1008  ; IF EQL NONE
MOVZBL IPRB_B_CARCON+1(R3),R8  ; GET CHARACTER TO OUTPUT
BRR 0599  ;
8051 MOVZBL IPRB_B_CARCON+2(R3),=(SP)  ; GET NUMBER OF CHARACTERS TO OUTPUT
BEUL 1008  ; IF EQL NONE
MOVZBL IPRB_B_CARCON+3(R3),R8  ; GET CHARACTER TO OUTPUT
8591 BNEQ 903  ; IF NEQ CHARACTER SPECIFIED
MOVZBL #C_CR,R0  ; GET CARRIAGE RETURN
BSBSS WRITE_BYTE  ; WRITE BYTE IN SYSTEM BUFFER
MOVZBL #C_LF,R0  ; GET LINE FEED
BSBSS WRITE_BYTE  ; WRITE BYTE IN SYSTEM BUFFER
9081 SOCTR (SP),R9  ; ANY MORE LEFT TO INSERT?
10081 TSTL (SP)+  ; REMOVE COUNT FROM STACK
R59  ; PAGE
JMP *GEXEBS10WRITE  ; WRITE BYTE INTO SYSTEM BUFFER

11001 MOVE BSBC #V_CREPEN,R0,605  ; SET, CARRIAGE RETURN PENDING
CMPL #A/",R0  ; POSSIBLY LOWER CASE CHARACTER
BCHKR 605  ; IF BTRU YES
BSSBC #V_CREPEN,R0,605  ; IF SET, CARRIAGE RETURN PENDING
CMPL #A/",R0  ; POSSIBLY LOWER CASE CHARACTER
BCHKR 605  ; IF BTRU NO
CMPP #A/",R0  ; DELETE CHARACTER
BCHKR 305  ; IF EQL YES
SEUL 305  ; CONVERT CHARACTER TO UPPER CASE
SAUL AP,R0  ;
Input/Output Drivers

1001 CMPL R4,R10  ; STILL ROOM ON CURRENT LINE?
BGTRU 308  ; IF GTRU NO
INCL R4  ; INCREMENT HORIZONTAL POSITION
2001 DECL R1  ; ANY ROOM LEFT IN SYSTEM BUFFER?
BLSS 1508  ; IF LSS NO
MOVB R0,(R2)+  ; INSERT CHARACTER IN SYSTEM BUFFER
3001 CALL  ;

; CONTROL CHARACTER ENCOUNTERED

4001 CMPL #C_CR,R0  ; CARRIAGE RETURN?
BLSS 505  ; IF LSS NO
BGTRU 705  ; IF GTRU NO
BBS #LPSV_CR,UCBSL_DEVDPEND(R5),1408  ; IF SET, CARRIAGE RETURN REQUIRED
BICL #M_CRPEND,R0  ; SET CARRIAGE RETURN PENDING
RBB  ;
5001 BBCC #V_CRPEND,R6,205  ; IF CLR, CARRIAGE RETURN NOT PENDING
6001 PUSHL R0  ; SAVE CURRENT CHARACTER
MOV2BL #C_CR,R0  ; GET CARRIAGE RETURN CHARACTER
BBBB 1408  ; INSERT CARRIAGE RETURN IN BUFFER
PDLR R0  ; RETRIEVE CURRENT CHARACTER
BRR WRITE_BYTE  ;

; CHARACTER IS A TAB, LINE FEED, VERTICAL TAB, OR FORM FEED

7001 CMPL #C_TAB,R0  ; TABULATION CHARACTER?
BGTRU 505  ; IF GTRU NO
BLSS 808  ; IF LSS NO

; CHARACTER IS A TAB

BBCC #V_CRPEND,R6,60S  ; IF SET, CARRIAGE RETURN PENDING
PUSHAB R4  ; CALCULATE NEXT TAB POSITION
BICL #7,(SP)  ; CLEAR EXCESS BITS
SUBL R4,(SP)  ; CALCULATE BLANK COUNT
MOV2BL #"/",R0  ; SET SPACE CHARACTER
BRB 1005  ;

; CHARACTER IS A LINE FEED, VERTICAL TAB, OR FORM FEED

8001 CMPL #C_VT,R0  ; VERTICAL TAB?
BEQL 505  ; IF EQL YES
BGTRU 1105  ; IF GTRU LINE FEED

; CHARACTER IS A FORM FEED

MOV2BL UCBSL_DEVDPEND+3(R5),R0  ; GET NUMBER OF LINES PER PAGE
SUBL R4,R4,(SP)  ; CALCULATE NUMBER OF LINES TO END OF PAGE
BCC #LPSV MechForm,UCBSL_DEVDPEND(R5),R0  ; IF CLR, NO MECHANICAL FEED
ADDL (SP),#IRPSL_MEDIA(R3)  ; UPDATE NUMBER OF LINES PRINTED
MOV2BL #C_FF,R0  ; SET FORM FEED CHARACTER
BRR 128S  ;
9001 MOV2BL #C_LF,R0  ; SET LINE FEED CHARACTER
10001 BSBBR WRITE_BYTE  ; INSERT BYTE IN SYSTEM BUFFER
SOGTR (SP),1005  ; ANY MORE BYTES TO INSERT?
TSLL (SP)+  ; REMOVE LOOP COUNT FROM STACK

; CHARACTER IS A LINE FEED

11001 INCL R7  ; INCREMENT LINE POSITION ON PAGE
INCL IRPSL_MEDIA(R3)  ; INCREMENT NUMBER OF LINES PRINTED

340
Input/Output Drivers

CMPB R7,UCBSLENybrid(R5) ; END OF PAGE?
BNEC 138 ; IF NEG NO
128100 ; CLEAR LINE POSITION ON PAGE
138100 ; CLEAR CARRIAGE RETURN PENDING
148100 ; CLEAR HORIZONTAL POSITION

; OUTPUT WILL NOT FIT IN ALLOCATED BUFFER

1581100 MOVL IRPSL_SVAPTE(R3),R0 ; GET ADDRESS OF BUFFER TO DEALLOCATE
CLRL IRPSL_SVAPTE(R3) ; INDICATE NO BUFFER ALLOCATED
MOVHL IRPSL8_SIZE(R5),R10 ; SAVE SIZE OF BUFFER
JSB G*EXECDEADNAPGED ; DEALLOCATE BUFFER
MOVAB =m61(FP),SP ; REMOVE ALL TEMPORARIES FROM STACK
POPR #4< R3,R4,R5,R6,R7,AP ; RESTORE REGISTERS
ADDD R16,PCBSW_BYTECNT(R4) ; ADJUST COUNT BYTE QUOTA
ADDO #32,R11 ; ADJUST COUNT OF OVERHEAD BYTES
SAVE ADDRESS OF I/O PACKET
SAVE ADDRESS OF I/O BUFFER
POPL R3 ; UNLOCK UCB
POPQ 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 INTERRUPT STATE
; OR THE I/O IS COMPLETE.
;
; STARTIO:
MOVL UCBSLENIRR(R5),R3 ; RETRIEVE ADDRESS OF I/O PACKET
MOVM IRPSL_MEDIA+2(R3),UCBSW_BOFF(R5) ; SET NUMBER OF CHARACTERS TO PRINT
MOVL UCBSLENRIORITY(R5),R3 ; GET ADDRESS OF SYSTEM BUFFER
MOVAB 12(R3),R3 ; GET ADDRESS OF DATA AREA
MOVL UCBSLENCRB(R5),R4 ; GET ADDRESS OF CRB
MOVL *IRSBSL_INTD+VCLS_IDB(R4),R4 ; GET DEVICE CSR ADDRESS
;
; START NEXT OUTPUT SEQUENCE
;
1081 ADDL3 #LP_OBR,R4,R0 ; CALCULATE ADDRESS OF DATA BUFFER REGISTER
MOVHL UCBSW_BOFF(R5),R1 ; GET NUMBER OF CHARACTERS REMAINING
MOVAB #898888,R2 ; GET CONTROL REGISTER TEST MASK
BRB 258 ;
2081 BTW R2,(R4) ; PRINTER READY OR HAVE PAPER PROBLEM?
BLEQ 305 ; IF LEG NOT READY OR PAPER PROBLEM
;
; MOVAB (R3),(R0) ; OUTPUT NEXT CHARACTER
2581 SOGED R1,205 ; ANY MORE CHARACTERS TO OUTPUT?
BRB 708 ;
;
; PRINTER IS NOT READY OR HAS PAPER PROBLEM
;
3081 BNEC 408 ; IF NEG PAPER PROBLEM
ADDW3 #1,R1,UCBSW_BOFF(R5) ; SAVE NUMBER OF CHARACTERS REMAINING
DBBINT ; DISABLE INTERRUPTS

341
Interrupt Service Routine Code

```
PAGE

.68TL LP11/L81/LV11 LINE PRINTER INTERRUPT DISPATCHER

; LPSINT = LP11/L811/LV11 LINE PRINTER INTERRUPT DISPATCHER.

; THIS ROUTINE IS ENTERED VIA A JOB INSTRUCTION WHEN AN INTERRUPT OCCURS ON AN
; LP11/L81/LV11 LINE PRINTER CONTROLLER. THE STATE OF THE STACK ON ENTRY IS:
; 08(SP) = ADDRESS OF IDB ADDRESS,
; 06(SP) = SAVED R0,
; 08(SP) = SAVED R1.
; 12(SP) = SAVED R2,
; 16(SP) = INTERRUPT PC,
; 20(SP) = INTERRUPT P8L.

; 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.

LPSINT: MOVL #(SP)+,R3 ;ENTRY FROM DISPATCH
MOVQ IDBSL,C8R(R3),R4 ;GET CONTROLLER C8R AND OWNER UCB ADDRESS
BBC #UCBSV_INT,UCBSW_STS(R5),108 ;IF CLR, INTERRUPT NOT EXPECTED
CLRW (R4) ;DISABLE OUTPUT INTERRUPTS
```
Input/Output Drivers

MOVL UCB3L_F3(R5), R3  ; RESTORE REMAINDER OF DRIVER CONTEXT
JSB @UCB3L_FPC(R5)  ; CALL DRIVER AT INTERRUPT WAIT ADDRESS
106: MOVQ (SP)+, R6  ; RESTORE REGISTERS
MOVQ (SP)+, R2
REI

; PAGE 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 #UCB3M_ONLINE, UCB3W_STS(R5)  ; SET UNIT ONLINE
MOVL UCB3L_CRB(R5), R0  ; GET ADDRESS OF CRB
MOVL CRB3L_INT+VEC3L_IDB(R0), R0  ; GET ADDRESS OF IDB
MOVL RS, IDB3L_OWNER(R0)  ; SET ADDRESS OF DEVICE UCB
LP_LX11_INIT:
LP_END:

; END

; ADDRESS OF LAST LOCATION IN DRIVER

343
CHAPTER OVERVIEW
Sophisticated techniques of memory management and concepts and details of virtual memory design are explained. 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 13
VIRTUAL MEMORY AND MEMORY MANAGEMENT

INTRODUCTION
The function of an operating system is to manage the system's available resources. VAX/VMS is a multi-user, multiprogram operating system. To accommodate multiprogramming, main memory must be shared by more than one process. Therefore, main 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 main 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 data base 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 function of memory management is to map virtual pages into physical address space, to control the paging of the working set of 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 main 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-11 virtual memory scheme are:

1. Only a portion of the program (those pages which are being actively referenced) need reside in main memory during execution.
2. Programs (processes) are allowed to exceed the maximum amount of main memory available.

Virtual Address Space
Because of the VAX-11 family 32-bit architecture, the address of a byte datum is a longword. Therefore, VAX-11 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
Virtual Memory and Memory Management

Process address space is further divided into a program and control region. Virtual address space is described in Figure 13-1.

```
| BIT 30 = 0 |
| BIT 31 = 0 |
| BIT 30 = 1 |
| BIT 31 = 0 |
| BIT 31 = 1 |

0  | GROWTH DIRECTION  |
P0 (PROGRAM) REGION
P1 (PROGRAM) REGION
SYSTEM REGION (COMMON)
RESERVED REGION

| PROCESS SPACE |
| SYSTEM SPACE |

Figure 13-1  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.* 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 tables containing references to each individual page in the program region. Therefore, virtual addresses in the program region are translated to physical address using registers P0BR and P0LR. The program region corresponds to P0 space.

The control region of a process's address space contains process-related information maintained by the system and process control

* Some technical terms are defined below or in other chapters. See the Glossary and the Index.
Virtual Memory and Memory Management

structures such as the kernel, executive, supervisor, and user stacks, and the process I/O data base. The control region originates at location \(2^{31}\) and extends toward lower addressed locations. P1BR and P1LR are hardware registers that describe the page tables containing references to each individual page in the control region. Therefore, the base address and length of the control region are mapped using 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.

![Figure 13-2 System Virtual Address Space](image)

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.
Virtual Memory and Memory Management

![Virtual Address Diagram](image)

**Figure 13-3** Virtual Address

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 main memory.

![Physical Address Diagram](image)

**Figure 13-4** 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 an even 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 main memory, but they remain virtually contiguous in the process's address space.

The VAX/VMS operating system maintains a unique translation map called the page table 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-11 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-11 processor, its hardware context is stored in a software data structure called the hardware process control block (PCB). VAX/VMS maintains the hardware PCB in a software structure known as the process header (PHD). Saving the contents of the VAX-11 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 VAX/VMS.

The VAX/VMS operating system maintains 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 in Paging, below.

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 working sets residing in main memory at any one moment is called the balance set. The process’s working set memory requirements balance the total available main memory of the system. 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. (See Chapter 14 for greater detail.) The swapper utilizes three conditions to determine which processes should be swapped in and which should be swapped out:

350
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.

**PROCESS CONTROL STRUCTURES**

VAX-11 hardware defines a process by using registers and a hardware 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 and quota information
5. Working set list
6. Process section table
7. P0 and P1 page tables
The working set list contains entries to describe that portion of the process's virtual address space that is resident in physical memory. This data base 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 cross linkages among modules and the assignment of virtual address space.

An image is the result of linking 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. For example, if the user wishes to assemble and link a program, the process executes two images: the assembler and then the linker. Images are referred to by file name.

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 user attributes, such as read-only access, read/write access, absolute, relocatable, etc.

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 image name of the AME is contained in the shell process used in process creation.

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 file 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. It creates one page table entry for each page of the image. Each page table entry corresponds to one virtual page of 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 VAX/VMS 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 depending on a cluster factor), 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 resident 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 data bases:

- Page Frame Number (PFN) data base
- Free page list
- Modified page list
- Working set data base
- Page table entries

The paging philosophy implemented in the VAX/VMS 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. Though this may reduce the speed with which an individual process executes, and may increase that process’s overhead, it protects all the other processes executing in the system. Another possible philosophy, paging against the system, was not implemented in VAX/VMS because it allows one process to push out pages belonging to another process.

**Page Frame Number (PFN) Data Base**

The PFN data base 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 data base 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 data base 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 data base 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 data base 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 data base 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 overall philosophy being used by the pager and the swapper is
that all processes in the balance set have all of their working set pages
in physical memory. A process only pages when it needs more memo-
ry than it is allowed. In that case, it pages against itself; it cannot cause
pages of another process to be released.

The pager enforces the rule that a process pages against itself. When
a page fault occurs and the process currently occupies all of the physi-
cal memory that it is allowed, the pager chooses the next releasable
working set list entry, releases the page from the working set, and
replaces it with the desired page. Assuming that the reference count
for the released page went to zero, the page is placed on the free page
list or the modified page list, as appropriate. A new page is allocated to
the process from the front of the free page list and the desired virtual
page is placed in it.

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 but not 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.
Virtual Memory and Memory Management

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.

PROCESS SPACE SHAREABLE MEMORY
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 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 known image, it is available for mapping into the virtual address space of many processes. The installation procedures result in the creation of the data base required for the sharing of global sections.

Global Section Data Base
Sharing sections in the process address space requires the creation of a global section data base. The global section data base is created as a result of a Create and Map Section system service issued when a shareable image is installed as a known image or issued by a process to create a global data section. The data base 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 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.

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 preparing for executing an executable image that is bound to a linkable 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 linking with a shareable image containing the desired sections. The image must be made into a known image before the process can map it.
2. By issuing a request for the Map Global Section system service.

361
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 of 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 an indirect pointer to the global page table entry and its associated data base 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

**SWAPPING**

The working set 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 themselves.
CHAPTER OVERVIEW

Central to the effectiveness of multi-user computers is the algorithm that controls swapping of working sets and the allocation of CPU time. VAX/VMS 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 real-time 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 14
PROCESS SCHEDULING AND SWAPPING

INTRODUCTION
The VAX/VMS scheduler performs normal and real-time 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 real-time process is sometimes referred to as time-critical.

VAX/VMS 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 real-time processes. Real-time 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 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, in local event flag wait, hibernating, etc.

Regardless of which state queue a process is in, the process comprises a collection of pages that is referred to as its working set. The working set 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
Real-time processes take precedence over background processes in the queue for execution, since 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-real-time 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 status 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
Process Scheduling and Swapping

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.

* The **VAX-11 Architecture Handbook** gives detailed descriptions and examples of many VAX/VMS instructions.

![Diagram of State Queue Header](image)

**Figure 14-1** 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.

![Diagram of Wait Queue Header](image)

**Figure 14-2** Wait Queue Header
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.

![Executable Process State Queue Diagram](image)

Figure 14-3 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.

![Process State Transition Diagram](image)

**Figure 14-4  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 (CLOPG), 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.

Real-time 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 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 operation:
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 unsatisfied wait request saves the time at the start of the wait interval. When the wait is satisfied, the amount of time spent waiting 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 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 real-time 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 real-time processes and using a modified priority for other processes allows the scheduler to achieve maximum overlap of compute and I/O activities while still remaining responsive to high-priority real-time requests. Figure 14-5 illustrates process priority scheduling.
Figure 14-5  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 source 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 never is decreased to a value below its base priority or increased above a priority of 15. A real-time 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 their base priorities where they can be interrupted by more event-driven (I/O-bound) processes.

**WORKING SET SWAPPING**

Working set 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 real-time processes and higher than all normal processes. Process creation is discussed in Chapter 5, 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 backup storage. The modified page is written to backup storage only when that page is required as a free page. A page on the modified page list that is being written to backup storage 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.

Modified Page Writing
Modified pages are placed on the modified page list to be processed by the swapper and written to their backup storage address. After the backup 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

**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 decision-making required to cause an inswap to occur is divided into two phases:

1. Insawp 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 soon 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 initiated 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 re-
Process Scheduling and Swapping

quires 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

It is frequently important that processes be able to communicate with one another. This chapter expands on the interprocess communications section of Chapter 2, and goes on to detail the use of common event flags and mailboxes as structures by which processes pass status information and data to one another. The methods by which shared memory areas are established are explained.

Topics are:

- Common Event Flags
- Mailboxes
- Shared Areas of Memory
CHAPTER 15
INTERPROCESS COMMUNICATION

INTRODUCTION
VAX/VMS provides interprocess communication facilities for synchronizing execution, for sending messages, and for sharing common data. The four communication techniques utilized by cooperating processes are:

- Common event flags
- Mailboxes
- Shared areas of memory
- Shared files

Common event flags are associated with group identification. Mailboxes, shared areas of memory, and shared files are more general purpose facilities which can be limited or unlimited in access. They can be limited to processes with the same UID or to a specific group of jobs, or they can be extended to all jobs in the system.

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. 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
Interprocess Communication

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 following functions:

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 10.

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, RMS, 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 AST will be delivered, eliminating the need for an outstanding read to each terminal or network link.

**SHARED AREAS OF MEMORY**

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 are known shareable images created using the linker. 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 literal data. Normally, only one copy of a global section actually resides in memory while cooperating processes reference it. For example, should a global section contain initialization data that processes using the data are expected to change, part of 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 includes a demand allocate zero-filled page. When a process references the global section for the first time, a zero-filled page is mapped into its virtual address space. Such a page is created dynamically and eliminates the necessity of filling up space in secondary storage with pages of zeros. (Typical use of the demand allocate zero-filled page is in buffer or image 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 UIC identifying the group and member family 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.

**SHARED FILES**

A fourth method of interprocess communication is via shared files. Compared to the three methods listed above, the use of shared files is more indirect and carries more restrictions. Some files, for example, cannot be shared. More details can be found in the Record Management Services section of Chapter 11, Input/Output Services.
CHAPTER OVERVIEW
Dealing with exception, exit, and asynchronous conditions and events requires sophisticated software mechanisms such as those incorporated into VAX/VMS. The most 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 16
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. 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 conditions.

Asynchronous System Traps (ASTs), on the other hand, allow a process to respond asynchronously to such expected conditions as I/O completion. Condition handlers are used to manage hardware-generated exceptions and software-generated signals, while exit handlers are used to clean up local data bases during the termination of an image's execution.

Hardware generated exception conditions are the most common; they always 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 compatibility mode traps, 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.

VAX/VMS provides two methods for specifying condition handlers:
- By specifying the address of a condition handler in the first long-word of the procedure call frame
- By 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. 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. The dispatcher examines only the stack and vectors for the access mode in which the exception occurred to locate a condition handler.

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 condi-
tion 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, FP points to the current call frame. Since the condition handler address is the first longword in a call frame, 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, detecting a previous frame pointer that is zero, or detecting an access violation. 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 crashed:

1. No condition handler could be found.
2. All condition handlers that were found re-signaled the condition.
3. An access violation was detected while searching the stack.

Not finding a condition handler for kernal 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
Figure 16-1a  Stack Search of Multiple Conditions

Figure 16-1b  Conceptual Flow Diagram of Stack
Special Event Handling

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.

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 re-sigaled.
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-own data bases.

Exit handlers are specified using the Declare Exit Handler system service. This service accepts as an argument the address of an exit control block (ECB). The exit control block minimally contains: a longword used to link ECBs 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 data base. Figure 16-2 illustrates the format of an ECB.

VAX/VMS maintains a separate list of ECBs for each access mode. Each list is in last-in/first-out order. As each exit handler is specified, its ECB 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 terminated after running down I/O and performing other cleanup operations. Otherwise,
the appropriate lists of ECBs are examined to determine if any exit handlers were specified.

The exit handler dispatcher scans the list of ECBs one entry at a time. For each ECB, the respective exit handler is called. The argument list specified in the call to the exit handler is that specified in the ECB itself; the reason for the exit is filled into the longword whose address is specified by the first argument. If the entire list of ECBs 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.
Special Event Handling

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.

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.

![Figure 16-3 AST Receiver Stack Content](image)

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 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 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 resident, 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, decreases the I/O count for the channel, 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 AST is initiated by the AST delivery interrupt at IPL 2. The interrupt is triggered as a result of an REI instruction and is processed entirely on the kernel stack. When the interrupt occurs, the system first checks for the deliverability of the
Special Event Handling

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 posting 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 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 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 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. 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

One very important consideration in the design of the VAX-11 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. In addition, it allows the VAX to be used as a development environment for certain non-privileged PDP-11 tasks, namely those that will run under an RSX-11M system. In this chapter, compatibility is discussed.

Topics include:
• The Application Migration Executive (AME)
• Compatibility Mode
• Transportable Languages
• Compatibility Considerations
CHAPTER 17
PDP-11 COMPATIBILITY

INTRODUCTION
A major feature of the VMS 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 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
- RMS file access methods
- DIGITAL Command Language (DCL) and RSX-11 MCR command language

The VAX-11 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-11 high level languages are closely compatible with those of the PDP-11 family.

VMS may effectively serve as a high-performance RSX-11M/S program development system. RSX-11M/S programs can be edited, compiled, and linked on the VMS system. In addition, the task can be partially debugged on VMS. That is, the software development can largely be accomplished on VAX and need only migrate to the target RSX-11M/S system for final debugging and execution.

VMS supports system generation of RSX-11M and RSX-11S systems. In addition, through DECrnet, it supports down-line loading of RSX-11S systems and RSX-11M tasks. However, the 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.

399
Under the VAX/VMS operating system, programs may execute in either of two modes, native or compatibility. Native mode programs use the VAX-11 instruction set and execute under VAX/VMS. Compatibility mode programs, however, are those which can execute on other PDP-11s.

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 VMS system with little or no modification to the task image. The Applications Migration Executive is part of the VAX/VMS executive. It supports a mapped RSX-11M environment without supporting the directives to manipulate Program Logical Address Space (PLAS) or DECnet calls. 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 VMS system service of equivalent function (except for the memory management directives and DECnet I/O calls, which are not supported). If there is no equivalent 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, COBOL and BASIC-PLUS-2 compilers as well as many existing PDP-11 utilities.

When programming in compatibility mode, certain points should be established:

- Users’ images are limited to 64 Kb 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
- 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 assembler
- PDP-11 COBOL compiler
- PDP-11 BASIC-PLUS-2 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's command language interpreter can also accept the RSX-11M Monitor Console Routine (MCR) commands, either typed directly on a terminal, or submitted as command files. For the most part, the commands are interpreted as functionally equivalent DCL commands.

Any task (PDP-11 COBOL, BASIC-PLUS-2, FORTRAN) linked for RSX-11M will run, assuming the task is non-privileged. FORTRAN IV-PLUS PDP-11 programs can be recompiled and run in native mode. If the FORTRAN IV-PLUS PDP-11 program contains system calls, each system call must be converted to its native mode equivalent.

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 floating point or 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 SPAWN or CONNECT executive directives
- It must not rely on environmental features of RSX-11M that VAX/VMS does not support, such as significant events or a task's STOP bit
- It must not use DECnet

401
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, VAX/VMS 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 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**
VAX/VMS 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 VAX/VMS. VAX/VMS loads overlays at the appropriate point in image execution from the image file.

VAX/VMS 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 VAX/VMS by the system manager.

VAX/VMS 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 VAX/VMS receives an error status return.

**Command Languages**
The operating system's command language interpreter accepts both DCL commands and a subset of the RSX-11M MCR 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.
RSX-11M Directive Requests

In RSX-11M, a task image interfaces with the operating system by issuing directive requests. As a result of a directive request, RSX-11M performs the desired function and returns control to the task. VAX/VMS duplicates the task/system interaction achieved by a directive request in RSX-11M. When an RSX-11M task issues a directive, the hardware traps to VAX/VMS. With the exception of the RSX-11M memory management directives, VAX/VMS duplicates the requested RSX-11M function with either of the following results:

- The RSX-11M directive function is duplicated in VAX/VMS and the task continues execution.
- VAX/VMS cannot duplicate the requested function but does take the necessary action to allow continued task execution.

VAX/VMS 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 VAX/VMS 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, VAX/VMS can not 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**
VAX/VMS will support the following RSX-11M directives:

- **ABRT$** Abort Task
- **ALUN$** Assign Task
- **ALPT$** Alter Priority
- **ASTX$** AST Service Exit
- **CLEF$** Clear Event Flag
- **CMKT$** Cancel Mark Time Requests
- **DECL$** Declare Significant Event
- **DSAR$** Disable AST Recognition
- **DSCP$** Disable Checkpointing
- **ENAR$** Enable AST Recognition
- **ENCPS** Enable Checkpointing
- **EXIT$** Exit If
- **EXITSS** Task Exit
- **EXST$** 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$** Receive Data or Exit
- **RDAFS** Read All Event Flags
- **RQST$** Request
PDP-11 Compatibility

RSUM$  Resume
RUN$    Run
SDAT$   Send Data
SETF$   Set Event Flag
SPND$   Suspend
SPRA$   Specify Power Recovery AST
SRDA$   Specify Receive Data AST
SVDB$   Specify SST Vector Table for Debugging Aid
SVTK$   Specify SST Vector Table for Task
WSIG$$  Wait for Significant Event
WTLO$   Wait for Logical or of Event Flags
WTSE$   Wait for Single Event Flag

VAX/VMS does not support a number of RSX-11M directives, principally because of different techniques of memory management in PDP-11 and VAX-11 hardware.

VAX/VMS returns an error status of IE.SDP (invalid directive) to any RSX-11M image that issues an unsupported directive.

VAX/VMS does not support the Specify Floating Point Processor Exception AST directive because the VAX-11 compatibility mode instruction set does not include floating point instructions.

The VAX-11 processor does not have sense switches. Therefore, VAX/VMS handles the Get Sense Switch directive in the same manner as RSX-11M 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 VAX/VMS controls memory management very differently from the way that RSX-11M does. Others are directives for STOP-bit synchronization, SPAWN-CONNECT, Group Global Event Flag numbers, and so forth.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACP</td>
<td>Ancillary Control Process</td>
</tr>
<tr>
<td>ANS</td>
<td>American National Standard</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>AST</td>
<td>Asynchronous System Trap</td>
</tr>
<tr>
<td>ASTLVL</td>
<td>Asynchronous System Trap Level</td>
</tr>
<tr>
<td>CCB</td>
<td>Channel Control Block</td>
</tr>
<tr>
<td>CEB</td>
<td>Common Event Block</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Language Interpreter</td>
</tr>
<tr>
<td>CM</td>
<td>Compatibility Mode bit in the hardware PSL</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CRB</td>
<td>Channel Request Block</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CSR</td>
<td>Central Status Register</td>
</tr>
<tr>
<td>DAP</td>
<td>Data Access Protocol</td>
</tr>
<tr>
<td>DCL</td>
<td>DIGITAL Command Language</td>
</tr>
<tr>
<td>DDB</td>
<td>Device Data Block</td>
</tr>
<tr>
<td>DDCMP</td>
<td>DIGITAL Data Communication Message Protocol</td>
</tr>
<tr>
<td>DDT</td>
<td>Driver Data Table</td>
</tr>
<tr>
<td>DST</td>
<td>Debug Symbol Table</td>
</tr>
<tr>
<td>DV</td>
<td>Decimal Overflow trap enable bit in the PSW</td>
</tr>
<tr>
<td>ECB</td>
<td>Exit Control Block</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correction Code</td>
</tr>
<tr>
<td>ESP</td>
<td>Executive Mode Stack Pointer</td>
</tr>
<tr>
<td>ESR</td>
<td>Exception Service Routine</td>
</tr>
<tr>
<td>FAB</td>
<td>File Access Block</td>
</tr>
<tr>
<td>FCA</td>
<td>Fixed Control Area</td>
</tr>
<tr>
<td>FCB</td>
<td>File Control Block</td>
</tr>
<tr>
<td>FCS</td>
<td>File Control Services</td>
</tr>
<tr>
<td>FDT</td>
<td>Function Decision Table</td>
</tr>
<tr>
<td>FMS</td>
<td>Forms Management System</td>
</tr>
<tr>
<td>FNM</td>
<td>File Name</td>
</tr>
<tr>
<td>FP</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>FPD</td>
<td>First Part (of an instruction) Done</td>
</tr>
<tr>
<td>FU</td>
<td>Floating Underflow trap enable bit in the PSW</td>
</tr>
<tr>
<td>GSD</td>
<td>Global Section Descriptor</td>
</tr>
<tr>
<td>GST</td>
<td>Global Symbol Table</td>
</tr>
<tr>
<td>IDB</td>
<td>Interrupt Dispatch Block</td>
</tr>
<tr>
<td>IOSB</td>
<td>I/O Status Block</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>IPL</td>
<td>Interrupt Priority Level</td>
</tr>
<tr>
<td>IRP</td>
<td>I/O Request Packet</td>
</tr>
<tr>
<td>IS</td>
<td>Interrupt Stack bit in PSL</td>
</tr>
<tr>
<td>ISECT</td>
<td>Image Section</td>
</tr>
<tr>
<td>ISD</td>
<td>Image Section Descriptor</td>
</tr>
<tr>
<td>ISP</td>
<td>Interrupt Stack Pointer</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>IV</td>
<td>Integer Overflow trap enable bit in the PSW</td>
</tr>
<tr>
<td>JSB</td>
<td>Jump to Subroutine</td>
</tr>
<tr>
<td>KED</td>
<td>Keypad Editor</td>
</tr>
<tr>
<td>KSP</td>
<td>Kernel Mode Stack Pointer</td>
</tr>
<tr>
<td>MBA</td>
<td>MASSBUS Adapter</td>
</tr>
<tr>
<td>MBZ</td>
<td>Must be Zero</td>
</tr>
<tr>
<td>MCR</td>
<td>Monitor Console Routine</td>
</tr>
<tr>
<td>MFD</td>
<td>Master File Directory</td>
</tr>
<tr>
<td>MFPR</td>
<td>Move From Process Register instruction</td>
</tr>
<tr>
<td>MME</td>
<td>Memory Mapping Enable</td>
</tr>
<tr>
<td>MTPR</td>
<td>Move To Process Register instruction</td>
</tr>
<tr>
<td>MUTEX</td>
<td>Mutual Exclusion semaphore</td>
</tr>
<tr>
<td>NETACP</td>
<td>Network Ancillary Control Process</td>
</tr>
<tr>
<td>NCB</td>
<td>Network Connect Block</td>
</tr>
<tr>
<td>NSP</td>
<td>Network Services Protocol</td>
</tr>
<tr>
<td>OPCOM</td>
<td>Operator Communication Manager</td>
</tr>
<tr>
<td>P0BR</td>
<td>Program region base register</td>
</tr>
<tr>
<td>P0LR</td>
<td>Program region length register</td>
</tr>
<tr>
<td>P1BR</td>
<td>Control region base register</td>
</tr>
<tr>
<td>P1LR</td>
<td>Control region limit register</td>
</tr>
<tr>
<td>P1PT</td>
<td>Control region page table</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PCB</td>
<td>Process Control Block</td>
</tr>
<tr>
<td>PCBB</td>
<td>Process Control Block Base register</td>
</tr>
<tr>
<td>PFN</td>
<td>Page Frame Number</td>
</tr>
<tr>
<td>PID</td>
<td>Process Identification Number</td>
</tr>
<tr>
<td>PLAS</td>
<td>Program Logical Address Space</td>
</tr>
<tr>
<td>PME</td>
<td>Performance Monitor Enable bit in PCB</td>
</tr>
<tr>
<td>PSECT</td>
<td>Program Section</td>
</tr>
<tr>
<td>PSL</td>
<td>Processor Status Longword</td>
</tr>
<tr>
<td>PSW</td>
<td>Processor Status Word</td>
</tr>
<tr>
<td>PTE</td>
<td>Page Table Entry</td>
</tr>
<tr>
<td>QIO</td>
<td>Queue Input/Output Request system service</td>
</tr>
<tr>
<td>RAB</td>
<td>Record Access Block</td>
</tr>
<tr>
<td>REI</td>
<td>Return from Exception or Interrupt</td>
</tr>
<tr>
<td>RFA</td>
<td>Record's File Address</td>
</tr>
<tr>
<td>RMS</td>
<td>Record Management Services</td>
</tr>
</tbody>
</table>
## Appendix A

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWED</td>
<td>Read, Write, Execute, Delete</td>
</tr>
<tr>
<td>SBI</td>
<td>Synchronous Backplane Interconnect</td>
</tr>
<tr>
<td>SBR</td>
<td>System Base Register</td>
</tr>
<tr>
<td>SCB</td>
<td>System Control Block</td>
</tr>
<tr>
<td>SCBB</td>
<td>System Control Block Base register</td>
</tr>
<tr>
<td>SLR</td>
<td>System Length Register</td>
</tr>
<tr>
<td>SP</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>SPT</td>
<td>System Page Table</td>
</tr>
<tr>
<td>SSP</td>
<td>Supervisor Mode Stack Pointer</td>
</tr>
<tr>
<td>SVA</td>
<td>System Virtual Address</td>
</tr>
<tr>
<td>TP</td>
<td>Trace trap Pending bit in PSL</td>
</tr>
<tr>
<td>UBA</td>
<td>UNIBUS Adapter</td>
</tr>
<tr>
<td>UCB</td>
<td>Unit Control Block</td>
</tr>
<tr>
<td>UETP</td>
<td>User Environment Test Package</td>
</tr>
<tr>
<td>UFD</td>
<td>User File Directory</td>
</tr>
<tr>
<td>UIC</td>
<td>User Identification Code</td>
</tr>
<tr>
<td>USP</td>
<td>User Mode Stack Pointer</td>
</tr>
<tr>
<td>VAX</td>
<td>Virtual Address Extender</td>
</tr>
<tr>
<td>VBF</td>
<td>Variable-Length Bit Field</td>
</tr>
<tr>
<td>VCB</td>
<td>Volume Control Block</td>
</tr>
<tr>
<td>VMS</td>
<td>Virtual Memory Operating System</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Page Number</td>
</tr>
<tr>
<td>WCB</td>
<td>Window Control Block</td>
</tr>
<tr>
<td>WCS</td>
<td>Writable Control Store</td>
</tr>
<tr>
<td>WDCS</td>
<td>Writable Diagnostic Control Store</td>
</tr>
</tbody>
</table>
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 op-
erand 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 oper-
and is calculated using the general registers. The basic general
register addressing modes are called: register, register deferred, au-
toincrement, autoincrement deferred, autodecrement, displacement,
and displacement deferred. In addition, there are six indexed ad-
dressing modes using two general registers, and literal mode ad-
dressing. The PC addressing modes are called: immediate (for regis-
ter 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 Intercon-
nect (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 direc-
tory 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 con-
tains 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 op-
erations correspond directly to machine opcodes into an object lan-
guage.
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 (217₁₀) 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.
Glossary

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.
**Glossary**

**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 specification, 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 pro-
cedures (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
Glossary

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
Glossary

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 ($\pm$) $0.29 \times 10^{-38}$ to $1.7 \times 10^{38}$. 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.

dynaco 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.
Glossary

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.
Glossary

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 ($\pm$) $0.29 \times 10^{-38}$ to $1.7 \times 10^{38}$. 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.
Glossary

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), auto decrement 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.
Glossary

I/O Data Base  A group of data control blocks that are an important part of the communications link between the operating system 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-orient-
ed 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.

**privileged**  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.
glossary

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 low-
est-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 ad-
dresses.

**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 program-
mer's viewpoint, the secondary storage locations appear to be loca-
tions 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 memo-
ry.

**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

abbreviation rule 59
absolute time 279
ACCEPT statement 126-127
access
  mode for 300-303
  system 43,46
  types of 31
accounting, for use of system resources 32
adapter control block 321
addresses
  virtual address space and 7, 347-348
  virtual memory allocation of 211
address sort 179
Adjust Outer Mode Stack Pointer ($ADJSTK) 287
Adjust Working Set Limit ($ADJWSL) 286
ALLOCATE command 62
Allocate Device ($ALLOC) 268-270
allocation
  memory, algorithm for 212
  of resources, using VAX-11 Common Run Time Procedure Library 225
  in virtual memory 211
alternate keys 305-306
altmode (escape) key 110
ANALYZE/OBJECT command 63
ancillary control processes (ACP) 19, 292
APPEND Command 63-64, 154
Application Design Tool (ADT) 171
application migration execution (AME) 346, 355, 400
applications
  using VAX-11 Forms Management System 184-186
  on VAX-11 PL/I 158-159
APPLY clause 137
approximate key matches 309
architecture, of VAX-11 systems 11
argument list 389-391
Argument Pointer register (AP) 261
assembler languages 5
VAX-11 MACRO 113-116
ASSIGN command 35, 51, 64, 263
Assign I/O Channel ($ASSIGN) 244, 247, 248, 266, 294, 295
assignment statements 121
Associate Common Event Flag Cluster ($ASCEFC) 258
asterisks (*) 64, 192
AST Level register (ASTLVL) 394, 395
asynchronous system trap (AST) services 16, 260-263, 296, 370
  special event handling by 387, 392-396
attributes, of files and records 303-311
AUTHORIZE (User Authorization Program) 33
background processes 365, 366
background store (secondary storage) addresses 356-357
BACKUP utility 310
balance sets 12, 350-351, 370-371
BASIC 22
BASIC command 64-65
BASIC-PLUS-2 160-161
BASIC-PLUS-2/VAX 22
batch queues 37
binary codes 5
binary system 8
binding 354
bits 3
blanks, in qualifiers 58
BLISS 113, 141
BLISS command 65
block I/O processing 309-310
BREAK command 163
breakpoints 163
Broadcast ($BRDCST) 272
BUILTIN declarations 143
bytes 3
CAI (Computer-Assisted Instruction) program 196
CALL command 220-221
CALL facility 130-131
call frames 389
CALLG instruction 261
calls
for asynchronous system trap services 263
for change mode services 286-287
for condition handling services 283
for event flag services 258-260
for I/O services 266-273
for logical name services 265
macro 116, 245, 250, 255
for memory management services 283-286
for process control services 273-279
for timer and time conversion services 280-282
in VAX-11 FORTRAN 118
with VAX-11 SORT 181, 182
call stack 389
CALL statement
in PDP-11 BASIC-PLUS-2/VAX 162
RMS invoked by 19
in VAX-11 BASIC 153
in VAX-11 COBOL-74 130
CANCEL command 65, 221-222
Cancel Exit Handler ($CANEXH) 278
Cancel I/O on Channel ($CANCEL) 270-271
Cancel Timer Request ($CANTIM) 281-282
Cancel Wakeup ($CANWAK) 276, 282
carriage return (CR; return) key 110
central processor unit (CPU), scheduling of 2
change mode 196-197, 200-201
change mode services 18, 286-287
Change To Executive ($CMEXEC) 286
Change to Kernel Mode ($CMKRN$) 286-287
channel request block 321
channels 289
assignment of 294
deassigning 249-250
CHARACTER data type 117-118, 122
characters
lowercase 59
in VAX-11 FORTRAN 117-118
CHARACTER variables 117, 122
C key 110
↑C key 45
Clear Event Flag ($CLREF) 259
CLOSE command 65-66
CLOSE statement 117, 242
COBOL 22, 133
COBOL/C74 command 66
COBOL/RSX11 command 66-67
collections, in DATATRIEVE 170-171
command input, for SLP editor 203
command languages 9, 57-110
compatibility of 402
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>commands</td>
</tr>
<tr>
<td>with DATATRIEVE 168-169, 173-176</td>
</tr>
<tr>
<td>for DEBUG facility 217-224</td>
</tr>
<tr>
<td>entering 43-45</td>
</tr>
<tr>
<td>formats for 57-60</td>
</tr>
<tr>
<td>language name 60-61</td>
</tr>
<tr>
<td>procedures 61-62</td>
</tr>
<tr>
<td>for SLP editor 204-206</td>
</tr>
<tr>
<td>for SOS text editor 192-193</td>
</tr>
<tr>
<td>command strings 180-181</td>
</tr>
<tr>
<td>commas (,), in commands 58</td>
</tr>
<tr>
<td>comment characters 59</td>
</tr>
<tr>
<td>common event blocks (CEB) 381-382</td>
</tr>
<tr>
<td>common event flag clusters 381</td>
</tr>
<tr>
<td>common event flags 381-382</td>
</tr>
<tr>
<td>Common Run Time Procedure Library (RTL) 24, 160, 224-227</td>
</tr>
<tr>
<td>COMMON statement 153, 162-163</td>
</tr>
<tr>
<td>communications 229-252</td>
</tr>
<tr>
<td>channels of 289</td>
</tr>
<tr>
<td>interprocess 2, 18, 381-384</td>
</tr>
<tr>
<td>services for 21-22</td>
</tr>
<tr>
<td>communication networks 229</td>
</tr>
<tr>
<td>compatibility mode 22, 113, 155, 400</td>
</tr>
<tr>
<td>compat mode 39</td>
</tr>
<tr>
<td>compiled-code support procedures 227</td>
</tr>
<tr>
<td>compilers 5</td>
</tr>
<tr>
<td>VAX-11 BLISS-32 145-146</td>
</tr>
<tr>
<td>VAX-11 FORTRAN 119-120</td>
</tr>
<tr>
<td>COMPUTED BY fields 173</td>
</tr>
<tr>
<td>Computer-Assisted Instruction (CAI) program 196</td>
</tr>
<tr>
<td>concatenation characters 59</td>
</tr>
<tr>
<td>conditional assembly directives 115</td>
</tr>
<tr>
<td>condition handlers 387-391</td>
</tr>
<tr>
<td>condition handing 225, 282</td>
</tr>
<tr>
<td>of special events 387-389</td>
</tr>
<tr>
<td>condition handling services 17, 282-283</td>
</tr>
<tr>
<td>CONTAINING relational operator 172</td>
</tr>
<tr>
<td>contents, tables of 210</td>
</tr>
<tr>
<td>context 12, 350</td>
</tr>
<tr>
<td>context switching 350</td>
</tr>
<tr>
<td>continuation characters 58-59</td>
</tr>
<tr>
<td>CONTINUE command 67, 163</td>
</tr>
<tr>
<td>Contract Program/Control Region ($CNTREG) 284</td>
</tr>
<tr>
<td>control blocks 321</td>
</tr>
<tr>
<td>control statements 121</td>
</tr>
<tr>
<td>Control/Status Register (CSR) 332</td>
</tr>
<tr>
<td>Convert ASCII String to Binary Time ($BINTIM) 280-281</td>
</tr>
<tr>
<td>Convert Binary Time to ASCII String ($ASCTIM) 280</td>
</tr>
<tr>
<td>Convert Binary Time to Numeric Time ($NUMTIM) 280</td>
</tr>
<tr>
<td>CONVERT utility 310</td>
</tr>
<tr>
<td>COPY command 67-68</td>
</tr>
<tr>
<td>CORAL 113, 163-164</td>
</tr>
<tr>
<td>CORAL command 68</td>
</tr>
<tr>
<td>correction input files 203</td>
</tr>
<tr>
<td>count fields 304</td>
</tr>
<tr>
<td>CPU (central processor unit), scheduling of 2</td>
</tr>
<tr>
<td>crashes, system 389-391</td>
</tr>
<tr>
<td>Create and Map Section ($CRMPSC) 285, 360</td>
</tr>
<tr>
<td>CREATE command 68-69</td>
</tr>
<tr>
<td>Create Logical Name ($CRELOG) 265</td>
</tr>
<tr>
<td>Create Mailbox and Assign Channel ($CREMBX) 271, 382-383</td>
</tr>
<tr>
<td>Create Process ($CREPRC) 255, 273-274</td>
</tr>
<tr>
<td>Create Virtual Address Space ($CRETVA) 284, 356</td>
</tr>
<tr>
<td>CR (return; carriage return) key 110</td>
</tr>
<tr>
<td>CTRL/C key 110</td>
</tr>
</tbody>
</table>

455
CTRL/I key 110
CTRL/K key 110
CTRL/L key 110
CTRL/O key 110
CTRL/Q key 110
CTRL/R key 110
CTRL/S key 110
CTRL/U key 110
CTRL/X key 110
CTRL/Y key 110
CTRL/Z key 110
cursors, in DATATRIEVE 170, 171
data
  hierarchies of 172
  in operating systems 3-6
  protection of 31
  sending and receiving, between
tasks 249
  in VAX-11 BLISS-32 141
Data Access Protocol (DAP) 233
Data Dictionary 167, 169
data link layer 233
data management facilities 167-186
  compatibility of 402
data processing, on VAX-11
  PL/I 158-159
DATATRIEVE 25, 167-176
DATATRIEVE Inquiry Facility 167
DATATRIEVE Report Writer
  Facility 167-168
data types
  in VAX-11 COBOL 136-137
  in VAX-11 COBOL-74 126
DATE data type 172
DEALLOCATE command 69
Deallocate Device ($DALLOC) 270
DEASSIGN command 69-70
Deassign I/O Channel
  ($DASSGN) 249-250, 267
DEBUG command 70
debuggers 25
debugging 9
  in PDP-11 BASIC-PLUS-
  2/VAX 163
  in VAX-11 BASIC 154-155
  in VAX-11 BLISS-32 146-147
  in VAX-11 COBOL 139
  in VAX-11 COBOL-74 132-133
  in VAX-11 FORTRAN 119
  in VAX-11 PL/I 159-160
DEBUG program 139, 147, 154,
  216-224
DECK command 70-71
Declare AST ($DCLAST) 263
Declare Exit Handler
  ($DCLEXH) 277, 391
Declare Change Mode or
  Compatibility Mode Handler
  ($DCLCMH) 283
DECLARE statement 154
DECnet products 21-22, 229-233
  PDP-11 compatibility and 399,
  400
DECnet VAX 234-237, 247
defaults
  qualifiers for 57-58
  in VAX-11 RUNOFF 206-207
DEFINE command 71
DEFINE utility 310
DELETE command 45, 72
Delete Common Event Flag Cluster
  ($DLCEFC) 258-259
DELETE/ENTRY command 72
Delete Global Section
  ($DGBLSC) 285
delete (rubout) key 44-45.110
Delete Logical Name
  ($DELLOG) 265
Delete Mailbox ($DELMBX) 271,
  383
Delete Process ($DELPRC) 274
DELETE/SYMBOL command 73
Index

Delete Virtual Address Space (SDELTVA) 284
delta time 279
DEPOSIT command 73-74, 224
device data block 321
device drivers, I/O 292, 317-343
D format 304
diagnostic messages
in VAX-11 COBOL 139
in VAX-11 FORTRAN 118-119
DIFFERENCES command 74
directives 114-115
directories 6, 50-51
DIGITAL Command Language (DCL) 9, 57-110, 234
file handling using 236-237
DIGITAL Data Communications Message Protocol (DDCMP) 223
DIGITAL Network Architecture (DNA) 21-22, 232-234
directive requests 403-405
DIRECTORY command 74-75
Disassociate Command Event Flag Cluster ($DACEFC) 258
DISMOUNT command 75
DISPLAY statement 126-127
DISPLAY utility 38-39, 310
distributed computing networks 231
document formatting utility 206-211
dollar sign ($) 57, 61
domains, in DATATRIEVE 169
down-line command file loading 232
down-line system loading 232
down-line task loading 232
driver dispatch tables (DDT) 319, 331
driver prologue tables 319
DUMP command 75
dynamic access 303
dynamic page tables 348-349
dynamic strings
in PDP-11 BASIC-PLUS-2/VAX 161-162
in VAX-11 BASIC 154
E-class messages 118
EDIT command 75-76
edit mode 192-193
editors 23
EDT 194-202
SLP 202-206
SOS 189-194
EDT text editor 23, 189, 194-202
ellipses (...), in commands 58
end of data 59
enqueuing, see queuing
entering, of commands 43-45
environmental commands 60
ENVIRONMENT options 158
EOD command 76-77
EOJ command 77
ERR keywords 117
error logging 40
error processing procedures 227
ersors
in command inputs 44-45
debugging, in VAX-11 BLISS-32 146-147
debugging, in VAX-11 COBOL 135, 139
debugging, in VAX-11 COBOL-74 132
debugging, in VAX-11 PL/I 159-160
detected by VAX-11 Common Run Time Procedure Library 227
diagnostic messages for 118-119
exceptions generated by 387
fatal, system crashes and 389-391
recognition of and dealing with 40
Index

signaling and condition handling of 225
in UETP 28
VAX/VMS operating system for logging and recovery from 2, 9
escape (altmode) key 110
EVALUATE command 220
event flags 295, 381-382
 allocation of 225
 for interprocess communications 18
event flag services 16, 256-260, 381
events, special (exceptions) 287-289
exact key matches 309
EXAMINE command 77, 223
exception dispatching 338
exceptions 387-389
during AST delivery 396
exclamation marks (!) 59
exec mode 39
executable images 215-216
executable process state queues 368-369
executing
 DEBUG program 216-217
 programs 55
Exit ($EXIT) 276, 389, 391, 392
EXIT command 77-78, 224
exit control blocks (ECB) 391, 392
exit handlers 391-392
Expand Program/Control Region ($EXPREG) 283-284
expressions, in VAX-11 BLISS-32 141
EXTERNAL statement 154
external subprograms, see subprograms
external (global) symbols 114
fatal errors 118, 132, 139
 system crashes and 389-391
faulting 358
F-class messages 118
fields 6
 in DATATRIEVE 169
field validation 184
File Access Block (FAB) 311
file I/O interface 182
file maintenance command 60
file processing level 310
files 46-53
 access modes for 300-303
 attributes of 303-311
 backing up 34
 compatibility of 402
 in DATATRIEVE 169-171
definition of 5
 handling, using DECnet-VAX 234
 handling, using DIGITAL Command Language 236-237
 handling, using Record Management Services 237-240
 handling, using VAX-11
 BASIC 153
 linker and 213-214
 manipulation of, in VAX-11
 FORTRAN 117
 names and types of, in SOS text editor 190
 organization of, in VAX-11 COBOL-74 125-126
 RMS organization of 296-300
 shared 384
 SLP editing of 203-204
 in VAX-11 COBOL 137-138
file specifications (file-spec) 46-47
file types 190
flags 209-210
FMS (Forms Management System) 183-186
Forces Exit ($FORCEx) 276-277
fork blocks 16, 322
fork processes 16, 321-323
formats
Index

for command 57-60, 62-110
for command strings 180-181
for file specification 46-47
record 304-305
format statements 122
Formatted ASCII Output
($FAO) 268
Formatted ASCII Output with List Parameter ($FAOL) 268
formatting 206-211
Form Driver (FDV) 185
Form Editor (FED) 185, 186
Forms Management System (FMS) 183-186
Form Utility (FUT) 185
FORTRAN 22
I/O operations requested in 324-325
FORTRAN IV 22
FORTRAN-66 117
FORTRAN-77 116
FORTRAN command 78-79
frame pointer (FP) 389
free page lists 358, 375
free pages 375, 376
full processes 16
function decision table (FDT)
routine 337-341
function decision tables (FDT) 319, 328, 330, 331
functions
in PDP-11 BASIC-PLUS-
2/VAX 161
in VAX-11 BASIC 152-153
in VAX-11 BLISS-32 143-145
in VAX-11 FORTRAN 118
general utility (LIB$) 226, 227
generic key matches 309
Get I/O Channel Information
($GETCHN) 270
Get I/O Device Information
($GETDEV) 270
Get Job/Process Information
($GETJPI) 279
Get Message ($GETMSG) 273
Get Time ($GETTIM) 280
global section descriptor
(GSD) 360-361
global sections 14-15, 360-361
shared areas of memory and 383-384
global section table index 361
global section tables 361
global (external) symbols 114
GO command 222
GOTO command 79
graphic formatting 208-209
group logical names 264
groups 30-31
hardware, exceptions generated by 387, 388
HELP command 45-46, 79-80
HELP facility 60
in EDT text editor 195
in VAX-11 Forms Management System 184
hexadecimal system 8
Hibernate ($HIBER) 274-275
hierarchies, data 172
hyphens (-) 58-59
idle mode 39
IF command 80
I key 110
image activation 361-362
image activators 14, 354-355, 361-362
images
definition of 212
initialization of 212
known 34-35, 360
produced by linker 215-216
Index

I/O system services 2, 8-9, 17-19, 266-273, 289-315
JOB command 82-83
$JOB command 37
journal processing 196
justifying 206
K, definition of 8
kernel mode 39
  crashes in 389
  exception dispatching in 288
keyfields 305
key matches 309
keypad, for EDT text editor 197-198, 200-1
keys (keypad)
  redefining 195-197
  terminal function 110
keys (record)
  defining 305-306
  in indexed file organization 299-300
keywords 44
  in DATATRIEVE 171
K key 110
known images 34-35, 360
labels 57
language-independent support (OTSS$) 226
language name commands 60-61
language processors 2
languages 5
  DIGITAL Command 9, 57-100
    programming 22, 113-165
language-specific support (FOR$, BAS$) 226
LET statement 163
LEXICAL functions 142
librarians 24
libraries
  macro 116
  object module 214
shareable image 215
in VAX-11 BASIC 154
in VAX-11 BLISS-32 146
in VAX-11 COBOL 138
in VAX-11 COBOL-74 130
VAX Common Run Time Procedure Library* 224-227
in VAX-11 FORTRAN 118
in VAX-11 PL/I 160
LIBRARY command 83
LIB$SIGNAL procedures 227
limits 31, 32
line mode 196, 198-199
lineprinters
  I/O driver source program listing
    for 335-337
  I/O requests to 327-335
LINKAGE declarations 142
LINK command 55, 83-84, 216
linkers 8, 24, 55, 211-216, 361
linking
  DEBUG with user programs 216
    in virtual memory 354
LINK/RSX-11 command 84
list formatting 208-209
listing control directives 115
L key 110
Lock Pages in Memory
  ($LCKPAG) 286
Lock Pages in Working Set
  ($LKWSET) 285
logical I/O mode 294
logical links 229, 233
  between tasks, creating 240-242, 244, 247-249
logical names 35, 51-53
logical name services 17, 263-265
logical unit numbers 225
login procedure 84-85
LOGOUT command 46, 85
longword index (offset) 359
Index

longwords 3, 368
condition handler and 389
in exit handlers 391, 392
lowercase characters 59
LP11-R lineprinter 238
MACRO (assembler language) 113
nontransparent intertask communications in 245-246
system services called by 255
transparent intertask communications in 243-245
MACRO-11 (assembler language) 22, 113
MACRO command 85
macro definitions 115-116
macros
calls for 116, 245, 250, 255
VAX-11 BLISS-32 146
macro symbols 114
mailboxes 18, 235, 245, 381-383
Map Global Section
($MGBLSC) 285, 361, 362, 384
mathematical functions
(MTHS$) 226
MCR command 85-86
mechanism arrays 389-391
memory allocation algorithm 212
memory
balancing available page count in 375
interface of linker to 213
management of 2, 13-15, 283-286
shared areas of 383-384
virtual 13, 345-363
memory management services 17, 383-386
MERGE facility 25
messages
diagnostic 118-119, 139
interrupt 249
mailboxes and 383
in nontransparent intertask communications 235
sending and receiving 242, 244-245
task 246-247
mnemonics 5
modes of operation
of EDT text editor 196-197
for I/O transfers 204
for record access 300-303
of SOS text editor 191-194
modified page lists 358, 375
modified pages 375-376
modify bits 358
modules
linking 211
of VAX/VMS 3
monitoring, of system activity 38-39
MOUNT command 86
MUX200/VAX Multiterminal Emulator 251-252
names
command 57
file 47-48
file, in SOS text editor 190
logical 35, 51-53
long variable and function, in PDP-11 BASIC-PLUS-2/VAX 161
long variable and function, in VAX-11 BASIC 152-153
for physical devices 49
native mode 113
nested procedures 171
Network Ancillary Control Process (NETACP) 235-236
network application layer 233
network command terminals 232
Network Connect Block (NCB) 248
networks 2, 229-233
network service layer 233
Network Services Protocol (NSP) 233
nodes 229
NOKEYPAD commands 197

462
nontransparent intertask communications 235-236
using MACRO 245-246
NO prefix 58
normal processes 365
note formatting 208-209
object module files 213-214
object module libraries 214
object modules 55
OCCURS clause 172
octal system 8
offset (longword index) 359
O key 110
ON command 86-87
OPEN command 87-88
OPEN statement 117
operating systems 1-9
operators, SLP 204
operator's log file 39
output files, for SLP editor 203
output file specification parameters 180
outputs, of linker 215-216
outswapping 365, 375-378
overlays 402
owners (users) 31
page faults 356-360, 362
page formatting 207
page frame numbers (PFN) 356-358, 362
pagers 14, 356-359, 362
pages 3, 13, 345, 348
free and modified 375
page table entries (PTE) 13, 349, 355, 356, 359-360
page tables 345, 349, 353-356
paging 355-360
in system space 363
parameters
command 60
for free pages 375
installation setting of 27-29
used in commands 44, 58
PARAMETER statement 118
PASCAL 113, 155, 156
PASCAL command 88
PASSWORD command 88-89
PCB see process control blocks
PDP-11 BASIC-PLUS-2/VAX 113, 151, 155, 160-163
PDP-11 COBOL 125, 139
PDP-11 CORAL-66/VAX 22
PDP-11 FORTRAN 113
PDP-11 FORTRAN-IV-PLUS/VAX 22
PDP-11 MACRO (assembler language) 113
PDP-11 systems 1
VAX/VMS compatibility with 399-405
PERFORM statement 134, 136
permanent global sections 34-35
permanent symbols 114
physical addresses 248
physical devices, names for 49
physical I/O mode 294
physical link layer 233
physical links 229
PL/I 22, 113, 158
plus sign (+) 59
pointers, in page tables 354
primary keys 305
PRINT command 89
print queues 37
PRINT USING statement 153, 162
priorities 31, 32
increments in 373-374
Index

for process scheduling 15, 365
private volumes 34
privileges 31, 32, 255-256
change mode services for 18
Procedure Calling Standard 159
procedure editor 173
procedures
command 61-62
definition of 12
login 84-85
nested 171
in VAX-11 FORTRAN 118
process control blocks (PCB) 12,
350-352
AST enqueuing and 394
event flags in 16
priority scheduling in 373
in process scheduling 15, 365-
367
process control services 17, 273-
279
process control structures 351-354
processes 12, 349-351
creation of 378-379
definition of 2
scheduling of 15, 365-374
system 16
see also interprocess
communications
process headers (PHD) 350-354
process identification 12
process logical names 263-264
process page tables 349
process-private pages 359-360
process section tables 353, 355
process space shareable
memory 360-362
process states 366
process state transition 369-370
program development 53-55
program development tools 23-25
program logical address space
(PLAS) 402
programming
interfaces for 291-292
intertask communications
procedures for 247-250
in VAX-11 SORT 182
programming languages 22, 113-
165
program
interactive 126-127
operated on RMS files 306, 307
program, source see source
programs
program sectioning 116
program sections (psects) 354
protocol emulators (internets) 229,
250-252
PSECT declarations 142
public files, management of 33-35
public volumes, management of 33-
35
PURGE command 89
Purge Working Set
($PURGWS) 286
Put Message ($PUTMSG) 273
QIO see Queue I/O Request
Q key 110
quadwords 3
qualifiers 44, 57-58
quantum control 371-372
Queue Input Request and Wait For
Event Flag ($INPUT) 268
Queue I/O interface 294-295
Queue I/O Request ($QIO) 10, 267,
289, 292-295, 323-326
sample of 327-335
Queue I/O Request and Wait For
Event Flag ($QIOW) 267
Queue Output Request and Wait For
Event Flag ($OUTPUT) 268

464
queuing
  of asynchronous system trap services 394
  batch, print, and terminal 36-37
  in process scheduling 15, 366-371
quotas 255-256
random record access mode 301-302
READ command 90
Read Event Flags ($READEF) 259
real-time environments 19-21
real-time processes 365, 366
  immune to quantum-end events 371
Record Access Block (RAB) 311
record access modes 300-3
record collections 169-71
Record File Address (RFA) access modes 302-303, 308
record I/O interface 182
record locking 307
Record Management Services (RMS) 6, 17-19 289-92, 296-314, 324
  DATATRIEVE and 25, 167
  DECrete-VAX and 234
  file handling using 237-340
  in VAX-11 COBOL 137
  in VAX-11 PASCAL 156
  in VAX-11 PL/I 158, 159
record processing level 310
records
  access modes for 300-303
  attributes of 303-311
  in DATATRIEVE 169-171
  definition of 5
  in VAX-11 COBOL 137-138
record sort 179
reference modification 137
REFORMAT utility program 133, 140
relative files 298-299, 307, 308
random record access to 302
sequential record access to 301
REM (remark) statement 152
RENAME command 90
repeat blocks 115-116
report writers 167-168
REQUEST command 90-91
requests
  I/O 19, 289, 292-296, 327-335
  in nontransparent intertask communications 235
  RSX-11M directive 403-405
require files 146
rescheduling interrupts 372
RESERVE AREAS clause 137
RESERVE clause 126
resource allocation commands 60
Resource Allocation Section (LIB$) 225
resource-sharing networks 230
RESTORE utility 310
Resume Process ($RESUME) 276
return (CR; carriage return) key 110
RFRMT (reformat) utility program 133, 140
R key 110
R key 45, 110
RMS see Record Management Services
routing 232
RSX-11M/S program development system 399-405
RTL (Common Run Time Procedure Library) 24, 160, 224-227
rubout (delete) key 45, 110
RUN (image) command 55, 91
RUNOFF 206-211
Run Time Procedure Library (RTL) 24, 160, 224-227

465
Index

schedulers 372, 373
Schedule Wakeup ($SCHDWK) 275-276, 282
scheduling, of processes 15, 365-374
scientific applications 159
security, User Identification code for 31
segmented keys 305
Send Message to Accounting Manager ($SNDACC) 272
Send Message to Error Logger ($SNDERR) 272
Send Message to Operator ($SNDOPR) 272
Send Message to Symbiont Manager ($SNDSCMB) 272
sequential files 298, 307, 308
random record access to 302
sequential record access to 300-301
sequential record access mode 300-301
services
  I/O 2, 8-9, 289-315
  system 16-18, 255-287, 382
Set AST Enable ($SETAST) 263
SET CARD-READER command 92
SET command 91-97
  in DEBUG facility 218-220
  in EDT text editor 196
SET CONTROL-Y command 92
SET DEFAULT command 93
Set Event Flag ($SETEF) 259
Set Exception Vector ($SETENV) 283, 388
SET MAGTAPE command 93
SET ON command 93-94
Set Power Recovery AST ($SETPRA) 263
Set Priority ($SETPRI) 278
Set Privileges ($SETPRV) 279
SET PROCESS command 94
Set Process Name ($SETPRN) 278
Set Process Swap Mode ($SETSWM) 286
SET PROTECTION command 94-95
Set Protection on Pages ($SETPRT) 286
SET QUEUE/ENTRY command 95
Set Resource Wait Mode ($SETRWM) 278
SET RMS-DEFAULT command 95-96
Set System Service Failure Exception Mode ($SETSFM) 283
Set System Time ($SETIME) 282
SET TERMINAL command 96
SET TERMINAL/LOWER command 59
Set Timer ($SETIMR) 281
SET VERIFY command 96-97
SET WORKING-SET command 97
shareable image files 214
shareable image libraries 215
shareable images 215-216, 383, 384
shareable image symbol tables 214
shareable regions 402
shared files 381, 384
shared programs
  RMS 307
  in VAX-11 BASIC 154
  in VAX-11 COBOL 138
  in VAX-11 FORTRAN 118
shell processes 15, 378-379
SHOW command 97-106
  in DEBUG facility 221
  in EDT text editor 196
SHOW DAYTIME command 98
SHOW DEFAULT command 47, 98

466
SHOW DEVICES command 99
SHOW LOGICAL command 99-100
SHOW MAGTAPE command 100
SHOW NETWORK command 100-1
SHOW PRINTER command 101
SHOW PROCESS command 101
SHOW PROTECTION command 102
SHOW QUEUE command 102-103
SHOW QUOTA command 103
SHOW RMS_DEFAULT command 103
SHOW STATUS command 103-104
SHOW SYMBOL command 104
SHOW SYSTEM command 105
SHOW TERMINAL command 105
SHOW TRANSLATION command 106
SHOW WORKING_SET command 106
signaling 225
simple keys 305
S key 110
SLP listing files 203
SLP output files 203-204
SLP text editor 23, 189, 202-206
small processes 16
software
exceptions generated by 387-88
for VAX-11 systems 11-25
RSX-11M/S 399
software process control 12
software process control blocks (PCB) 351, 352, 365, 366
quantum control in 371
see also process control blocks
solicited messages 246
SORT command 106
SORT/MERGE module 138
SORT utility 25, 176-183
SOS text editor 23, 189-194
source programs
for lineprinter I/O drivers 335-337
in PDP-11 BASIC-PLUS-2/VAX 161
in VAX-11 BASIC 152
in VAX-11 COBOL 139-140
in VAX-11 COBOL-74 133
source translator utility 139
special event handling 137-196
specification files 181
specification statements 122
spooling 36-37
Start I/O routine 332-333
STARTUP.COM 35, 36
start-up file 195
statements
conditional compilation of 119
in VAX-11 FORTRAN 121-122
state queue headers 366-367
state queues 365-371
outswapping and 378
STEP command 223
STOP command 107, 163
string functions 154
string keyfields 305
string manipulation 126
string processing (STR$) 226
STRING verb 137
structured programming 135-136
structures, in VAX-11 BLISS-32 142
subexpressions, in DATATRIEVE 176
subject-matter formatting 207-208
SUBMIT command 107-108
subprograms
external, in VAX-11 BASIC 153
external, in VAX-11 COBOL 138
external, in VAX-11 COBOL-74 131
in PDP-11 BASIC-PLUS-
Index

2/VAX 162
user-written, in VAX-11 FORTRAN 122-123
subqueues 368
subroutines, SORT 181
summary longwords 368
SUPER mode 39
support facilities 189-227
Suspend Process ($SUSPND) 276
swappers 359, 374, 375
swapping 350-351, 362-363, 365
working set 374-379
swap scheduling algorithm 376
SYE, error reports generated by 40
Symbolic Debuggers 25
VAX-11 BLISS-32 and 147
VAX-11 PL/I and 160
symbolic references 211-212
symbols, in VAX-11 MACRO 114
symbol tables 114
synchronization, interprocess 18
SYNCHRONIZE command 108
syntax, for command procedures 61-62
SYS$BATCH 37
SYS$COMMAND 53
SYS$DISK 53
SYS$ERROR 53
SYS$GEN parameter 358
SYS$INPUT 52
SYS$OUTPUT 52
SYSTARTUP.COM 36
system base registers (SBR) 13, 349
system crashes 389-391
system defined logical names 52-53
system events 366, 369-371
system images 215
system length registers (SLR) 13, 349
system logical names 264-265
system managers 27-40
system page tables (SPT) 13, 349
system procedures 226
system processes 16
system services 16-18, 255-287
for event flag handling 382
systems of accounts 29-33
system users 31, 43-55
tables
in DATATRIEVE 172
in device drivers 319
logical name 263-265
shareable image symbol 214
tables of contents 210
tag sort 179
task messages 246-47
tasks
creating logical links between 240-242, 244, 247-249
definition of 229
see also interprocess communications
TCX program 210
terminal function keys 110
terminal queues 36
terminals
DECnet-VAX and 234
VT100 183-185, 196, 197
text editors 23, 189-206
time conversion services 17, 279-282
time-critical processes 365
timer services 17, 279-282	imeshared processes 365
title formatting 207
TOC program 210
TOTAL statement 172
Index

traceback facility 132, 135, 139, 159, 162
Translate Logical Name
($STRNLOG) 265
Translation Not Valid Faults 356
transparent intertask communications 234-235
using MACRO 243-245
transportability features 147
transport layer 233
tutorial software 173
TYPE command 108
U key 110
†U key 45
unit control block (UCB) 321, 331-333
UNLOCK command 108-109
Unlock Page From Memory
($UNLPAG) 286
Unlock Pages From Working Set
($ULWSET) 285
unsolicited messages 246-247
UNSTRING verb 137
Unwind Call Stack ($UNWIND) 283, 391
Update Section File on Disk
($UPDSEC) 285
updating, command procedure for 28
user authorization file (UAF) 29-30, 255
User Authorization Program
(AUTHORIZE) 33
user-defined symbols 114
User Environment Test Package
(UETP) 28
User Identification Code (UIC) 30-31
user layer 233
user mode 39
user processes 12, 16
users
categories of 31
system 43-55
user-written subprograms 122-123
utilities 25
RMS 310-311
value assignments 141
value validation 173
variable-length records 304
variables
in PDP-11 BASIC-PLUS-2/VAX 161
in VAX-11 BASIC 152-53
variable-with-fixed-control (VFC) records 304-305
VAX-11 2780/3780 Protocol
Emulator 250-251
VAX-11/780 113
VAX-11 BASIC 22, 113, 151-155
VAX-11 BLISS-32 22, 113, 141-147
VAX-11 COBOL 22, 113, 133-141
VAX-11 COBOL-74 22, 125-133, 139
VAX-11 Common Run Time
Procedure Library (RTL) 24, 160, 224-227
VAX-11 CORAL-66 163-165
VAX-11 DEBUG program 216-224
VAX-11 Forms Management System (FMS) 183-186
VAX-11 FORTRAN 22, 113, 116-124
intertask communications
language 240-243
VAX-11 MACRO (assembler language) 5, 22, 113-116
VAX-11 PASCAL 22, 113, 155-158
VAX-11 PL/I 22, 113, 158-160
VAX-11 RMS (Record Management Services) 311-314
see also Record Management Services

469
Index

VAX-11 RUNOFF 206-211
VAX-11 Run Time Procedure Library (RTL) 24, 160, 224-227
VAX-11 SORT 25, 176-183
VAX-11 Symbolic Debugger 147, 160

VAX-11 systems
drivers for 317
installation of 27-29
PDP-11 compatibility with 399
processes on 351
program development tools for 23-25
programming environments on 113
software for 11-25
utilities for 25
VAX/VMS operating system for 1-2
virtual memory on 345

VAX/VMS operating system for 1-2
VAX/VMS operating systems 1-9
condition handlers on 388
data management facilities supported on 167
device drivers and 317
high level languages supported on 113
installation of 27-29
interprocess communications on 381
I/O devices supported on 389
I/O processing on 18-19
languages available on 22
monitoring system activity on 38-39
paging on 357
PDP-11 compatibility with 399-405
processes on 351, 355-356
protocol emulators supported on 250-52
real-time environments on 19-21
scheduler on 365, 366
support facilities for 255
system services for 255

VAX-11 BASIC supported on 155
virtual memory on 345, 349
version numbers 49
video terminals, VAX-11 Forms Management System for 183-186
views 172
virtual address space 7, 345-349
virtual arrays 162
virtual I/O mode 294
virtual memory 13, 345-363
allocation of 211, 225
memory management services for 283

Virtual Memory Systems (VMS)
PDP-11 compatibility with 399
virtual address space in 7
VT100 terminals 183-185, 196
EDT keypad on 197
WAIT command 109
Wait for Logical AND of Event Flags ($WFLAND) 260
Wait for Logical OR of Event Flags ($WFLOR) 260
Wait For Single Event Flag ($WAITFR) 259, 295
wait state queue headers 367
Wake ($WAKE) 275
W-class messages 118-119
window editing 195
WITH NO ADVANCING clause 126-127
words 3
working set lists 358-359
working sets 12, 350
working set swappers 15, 362-363
working set swapping 374-379
world users 31
WRITE command 109-110
X key 110
Y key 110
Z key 110

470
Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.)

What features are most useful?

Does the publication satisfy your needs?

What errors have you found?

Additional comments

Name

Title

Company

Address

City State Zip

(staple here)
digital
HANDBOOK SERIES
Microcomputer Processor
Microcomputer Interfaces
PDP-11 Processor
PDP-11 Software
Peripherals
Terminals and Communications
VAX-11 Architecture
VAX Software
VAX-11/780 Hardware
DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754. Telephone (617) 897-5111—SALES AND SERVICE OFFICES: UNITED STATES—ALABAMA, Birmingham, Huntsville • ARIZONA, Phoenix, Tucson • CALIFORNIA, El Segundo, Oakland, Sacramento, San Diego, San Francisco, Santa Ana, Santa Barbara, Santa Clara • COLORADO, Colorado Springs, Denver • CONNECTICUT, Fairfield, Meriden • FLORIDA, Miami, Orlando, Tampa • GEORGIA, Atlanta • HAWAII, Honolulu • ILLINOIS, Chicago, Peoria, Rolling Meadows • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, New Orleans • MARYLAND, Baltimore, Odenton • MASSACHUSETTS, Boston, Springfield, Waltham • MICHIGAN, Detroit • MINNESOTA, Minneapolis • MISSOURI, Kansas City, St. Louis • NEBRASKA, Omaha • NEW HAMPSHIRE, Manchester • NEW JERSEY, Cherry Hill, Fairfield, Princeton, Somerset • NEW MEXICO, Albuquerque, Los Alamos • NEW YORK, Albany, Buffalo, Long Island, Manhattan, Rochester, Syracuse, Westchester • NORTH CAROLINA, Chapel Hill, Charlotte • OHIO, Cincinnati, Cleveland, Columbus, Dayton • OKLAHOMA, Tulsa • OREGON, Portland • PENNSYLVANIA, Allentown, Harrisburg, Philadelphia, Pittsburgh • RHODE ISLAND, Providence • SOUTH CAROLINA, Columbia • TENNESSEE, Knoxville, Nashville • TEXAS, Austin, Dallas, El Paso, Houston • UTAH, Salt Lake City • VERMONT, Burlington • VIRGINIA, Richmond • WASHINGTON, Seattle • WASHINGTON, D.C. • WEST VIRGINIA, Charleston • WISCONSIN, Milwaukee • INTERNATIONAL—ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Darwin, Hobart, Melbourne, Perth, Sydney, Tasmania, Townsville • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Rio de Janeiro, Sao Paulo • CANADA, Calgary, Edmonton, Halifax, Kanata, London, Montreal, Ottawa, Quebec City, Toronto, Vancouver, Winnipeg • CHILE, Santiago • DENMARK, Copenhagen • EGYPT, Cairo • FINLAND, Helsinki • FRANCE, Lyon, Marseille, Paris, Puteaux • HONG KONG • INDIA, Bombay • IRAN, Tehran • IRELAND, Dublin • ISRAEL, Tel Aviv • ITALY, Milan, Rome, Turin • JAPAN, Osaka, Tokyo • MEXICO, Mexico City • NETHERLANDS, Amsterdam, Hague, Utrecht • NEW ZEALAND, Auckland, Christchurch, Wellington • NORTHERN IRELAND, Belfast • NORWAY, Oslo • PERU, Lima • PUERTO RICO, San Juan • SAUDI ARABIA, Jeddah • SCOTLAND, Edinburgh • SINGAPORE • SOUTH KOREA, Seoul • SPAIN, Madrid • SWEDEN, Gothenburg, Stockholm • SWITZERLAND, Geneva, Zurich • TAIWAN, Taoyuan • UNITED KINGDOM, Birmingham, Bristol, Ealing, Epsom, Leeds, Leicester, London, Manchester, Reading, Welwyn • VENEZUELA, Caracas • WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart • YUGOSLAVIA, Ljubljana