TECHNICAL SUMMARY

digital

TECHNICAL SUMMARY

digital

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 document.

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 its description.

The software described in this document is furnished under a license for use only on a single computer system and can be copied only with the inclusion of DIGITAL's 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 this software shall at all times remain in Digital Equipment Corporation.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

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.

Copyright©1980 Digital Equipment Corporation
Maynard, Massachusetts

Contents

1 INTRODUCTION

2 THE SYSTEM

INTRODUCTION 2-1

COMPONENTS 2-1
Processor 2-1
Virtual Memory Operating System 2-1
Peripherals 2-2

PERFORMANCE 2-3

RELIABILITY 2-3
Data Integrity 2-3
System Availability 2-4
System Recovery 2-4

FLEXIBILITY 2-4
Flexiblity in the Operating Environment 2-5
Flexibility in Programming Interfaces 2-5
Programmer Productivity 2-5
Extending the System 2-5
PDP-11 Compatibility 2-5

3 THE USERS

THE APPLICATION PROGRAMMER 3-1
The Command Language 3-1
Command Procedures 3-1
DCL Command Language Summary Table 3-2
RUN Command 3-3
Programming Languages 3-4
Record Management Services 3-5

THE SYSTEM PROGRAMMER 3-5
Job and Process Structure 3-5
Multiprogramming Environment 3-6
Program Development 3-6

THE SYSTEM MANAGER 3-6
User Authorization 3-6
Privileges 3-7
Resource Quotas and Limits 3-7
Privileges Summary Table 3-7
Resource Accounting Statistics 3-8
Performance Analysis Statistics 3-8
Display Utility Program 3-8

THE SYSTEM OPERATOR 3-8
Spooling and Queue Control 3-8
Batch Processing 3-9
Online Software Maintenance 3-9
System Recovery 3-9
Error Logging and Reporting 3-9
Online Diagnostics 3-10
Remote Diagnosis 3-10

THE USER ENVIRONMENT TEST PACKAGE 3-10

APPLICATIONS EXAMPLES 3-10
Commercial System Example 3-10
Real-Time Flight Simulation Example 3-13
Design Considerations 3-13

4 THE VAX PROCESSORS

INTRODUCTION 4-1

VAX ARCHITECTURE 4-1

PROCESSING CONCEPTS FOR USER PROGRAMMING 4-1
Process Virtual Address Space 4-1
Instruction Sets 4-1
Registers and Addressing Modes 4-1
Data Types 4-1
Data Type Table 4-2
Stacks, Subroutines, and Procedures 4-4
Condition Codes 4-4
Exceptions 4-4

USER PROGRAMMING ENVIRONMENT 4-4
Process Virtual Address Space Structure 4-4
General Registers 4-5
Addressing Modes 4-5
Addressing Modes Table 4-6
Program Counter 4-6
The Stack Pointer, Argument Pointer and Frame Pointer 4-7
Processor Status Word 4-8
Handling Exceptions 4-8

NATIVE INSTRUCTION SET 4-8
Instruction Set Summary Table 4-9
Integer and Floating Point Instructions 4-11
Packed Decimal Instructions 4-11
Edit Instruction 4-11
Character String Instructions 4-12
The Index Instruction 4-12
Variable-Length Bit Field Instructions 4-12
Queue Instructions 4-13
Address Manipulation Instructions 4-13
General Register Manipulation Instructions 4-13
Branch, Jump and Case Instructions 4-13
Subroutine Branch, Jump, and Return Instructions 4-14
Procedure Call and Return Instructions 4-15
Miscellaneous Special-Purpose Instructions 4-15

COMPATIBILITY MODE 4-15
PDP-11 Program Environment 4-15
PDP-11 Instruction Set 4-16

PROCESSING CONCEPTS FOR SYSTEM PROGRAMMING 4-16
Context Switching 4-16
Priority Dispatching 4-16
Virtual Addressing and Virtual Memory 4-17

SYSTEM PROGRAMMING ENVIRONMENT 4-17
Processor Status Longword 4-17
Processor Access Modes 4-17
Protected and Privileged Instructions 4-18
Memory Management 4-18
Virtual to Physical Page Mapping 4-20
Exception and Interrupt Vectors 4-22
Interrupt Priority Levels 4-22
I/O Space and I/O Processing 4-23
Process Context 4-23

CONSOLE 4-25
THE VAX-11/780 PROCESSOR 4-27
INTRODUCTION 4-27
VAX-11/780 PROCESSOR COMPONENTS 4-28
VAX-11/780 Console 4-28
VAX-11/780 Memory Interconnect 4-28
VAX-11/780 MAIN MEMORY AND CACHE SYSTEMS 4-29
Main Memory 4-29
Memory Cache 4-29
Address Translation Buffer 4-29
Instruction Buffer 4-29
I/O CONTROLLER INTERFACES 4-29
VAX-11 MASSBUS Interface 4-29
VAX-11/780 UNIBUS Interface 4-30
Data Throughput 4-30
VAX-11/780 FLOATING POINT ACCELERATOR 4-30
THE VAX-11/750 PROCESSOR 4-31
INTRODUCTION 4-31
VAX-11/750 PROCESSOR COMPONENTS 4-31
VAX-11/750 Console 4-31
VAX-11/750 Main Memory 4-32
VAX-11/750 Cache Systems 4-32
Peripheral Controller Interfaces 4-32
VAX-11 MASSBUS Interface 4-33
VAX-11/750 UNIBUS Interface 4-33

5 THE PERIPHERALS

COMPONENTS 5-1
MASS STORAGE PERIPHERALS 5-1
Disk Device Table 5-1
Disks 5-2
Magnetic Tape 5-2
UNIT RECORD PERIPHERALS 5-3
LP11 Line Printers 5-3
LA11 Line Printer 5-3
CR11 Card Reader 5-3
TERMINALS AND INTERFACES 5-3
LA120 Hard Copy Terminal 5-4
LA36 Hard Copy Terminal 5-4
VT100 Video Terminal 5-4
DZ11 Terminal Line Interface 5-5
REAL-TIME I/O DEVICES 5-5
LPA11-K 5-5
DR11-B 5-5
DR780 5-5
INTERPROCESSOR COMMUNICATIONS LINK 5-6
DMC11 5-6
MA780 5-6
CONSOLE STORAGE DEVICES 5-6
RX01 Floppy Disk Cartridge 5-7
TU58 Tape Cartridge 5-7

6 THE OPERATING SYSTEM

INTRODUCTION 6-1
COMPONENTS AND SERVICES 6-1
PROCESSING CONCEPTS 6-2
Programs and Processes 6-2
Resource Allocation 6-4
Privileges 6-4
Protection 6-4
USER PROCESS ENVIRONMENT 6-5
Virtual Address Space Allocation 6-5
System Services 6-5
System Services Table 6-7
I/O System Services 6-10
Local Event Flags 6-11
Asynchronous System Traps 6-11
Exception Conditions and Condition Handlers 6-11
INTERPROCESS COMMUNICATION AND CONTROL 6-12
Process Control Services 6-12
Interprocess Communication Facilities 6-12
Common Event Flags 6-13
Mailboxes 6-13
Shared Areas of Memory 6-14
Interprocessor Communication Facility 6-14
MEMORY MANAGEMENT 6-14
Mapping Processes into Memory 6-14
Process Virtual Memory and Working Set 6-15
Paging 6-15
Virtual Memory Programming 6-17
VAX/VMS Memory Management Services 6-17
PROCESS SCHEDULING 6-18
System Events and Process States 6-18
Priority: Real-Time and Normal Processes 6-18
Scheduling Real-Time Processes 6-18
Scheduling Normal Processes 6-19
Swapping and the Balance Set 6-19
VAX/VMS Process Control Services 6-19
I/O PROCESSING 6-19
Programming Interfaces 6-20
Ancillary Control Processes 6-20
I/O Processing Interfaces Table 6-21
Device Drivers 6-21
I/O Request Processing 6-21
COMPATIBILITY MODE OPERATING ENVIRONMENT 6-22
User Programming Considerations 6-22
File System and Data Management 6-23
Command Languages 6-23

7 THE LANGUAGES

INTRODUCTION 7-1
VAX COMMON LANGUAGE ENVIRONMENT 7-1
Symbolic Debugger Interface 7-1
Symbolic Traceback Facility 7-1
Common Run Time Library 7-1
VAX Calling Standard 7-1
Exception Handling 7-1
VAX-11 RMS 7-1
VAX-11 FORTRAN 7-2
Introduction 7-2
File Manipulation 7-2
Simplified I/O Formats 7-2
Character Data Type 7-2
Language Extensions to FORTRAN-77 Table 7-3
FORTRAN-77 Features Table 7-4
Source Program Libraries 7-5
Calling External Functions and Procedures 7-5
Shareable Programs 7-5
Diagnostic Messages 7-5
Compiler Operation and Optimizations 7-5
Debugging Facilities 7-7
Conditional Compilation of Statements 7-7
Symbolic Traceback 7-7
VAX-11 COBOL 7-7
Introduction 7-7
General Characteristics 7-8
Structured Programming 7-8
Data Types 7-9
Files and Records 7-9
SORT/MERGE Facility 7-10
Symbolic Characters Facility 7-10
CALL Facility 7-11
Source Library Facility 7-11
Shareable Programs 7-11
Debugging COBOL Programs 7-11
Source Translator Utility 7-12
Source Program Formats 7-13
Additional Features 7-13
Sample VAX-11 COBOL Code 7-13
VAX-11 BASIC 7-14
Introduction 7-14
General Characteristics 7-15
Structured Programming 7-15

Data Types 7-15
Data Types Table 7-15
Declarations 7-15
Files and Records 7-17
Symbolic Characters 7-17
CALL Facility 7-18
Shareable Programs 7-18
Developing BASIC Programs 7-18
The LOAD Command 7-18
Error Handling 7-18
Migration to VAX/VMS 7-18
Performance 7-20
Additional Functions 7-20

VAX-11 PL/I 7-21

Introduction 7-21
The G (General-Purpose) Subset 7-21
Program Structure 7-22
Program Control 7-22
Storage Control 7-22
Input/Output 7-22
Attributes and Pictures 7-22
Built-in Functions and Pseudovariables 7-22
Expressions 7-23
VAX-11 Extensions to the G Subset Standard 7-23
Procedure-Calling and Condition-Handling Extensions 7-23
Support of VAX-11 Record Management Services 7-23
Miscellaneous Extensions and Deviations 7-23
Full PL/I Features Supported 7-23
Implementation-Defined Values and Features 7-24
VAX-11 PL/I Programming Example 7-24

VAX-11 PASCAL 7-26

Introduction 7-26
Sample VAX-11 Pascal Code 7-27
Compiler Listing Format 7-27
Source Code Listing 7-27
Machine Code Listing 7-29
Cross-Reference Listing 7-29

VAX-11 BLISS-32 7-29

Introduction 7-29
Features of BLISS-32 7-29
VAX-11 Machine-Specific Function Table 7-30
The VAX-11 BLISS-32 Compiler 7-31
Library and Require Files 7-31
Macros 7-31
Debugging 7-31
Transportability Features 7-31
VAX-11 BLISS-32 Sample Program 7-32

VAX-11 CORAL 66 7-32

VAX-11 MACRO 7-33
Symbols and Symbol Definitions 7-33
Directives 7-33
Listing Control Directives 7-34
Conditional Assembly Directives 7-34
Macro Definitions and Repeat Blocks 7-34
Macro Calls and Structured Macro Libraries 7-34
Program Sectioning 7-34

PDP-11 BASIC-PLUS-2/VAX 7-34
Program Format 7-35
Long Variable and Function Names 7-35
Powerful File I/O 7-35
Powerful String Handling 7-35
Virtual Arrays 7-35
PRINT USING Output Formats 7-35
Subprograms and the CALL Statement 7-35
COMMON Statement 7-35
Debugging Statements 7-35

PDP-11 FORTRAN IV/VAX to RSX 7-36

MACRO-11 7-36
Symbols and Symbol Definitions 7-36
Directives 7-36

8 PROGRAM DEVELOPMENT AND SUPPORT FACILITIES

INTRODUCTION 8-1

TEXT EDITORS 8-1
File Names and File Types 8-1

SOS EDITOR 8-1
Initiating and Terminating SOS 8-1
SOS Examples 8-2

EDT EDITOR 8-2
What EDT Does 8-2

EDT SPECIAL FEATURES 8-2
Editing with a Window 8-2
Start-up File 8-2
HELP Facilities 8-2
The Keypad 8-2
Redefining Keypad Keys 8-3
The SET and SHOW Commands 8-3
Journal Processing 8-3
The EDT CAI Program 8-3
EDT Modes of Operation 8-3

SLP EDITOR 8-3
Initiating and Terminating SLP 8-3
SLP Input and Output Files 8-3
Correction Input File 8-3
SLP Output File 8-4

LINKER 8-4
The LINK Command 8-4
Virtual Memory Allocation 8-4
Resolution of Symbolic References 8-4
Image Initialization 8-4
Overview of Linker Interface to Memory Management 8-4
Linker Input 8-4
Object Module Files 8-5
Object Module Libraries 8-5
Shareable Image Files 8-5
Shareable Image Symbol Tables 8-5
Linker Output 8-5

COMMON RUN TIME PROCEDURE LIBRARY 8-5
Resource Allocation Group (LIB$) 8-5
Signal and Condition Handling 8-6
General Utility (LIB$) 8-6
Mathematical Functions (MTH$) 8-6
Language-Independent Support (OTS$) 8-6
Language-Specific Support (FOR$, BAS$) 8-6
String Processing (STR$) 8-6
System Procedures 8-6
Compiled-Code Support Procedures 8-6
Error Processing Procedures 8-6

VAX-11 SYMBOLIC DEBUGGER 8-6
DEBUG Commands 8-7

THE LIBRARIAN UTILITY 8-7
Librarian Routines 8-7
DCL LIBRARY Command 8-7

COMMAND LANGUAGE PROCEDURES 8-8
Passing Parameters to Command Procedures 8-8
Logical Commands 8-8
Lexical Functions 8-8
Command Procedure Example 8-8

DIFFERENCES UTILITY 8-9

VAX-11 RUNOFF 8-10
Filling and Justifying 8-10
Page Formatting 8-10
Title Formatting 8-10
Subject-Matter Formatting 8-10
Index and Table of Contents 8-10
Miscellaneous Formatting 8-10

9 DATA MANAGEMENT SERVICES

INTRODUCTION 9-1

FILE MANAGEMENT 9-1
File Directories and Directory Structures 9-1
File Specifications 9-1

Logical File Naming 9-3
File Management 9-4

RECORD MANAGEMENT SERVICES 9-4

RMS FILE ORGANIZATIONS 9-5
Sequential File Organization 9-5
Relative File Organization 9-5
Indexed File Organization 9-5

RMS RECORD ACCESS MODES 9-6
Sequential Record Access Mode 9-6
Random Record Access Mode 9-7
Record's File Address (RFA) Record Access Mode 9-7
Dynamic Access 9-7

FILE AND RECORD ATTRIBUTES 9-7
Record Formats 9-8
Key Definitions or Indexed Files 9-8

PROGRAM OPERATIONS ON RMS FILES 9-9
File Processing 9-9
Record I/O Processing 9-9
Block I/O Processing 9-10

RMS RUN TIME ENVIRONMENT 9-11
Run Time File Processing 9-11
Run Time Record Processing 9-11
RMS Record Locking 9-12

LANGUAGE UTILITIES 9-12

DATATRIEVE 9-12
DATATRIEVE Inquiry Facility 9-12
DATATRIEVE Report Writer Facility 9-12
Basic Commands 9-12
Terminology 9-13
Keywords 9-13
Additional DATATRIEVE Features 9-13

VAX-11 SORT/MERGE 9-14

VAX-11 SORT/MERGE FEATURES 9-15
SORT/MERGE as a Set of Callable Subroutines 9-16
File I/O Interface 9-16
Record I/O Interface 9-16
Programming Considerations 9-16

SORT/MERGE PERFORMANCE FEATURES 9-16

VAX-11 FORMS MANAGEMENT SYSTEM (FMS) 9-16
Using Forms in an Application 9-16
Developing Applications with VAX-11 FMS 9-17
Maintaining VAX-11 FMS Applications 9-17

10 DATA COMMUNICATIONS FACILITIES

INTRODUCTION 10-1

DIGITAL NETWORK ARCHITECTURE 10-2
User Layer 10-2
Network Services Layer 10-2
Data Link Layer 10-2
Pysical Link Layer 10-2

DECNET-VAX FEATURES 10-2
File Handling Using a Terminal 10-2
File Handling Using Record Management Services 10-3
Network Command Terminal 10-3
Intertask Communications 10-3

DIGITAL COMMAND LANGUAGE (DCL) FILE HANDLING 10-3

RECORD MANAGEMENT SERVICES FILE HANDLING ..10-4

SAMPLE VAX-11 FORTRAN INTERTASK COMMUNICATION 10-4
Creating a Logical Link Between Tasks 10-4
Sending and Receiving Messages 10-4
Disconnecting the Logical Link 10-4
VAX-11 FORTRAN Intertask Communication Example ..10-4

MACRO TRANSPARENT INTERTASK COMMUNICATION 10-4
Creating a Logical LInk Between Tasks 10-4
Sending and Receiving Messages 10-6
Disconnecting the Logical Link 10-6

MACRO CALLS 10-6

MACRO NONTRANSPARENT INTERTASK COMMUNICATION 10-6
Task Messages 10-7

PROTOCOL EMULATORS (INTERNETS) 10-7
VAX-11 2780/3780 Protocol Emulator 10-7
MUX200/VAX Multiterminal Emulator 10-7

APPENDIX A

APPENDIX B

APPENDIX C

GLOSSARY

INDEX

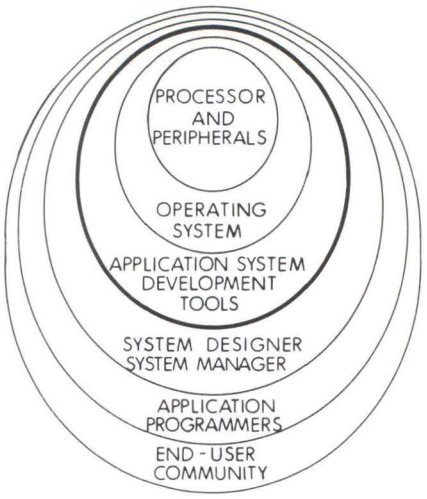
1
Introduction

The VAX Technical Summary introduces the characteristic features and capabilities of the VAX system to computer analysts and system programmers. Application programmers, and system managers and operators may also use this summary as a tool to become familiar with the components, services, and operations of the VAX system.

The intent of this summary is to familiarize the reader with the features and capabilities of the VAX (Virtual Address Extension) system. It serves as a detailed technical introduction to the growing family of VAX processors, the powerful and unique Virtual Memory operating System, VAX/VMS, and the increasing number of supported peripherals. In particular, this edition of the VAX Technical Summary introduces several significant system enhancements:

- the newest member of the VAX family of processors, the VAX-11/750
- version 2.0 of the VAX/VMS operating system
- the addition of major new languages including VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 CORAL 66, and VAX-11 BLISS-32

Although the VAX Technical Summary contains useful information for the application programmer, the system manager, and the computer operator, the level of technical detail makes the book particularly appropriate for the system programmer and/or computer system analyst.



As the reader proceeds through the technical summary, the term "VAX architecture" or simply "architecture" may seem confusing. VAX architecture is the collection of attributes that all family members have in common that assures software compatibility. For example, the architecture includes the instruction set, the addressing modes, data types, etc. Examples of attributes not included in the architecture are processor internal bus structure, implementation-specific privileged registers, execution speed, etc. As new processors are added to the VAX family, a significant part of the engineering effort will be dedicated to preserving this software compatibility. This will assure that programs written for today's VAX computers will execute on future VAX systems.

The VAX Technical Summary is designed to be read in either a selective or sequential manner. The reader might start with section 1 and continue through the book, or first glance through the table of contents to locate those topics of most interest. As an added convenience, an abstract can be found prefacing each section of the summary. Many of the system's concepts and features are repeated throughout the text in various contexts. Appendix A contains a collection of most frequently used mnemonics and their definitions.

The following paragraphs introduce the VAX Technical Summary.

The System section presents a comprehensive overview of VAX system features. This section serves as an introduction for the reader presently familiar with computer industry terminology, identifying VAX characteristics and introducing the system features described in detail throughout the remainder of the summary.

The Users section contains a description of the command language and its features. It also introduces many of the aspects of the system that support applications programming, system management, and operator control.

The VAX Family of Processors and Operating System sections provide an in-depth discussion of the system's characteristics and capabilities. The concepts developed in both sections are closely related; for example, the respective discussions of memory management, I/O processing, and the compatibility mode environment should be read together to gain a full appreciation for the system's design. These sections should prove beneficial to systems programmers or analysts already familiar with an assembly language or software executive.

Perhaps one of the most important features of the VAX system is that its programmers do not have to know assembly language to use the system effectively. Both the hardware and software contain many features that promote efficient and productive high-level language programming. High-level language programmers should find the Users, Languages, Data Management Services, and Network Services sections, of particular interest. The beginning of the Operating System section is also helpful because it introduces some of the VAX software terminology and concepts.

Also of interest to high level language programmers is the Program Development section which describes the basics of creating, editing, debugging, and executing a program written in any of the VAX high level languages. This section also contains an introduction to some of the advanced features of the command language.

If the reader is uncertain about a particular term or phrase, its definition can probably be found in the glossary at the end of the summary. The glossary does not generally contain standard computer-related terms, but it does contain most of the terms found throughout the VAX documentation that have special meanings in the context of the VAX system. The glossary is followed by a list of mnemonics that may be encountered in the text. The mnemonics list is particularly useful during the system familiarization stage.

For additional literature describing VAX features, capabilities and applications please contact the nearest DIGITAL sales office.

2
The System

The VAX system provides the performance, reliability, and programming features often found only in much larger systems. The VAX family of processors have a 32-bit architecture based on the PDP-11 family of 16-bit minicomputers. While using addressing modes and stack structures similar to those of the PDP-11, VAX provides 32-bit addressing for a 4 gigabyte program address space, and 32-bit arithmetic and data paths for processing speed and accuracy.

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

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

VAX is a highly reliable system. 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 power, hardware, or software fail.

The system is 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. The command language enables users to modify or extend their command repertoire easily, and allows applications to present their own command interface to users.

INTRODUCTION

VAX is a high performance multiprogramming computer system ideally suited for a wide variety of applications including real time, batch, time sharing, commercial, data processing, and program development. The system combines a 32-bit architecture, efficient memory management, and a virtual memory operating system to provide essentially unlimited program address space.

The processor's instruction set includes floating point, packed decimal arithmetic, and character string instructions. Many of the instructions are direct counterparts for high-level language statements. The software system supports programming languages that take advantage of these instructions to produce extremely efficient code.

The VAX/VMS virtual memory operating system provides a multiuser, multilanguage programming environment on the VAX hardware. The floating point instructions, efficient scheduler, and optional VAX-11 FORTRAN language are ideal for real-time and scientific computational environments. The optional VAX-11 COBOL language, data management services, and large capacity peripherals make the system equally suited to commercial applications. VAX/VMS supports many other optional high level languages suited for other applications. The system management facilities, command language, and program development tools provide the resources for efficient program applications development and execution. Spooling and extensive job control capabilities support batch processing.

The processor executes variable length instructions in native mode, and non-privileged PDP-11 instructions in compatibility mode. Native mode includes the PDP-11 data types, and uses addressing modes and instructions similar to those of the PDP-11. The software supports compatible languages and file and record formats.

COMPONENTS

The major components of a VAX system are:

- Processor—includes the basic central processing unit implementing the VAX architecture. The specific implementations of the VAX processors will be treated in greater detail in Section 4, The VAX Processors.
- Operating System—includes a virtual memory manager, swapper, system services, device drivers, file system, record management services, command language, and operator's and system manager's tools.
- Languages—includes the native mode languages VAX-11 MACRO and optionally, VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 BLISS-32, and VAX-11 CORAL 66. Also supported in compatibility mode are PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX, and MACRO-11. Development tools for both native and compatibility mode programs include editors, linkers, librarians, and debuggers.
- Peripherals—includes a range of small- and large-capacity disk drives, magnetic tape systems, hard copy and video terminals, line printers, card readers, and real-time I/O devices.
- Network Services—includes the DECnet-VAX network software and the DMC11 interprocessor communications link.

Processor

Architecturally, a VAX processor provides 32-bit addressing, sixteen 32-bit general registers, and 32 interrupt priority levels. The instruction set operates on integer, floating point, character and packed decimal strings, and bit fields. The instruction set supports nine addressing modes.

The processor includes an efficient memory cache resulting in greatly reduced memory access time.

Error Correcting Code (ECC) MOS memory is connected to the memory interconnect via a memory controller. Each memory controller includes a request buffer that substantially increases overall system throughput and eliminates the need for interleaving in most applications.

The processor uses two standard clocks—a programmable real-time clock used by the operating system and by diagnostics, and a time-of-year clock used for system operations. The time-of-year clock includes battery backup for automatic system restart operations.

The console is the operator's interface to the central processor. Via the console terminal, the operator can execute diagnostics, install new software, examine and deposit data in memory locations or processor registers, halt the processor, step through instruction streams, and boot the operating system.

Virtual Memory Operating System

VAX/VMS is a multiuser, multifunction virtual memory operating system that supports multiple languages, an easy to use interactive command interface, and program development tools. The VAX/VMS operating system is designed for many applications, including scientific/real-time, computational, data processing, transaction processing, and batch.

The operating system performs process-oriented paging, which allows execution of programs that may be larger than the physical memory allocated to them. Paging is handled automatically by the system, freeing the user from any need to structure the program. In the VAX/VMS operating system, a process pages only against itself—thus individual processes cannot significantly degrade the performance of other processes.

The memory management facilities provided by the operating system can be controlled by the user. Any program can prevent pages from being paged out of its working set. With sufficient privilege, it can prevent the entire working set from being swapped out, to optimize program performance for real-time or interactive environments. Sharing and protection are provided for individual 512-byte pages. The processor's memory management includes four hierarchical processor access modes that are used by the operating system to provide read/write page protection between user software and system software. The access modes from most to least privileged are kernel, executive, supervisor, and user.

VAX/VMS schedules CPU time and memory residency on a pre-emptive priority basis. Thus, real-time processes do

not have to compete with lower priority processes for scheduling services. Scheduling rotates among processes of the same priority. The scheduler adjusts the priorities of processes assigned to one of the low 16 priorities to improve responsiveness and to overlap I/O and computation. Real-time processes can be placed in one of the top 16 scheduling priorities, in which case the scheduler does not alter their priority. Their priorities can be altered by the system manager or an appropriately privileged user.

Interprocess communication is provided through shared files and shared address space, event flags, and mailboxes which are record-oriented virtual devices.

VAX/VMS provides system management facilities. A system manager and a system operator are given the tools necessary to control the system configuration and the operations of system users for maximum efficiency.

The VAX/VMS command language is easy to learn and use, and is suitable for both the interactive and batch environments. Extensive batch facilities under VAX/VMS include job control, multistream spooled input and output, operator control, conditional command branching and accounting.

Command procedures are supported by the command languages as an easy method of executing strings of frequently used sequences of commands, or creating new commands from the existing command set. Command procedures accept parameters and can include extensive control flow.

VAX/VMS provides a program development capability that includes editors, language processors, and a symbolic debugger. The VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 BLISS-32, VAX-11 CORAL 66, and VAX-11 MACRO language processors produce native code. The PDP-11 BASIC-PLUS-2/VAX,PDP-11 FORTRAN IV/VAX to RSX, and MACRO-11 language processors produce compatibility mode code.

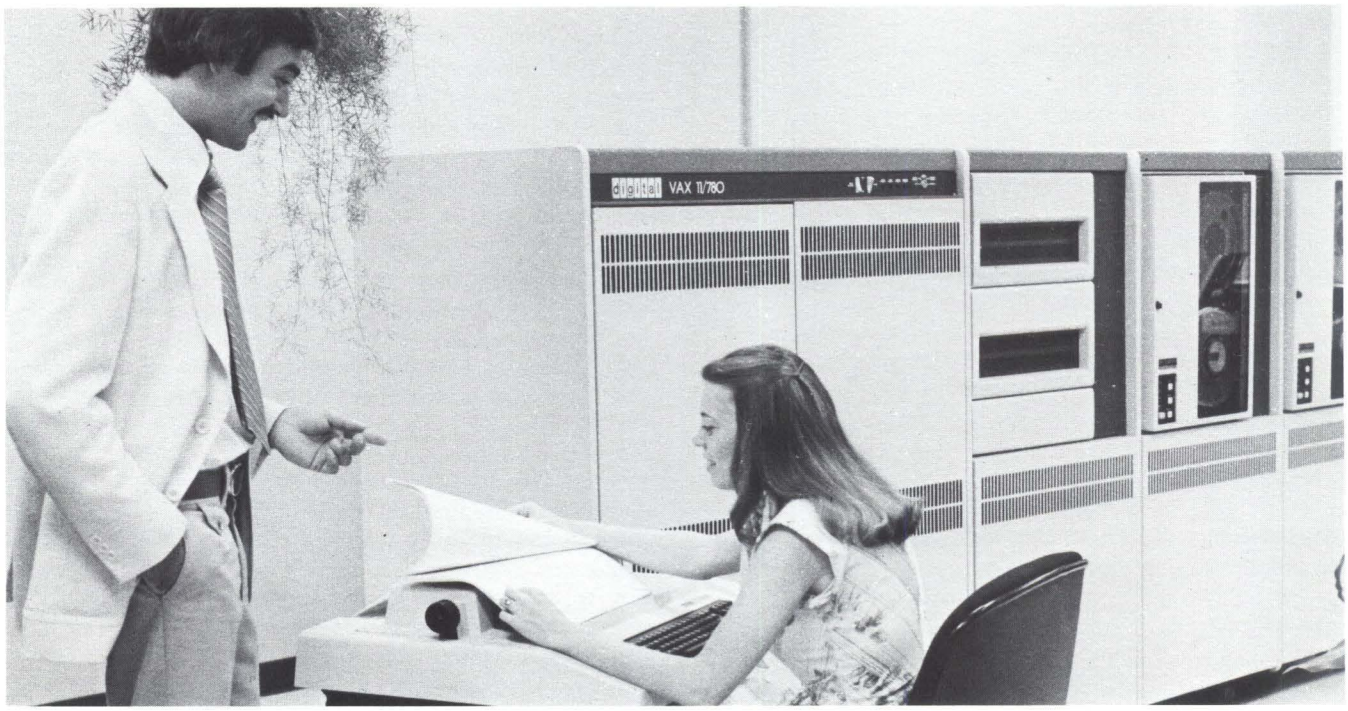
The VAX/VMS operating system provides a file and record management facility that allows the user to create, access, and maintain data files and records within the files with full protection. The record management services handle sequential, relative, and multi-key indexed file organizations, sequential and random record access, and fixed and variable-length records. Indexed files with sequential and random record access are available to compatibility mode programs, such as those written in PDP-11 BASIC-PLUS-2/VAX.

The VAX/VMS operating system supports the Files-11 On-Disk Structure Level 2 (ODS-2), which provides the facilities for file creation, extension, and deletion with owner-specified protections and multilevel directories. On-Disk Structure Level 2 is upwardly compatible with Level 1, the file system currently available under the PDP-11 IAS and RSX-11 operating systems. Both native and compatibility mode programs can access both Level 1 and Level 2 volume structures.

DECnet-VAX networking capabilities are available as an option for point-to-point interprocess communication, file access, and file transfer, and include down-line command file and RSX-11S system image loading. The Network Command Terminal facility allows users on one system to log into another VAX system on the network. The Mail facility allows electronic mail to be addressed to users on any VAX node in the network.

Peripherals

Medium capacity disks, unit record devices, terminals, the interprocessor communications links, and user-specific devices are UNIBUS peripherals. The UNIBUS adaptor(s)



provides the hardware pathways for data and control information to move between the UNIBUS and the memory interconnect.

High-performance MASSBUS mass storage peripherals are connected to the memory interconnect via a buffered MASSBUS adaptor. The MASSBUS adaptors provide the hardware pathways for data and control information to move rapidly between a MASSBUS peripheral controller and the memory interconnect.

PERFORMANCE

Many features of VAX ensure that the system provides real-time, computation, and transaction processing applications with the processing speed and throughput they need.

The processor provides 64-bit, 32-bit, 16-bit, and 8-bit arithmetic, instruction prefetch, and an address translation buffer.

An optional high-performance floating point accelerator (FPA) can be added to the VAX-11/780 system. The FPA is an independent processor executing in parallel with the base CPU. The FPA takes advantage of the CPU's instruction buffer to prefetch instructions and memory cache to access main memory. Once the CPU has the required data, the FPA overrides the normal execution flow of the standard floating point microcode and forces use of its own code. Then, while the FPA is executing, the CPU can be performing other operations in parallel. The FPA executes standard floating point instructions with substantial performance improvement. This execution is architecturally transparent to the programmer. In addition, the FPA enhances the performance of a number of additional instructions including:

- extended multiply and integerize (EMOD)
- polynomial evaluation (POLY), (F_floating and D_floating formats for both instructions)
- all floating to integer and integer to floating conversions
- 8- and 16-bit integer multiply (MULB and MULW)
- extended multiply (EMUL)
- 32-bit integer multiply (MULL)

The EMOD instruction is used for fast, accurate range reduction of mathematical function arguments. The POLY instruction is used extensively in the evaluation of mathematical functions such as sine and cosine (the VAX/VMS mathematics library makes use of the POLY instruction to significantly reduce the execution time of its subroutines). The MULL instruction is often used in matrix manipulation subscript calculations.

The VAX processor architecture is specifically designed to support high-level language programming. Because the instruction set is extremely bit efficient, compilation of high level language user programs is also very efficient. Among the features of the processor that serve to reduce program size and increase execution speed are the:

- variable length instruction format
- floating point, packed decimal, and character string data types
- indexed, short displacement, and program counter relative addressing modes
- small constant short literals

Furthermore, many instructions correspond directly to high-level language constructs:

- the Add Compare and Branch instruction for DO and FOR loop control
- the Case instruction for computed GO TO statements
- the 3-operand arithmetic instructions for statements such as "A = B+C"
- the Index instruction for computing index values, including subscript range checking
- the Edit instruction for output formatting

Much of the processor architecture also ensures that the operating system incurs minimal overhead for real-time multiprogramming. For example, the operating system uses:

- the context-switching instructions and queue instructions to schedule processes
- the asynchronous system trap (AST) delivery mechanism to speed returns from system service calls

Careful design, coding, and performance measurement ensure that the flow within the system is as rapid as possible.

RELIABILITY

Built-in reliability features for both hardware and software provide data integrity, increased up-time, and fast system recovery from power, hardware, or software failures. Several of the VAX reliability features are discussed in the following paragraphs.

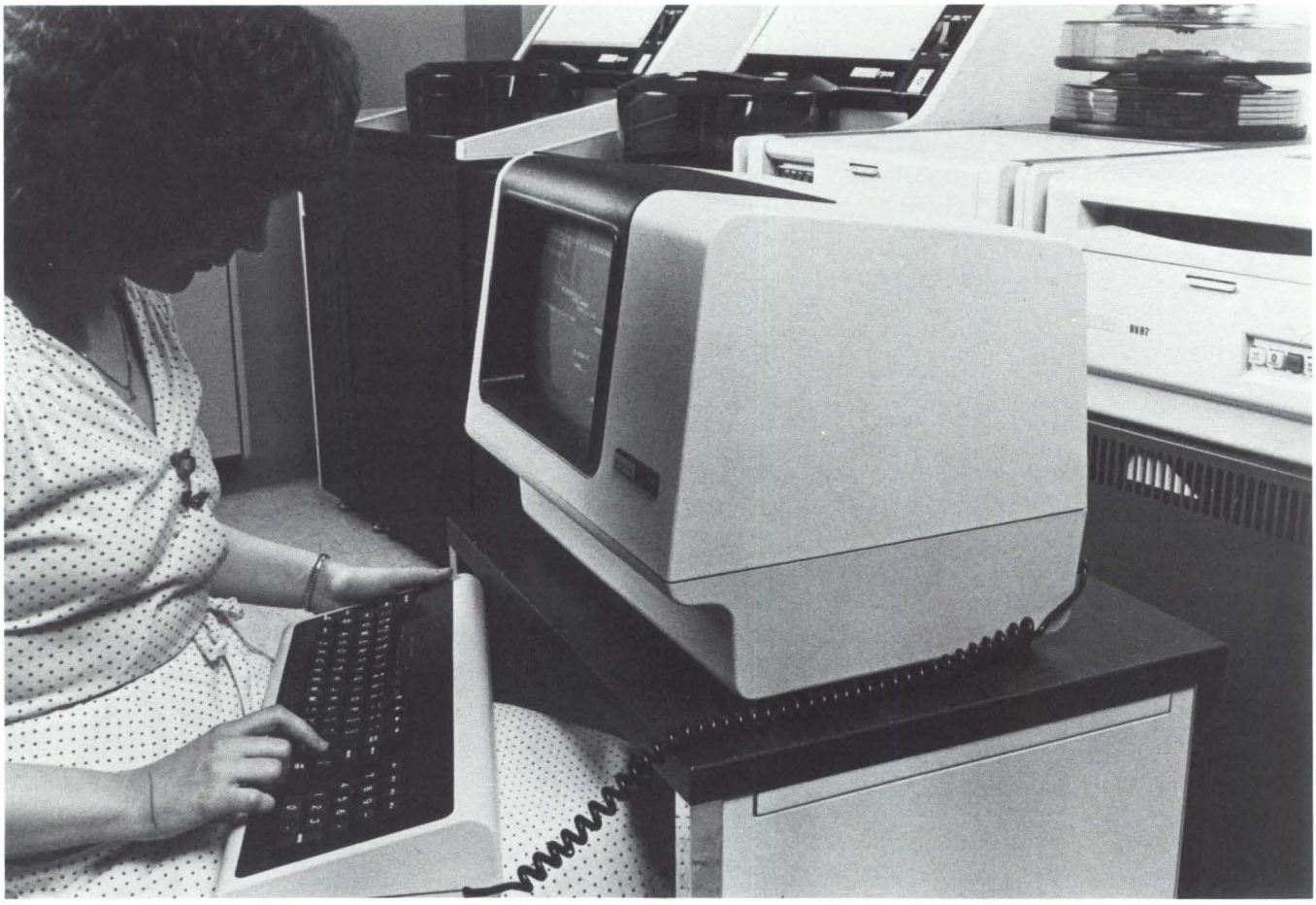
Data Integrity

VAX provides memory access protection both between and within processes. Each process has its own independent virtual address space which can be mapped to private pages or shared pages. A process cannot access any other process' private pages. The VAX/VMS operating system uses the four processor access modes to read and/or write protect individual pages within a process. Protection of shared pages of memory, files, and interprocess communication facilities such as mailboxes and event flags is based on file ownership and application group identification.

The VAX/VMS file system detects bad blocks dynamically and prevents re-use once the files to which they are allocated are deleted.

Integral fault detection hardware includes:

- memory error correcting code that detects all double-bit errors and corrects all single-bit errors
- disk error correcting code that detects all errors up to 11 bits and corrects errors in a single burst of 11 bits
- extensive parity checking
- peripheral write-verify checking that checks all input and output disk and tape operations and ensures data reliability
- track offset retry hardware that enables the operating system to recover from many disk transfer errors



System Availability

The VAX/VMS operating system allows the VAX system to continue running even though some of its hardware components have failed. The system automatically determines the presence of peripherals on the processor at bootstrap time. If the usual system device is unavailable, the system can be bootstrapped from any disk. If memory units are defective, memory is configured so that defective modules are not referenced. Software spooling allows output to be generated even if the normal output devices are not available. Also, devices can be added on line.

The system operator can perform software maintenance activities without bringing the system down for stand-alone use. The operator can perform disk backup for both full volume backup/restore and single file backup/restore concurrent with normal activities.

The VAX/VMS operating system supports on-line peripherals diagnostics. VAX/VMS performs on-line error logging of CPU errors, memory errors, peripheral errors, and software failures. The operator or field service engineer can examine and analyze the error log file while the system is in operation.

System Recovery

Automatic system restart facilities bring up the system without operator intervention after a system failure caused by a power interruption, a machine check hardware malfunction, or a fatal software error. The VAX/VMS operating system automatically performs machine checks and internal software consistency checks during system operation. If the checks fail, VAX/VMS performs a system dump and reboots the system if the operator has set the system for auto-restart.

Memory battery backup can be used to preserve the contents of memory during a power outage. A special memory configuration register indicates to the recovery software whether data in memory was lost. Following a power failure, the recovery software restarts all possible I/O in progress before the failure occurred. Programs can use the VAX/VMS power-fail asynchronous system trap facility to initiate user-specific power fail recovery processing. If memory battery backup is used, the time-of-year clock allows the recovery software to calculate elapsed time of the outage.

VAX remote diagnosis allows DIGITAL to run diagnostics, examine memory locations, and diagnose hardware/software problems from a remote diagnosis center. The engineer who goes to the site is prepared in advance to correct any problems that may have occurred.

FLEXIBILITY

VAX is a system that is easy to use because it is both flexible and easy to extend. Several of the ways in which VAX provides the user with flexible operating and programming environments are introduced below.

Flexibility in the Operating Environment

Virtual memory gives the user the ability to write and execute arbitrarily large programs without concern for addressing limitations. The paging and swapping algorithms allow more programs to execute than the available physical memory would allow if all programs had to be totally resident.

Both paging and swapping are transparent to the user, and therefore allow the system to be extended without reprogramming. The system's physical memory configuration can change without requiring program redesign or relinking. Programmers never have to structure their programs, although they can, at their option, to achieve maximum efficiency and performance for a given program. They can control working set size, lock pages in the working set or memory, and lock an entire working set in memory. In addition, the system manager can control the amount of time a process is guaranteed memory residency once it is swapped in.

The VAX/VMS scheduler recognizes 32 scheduling priorities. A program can modify its priority during execution. Real-time processes execute at one of the high 16 priority levels, and normal processes (including system processes) execute at one of the low 16 priority levels. The scheduler may temporarily increase the priority of a normal process to increase its response to I/O events or system events (but it can never lower the priority of real-time process).

Batch and printer output processing are completely flexible. The operator controls the number of batch jobs that can run concurrently. The operator defines the number of spooler queues. There can be multiple print queues: a generic queue for jobs that can be output on any printer, and several queues for jobs that are designated for a specific printer.

Batch jobs can be submitted to batch streams from the interactive environment using a terminal command, from another batch job, or by any program using a system call. Submitted batch jobs are queued, and a time can be specified after which a batch job should be executed.

Flexibility in Programming Interfaces

The I/O programming facilities can be as device-independent or device-specific as desired. The record management services support high-level programming languages by providing transparent record access and also enable the programmer to request specific record management services or system services to control file allocation, record blocking, or record accessing directly. Programmers can also use the system services to access file-structured devices or non-file structured devices if they wish to use their own record processing or volume structuring techniques.

Access to network facilities is device-independent, but a user who so desires can exert control over operations to obtain error reports or notification of broken connections (interrupt messages, inbound connections). System access protection applies to all network access.

Programmer Productivity

In addition to the system's reliability and performance features described above, VAX offers many tools to aid programmer productivity.

- Interactive editors with CAI startup—VAX/VMS supports several interactive and batch text editors, including SOS, SLP, and now the DIGITAL standard editor EDT. The system features a computer-aided instruction course to introduce the user to the power and flexibility of EDT.
- Interactive symbolic debugger—The interactive symbolic debugger provides a fast and efficient method by which the user can trace program errors. The debugger offers the user such features as; support of various native languages, support of many data types, and its interactive symbolic operation, i.e., the user can refer to program locations using those symbols created within the program.
- FMS interactive screen format generation—The Forms Management System contains an interactive editor which allows the application programmer to create and/or modify screen formats.
- DATATRIEVE—DATATRIEVE software provides fast and convenient access to data within a file or files. This query/report writing system provides the user with either video or hardcopy output.
- HELP facility—The HELP facility provides the user with on-line instructions pertaining to selected system operations.

Extending the System

The VAX/VMS command language can be extended with user-defined commands through the use of command procedures. A command procedure is a set of commands, data, or other command procedures processed in sequence. The user can invoke command procedures by a single command that can include parameters for the procedure, such as file names or values for symbols. Command procedures can execute programs, transfer control within the command procedure conditionally or unconditionally, request input from the user, and manipulate numeric and string symbols.

VAX/VMS uses a standard procedure call interface supported by the processor's call instructions. The calling program and called procedure can be written in different languages. This contributes to the writing of error-free, modular, and maintainable software that can be shared by many programs. The standard procedure call interface is particularly useful to systems programmers who want to add their own shareable libraries and library procedures to the VAX/VMS Common Run Time Procedure Library.

PDP-11 Compatibility

Users who already know the PDP-11 will find the native VAX-11 instruction set and programming characteristics easy to learn when developing new applications for the VAX system. The PDP-11 and the VAX systems have almost identical FORTRAN, BASIC, and COBOL languages. Users who have programmed in any of these languages on the PDP-11 will need to spend very little time learning the VAX system.

VAX offers many PDP-11 compatibility features:

- the VAX processor can execute a subset of PDP-11 16-bit instructions in compatibility mode
- the VAX/VMS operating system provides functionally equivalent system services for many RSX-11M execu-

tive directives

- the VAX/VMS high-level language compilers accept source languages that are upwardly compatible with the same PDP-11 compilers
- the VAX/VMS file system can read and write disk volumes and magnetic tapes written under RSX-11 and IAS operating systems
- the VAX/VMS record management services provide record processing methods that are upwardly compatible with RMS-11 record management services
- the VAX/VMS operating system provides an RSX-11 MCR command language interpreter
- the DECnet-VAX package supports RSX-11S system image down-line loading

These features make VAX an ideal host system to PDP-11 systems in a distributed processing environment.

3
The Users

The VAX system is designed to execute many different kinds of jobs concurrently. Jobs consist of one or more processes, each of which can be executing a program image that interacts with on-line users, controls peripheral equipment, and communicates with other jobs in the same system or in remote computer systems. Jobs include:

- customer-written interactive, batch, and real-time applications
- interactive and batch program development jobs
- system management and control jobs

Typically, VAX users interact with application or system jobs via an on-line terminal, or benefit from production batch or real-time jobs. To aid in the development of interactive, batch, and real-time applications, and manage and control system resources, VAX enables:

- *The application programmer* to write, compile, and test programs interactively or in batch mode, taking advantage of source code, object code, and program image libraries.
- *The system programmer* to design application systems that require a high degree of job and process interaction, data sharing, response time, and system and device independence.
- *The system manager* to authorize users, limit resource usage, and grant or restrict privileges individually.
- *The system operator* to monitor operations, service user requests, and control batch production.

Users can directly control the operation of VAX through the operating system's command language. In general, the command language is used by programmers to develop application software, by operators to monitor the system, and by system managers to assign user privileges.

Application programmers may also employ the command language to execute their application programs explicitly. The command language may be easily extended to provide custom-tailored commands defined by the user. Customer-written application programs can provide their own command interfaces for people using the system. Transaction processing applications may require several terminals to be slave terminals, meaning that they are tied to particular application programs that handle requests entered by the user.

The system manager can assign access user names and passwords to users who log on the system at a command terminal, and determine their privileges for obtaining services and limits for using resources. Users who access the system through an application terminal interface have the resources and privileges granted to the application programs run on their behalf. An application program itself determines who can request its services.

THE APPLICATION PROGRAMMER

The application programmer has four basic tools for requesting services of the system:

- command interpreter
- programming languages
- programmed file and record management services
- programmed system services

The application programmer gains access to the system through the command interpreter. The command interpreter enables the programmer to create, compile, and execute programs written in any of several programming languages. The record management services are available through any programming language to provide device-independent data processing. The system services, although primarily of interest to systems programmers, are also available to the application programmer for requesting special services of the operating system.

Command Language

The command interpreter is interactive, comprehensive, easy to use, and extremely flexible. It enables the user to log onto the system, manipulate files, develop and test programs, and obtain system information. Furthermore, it enables users to extend or redefine their command repertoire as well as write command procedures easily. The command language includes:

- a set of English commands that provide the basic command repertoire
- a set of control characters that provide special functions such as erase command line, interrupt the program currently executing, etc.
- a set of special operators and commands that can be used to automate command streams and extend the command repertoire

Table 3-1 lists the basic set of English commands accepted by the command interpreter. The command interpreter is easy to use because its commands can be abbreviated, because it prompts for necessary arguments, and because it assumes standard or user-selected default values for command parameters and qualifiers.

A command line normally consists of a command verb followed by one or more parameters that identify the object of the operation (a file, for example) or qualify how the operation is to be performed. If the interactive user enters an incomplete command, the command interpreter prompts for any necessary parameters. For example, the COPY command, which creates a copy of an existing file, accepts a total of two file specifications: one for the file to be created and one for the file to be copied. The file specifications identify the exact location and name of the files. The COPY command can be entered in any of several ways:

```
$ copy
$_FROM: file-name-1
$_TO: file-name-2

$ copy file-name-1
$_TO: file-name-2

$ copy file-name-1 file-name-2
```

The command interpreter displays the dollar sign to prompt for a command, and the dollar sign underscore to prompt for a missing parameter.

Command Procedures

To eliminate the need for typing frequently repeated sequences of commands, users can create **command procedures**. A command procedure is a file containing complete command lines (including the $ prompt character). The user can request the command interpreter to read and process the command lines in a command procedure file just as if they were being typed successively at the terminal. To execute a command procedure, the user simply precedes the name of the command procedure file with an "at" sign (@):

```
$ @procedure-file
```

Figure 3-1 is a simple example of how the user might create, interactively, a new command called EXECUTE. The user has previously written a VAX-11 PASCAL program named AVE, and now wishes to compile, link, and execute this program. To the user, EXECUTE appears to be a command like any other command in the DCL command repertoire. The first step is to create the command procedure file EXECUTE.COM. Following this, the user enters the PASCAL, LINK, and RUN DCL commands as input to the command file.

The blue dollar signs in Figure 3-1 represent DCL system prompts, the remainder of the text is user supplied.

```
$ CREATE EXECUTE.COM
$ PASCAL AVE
$ LINK AVE
$ RUN AVE
↑Z (CONTROL Z)
$
```

Figure 3-1
A Simple Command Procedure

Now, to execute the command procedure file, EXECUTE.COM, the user can type:

```
$ @EXECUTE
```

or the user can create a symbol to execute the command file. The user may equate a unique series of letters to the command procedure file. For instance:

```
$ EXE := @EXECUTE
```

Now, to execute the command procedure file, EXECUTE.COM, the user need only type the symbol EXE:

```
$ EXE
```

The user could just as easily have created a symbol called GO to execute the command procedure file EXECUTE.COM. In this instance:

```
$ GO:=@EXECUTE
```

Now, to execute the command file, EXECUTE.COM, the user can enter GO in response to the DCL prompt ($):

```
$ GO
```

Table 3-1
DCL Command Language Summary

GENERAL SESSION INFORMATION AND CONTROL

LOGIN	The user initiates an interactive session with the system by typing CTRL/C, CTRL/Y or by pressing the carriage return on a terminal not currently in use. The system then prompts for username and password, and validates them.
HELP	Displays information to assist the user in selecting the proper command qualifiers.
SHOW	Displays any of the following information: current day, date, and time; current default device and directory name; status of devices in the system; logical device name assignments; current characteristics and status of specified mag tape device; name, number, and status of local network node and lists available remote nodes; default characteristics of system printer; status of current process; current file protection to be applied to all new files created during terminal session or batch job; current status of entries in printer/batch job queues; current disk quota; current VAX-11 RMS default multiblock and multibuffer counts; status of currently executing image in process; current value of a local or global symbol; displays a list of processes and status information in system; current characteristics of a specified terminal; logical name translations; display of working set quota and limit assigned to current process.
SET	Defines default translation mode for cards read into system card reader; controls whether command interpreter receives control when CTRL/Y is pressed; changes the user's default device name or directory name; defines default characteristics for specific magtape device; determines whether command interpreter performs error checking following execution of commands in command procedures; changes execution characteristics of currently executing process; changes a file's protection; changes current status or attributes of a file queued for printing or for batch job execution; defines default values for multiblock and multibuffer counts used by VAX-11 RMS; changes characteristics of a specified terminal; controls whether or not command lines in command procedures are displayed at terminal or printed in batch job log; redefines default working set size for the current process.
ASSIGN	Assigns a logical name to a given character string (equivalence name) and stores the the pair of names in a process, group, or system logical name table. Generally used to create a logical name for a device.
DEFINE	Creates a logical name equivalence. (Same as ASSIGN except for syntax.)
DEASSIGN	Breaks the correspondence between a logical name and its equivalence name (see ASSIGN and ALLOCATE), or deletes a symbol (see DEFINE).
MCR	Signifies that the given command or following command lines are to be interpreted by the RSX-11 command interpreter.
LOGOUT	Terminates an interactive session and releases all resources allocated to the user.
REQUEST	Displays a message at a system operator's terminal and optionally requests a reply.

BATCH AND COMMAND PROCEDURE SPECIFIC CONTROL*

SUBMIT	Places a given batch command file or command procedure in a batch queue for processing.
$PASSWORD	Specifies the password associated with the user name specified on a JOB card for a batch job.
$JOB	Indicates the beginning of a batch command file and provides job control information (such as time limit).
$INQUIRE	Requests interactive assignment of a value to a symbol and assigns the symbol a name.
$GOTO	Transfers flow of control to a given labeled line.
$ON	Transfers flow of control to a given labeled line if an error of a given severity or greater is encountered at any time during command procedure processing.
$IF	Transfers flow of control to a given labeled line if the result of a logical comparison of symbolic values is true.
$EOD	Signifies the end of data in the input stream following a $DATA command.
$EXIT	Terminates the command procedure.
$EOJ	Marks the end of a batch job submitted through the system card reader. EOJ performs the same functions as the LOGOUT command.
DECK	Marks the beginning of an input stream for a command or program.

*Command names preceded by a $ are meaningful only in a batch command file or command procedure. All other commands listed in this table can either be issued interactively or used in a batch command file or command procedure.

VOLUME AND DEVICE RESOURCE CONTROL

MOUNT	Requests the operator to make a volume available to the user and optionally associates a logical name with the volume or volume set.
INITIALIZE	Writes a directory file and other volume structuring information on a disk or magnetic tape volume to prepare it for use.
DISMOUNT	Requests the operator to break the logical association of this device with the user's job.
ALLOCATE	Obtains exclusive ownership of device and enables the user to assign a logical name to the device.
DEALLOCATE	Releases allocated devices.

FILE MANIPULATION

DIRECTORY	Reports information (size, protection, ownership, creation time, etc.) on a given file or set of files.
CREATE	Creates a new file from data subsequently entered in the input stream (user at terminal or batch stream). Creates a directory file on a volume.
EDIT	Opens a text file and accepts commands to insert, delete or modify data in the file.
DELETE	Deletes one or more files from a mass storage disk volume.
DELETE/ ENTRY	Deletes one or more entries from a printer or batch job queue.
DELETE/ SYMBOL	Deletes one or more symbol definition from a local symbol table or from the global symbol table.

Table 3-1 (con't)
DCL Command Language Summary

PURGE	Deletes all but the latest version of a given file or files, optionally keeping the latest two or more versions.
RENAME	Changes the name of one or more existing files.
COPY	Copies the contents of a file or files, creating another file or files.
APPEND	Concatenates the contents of sequential files to a given output file, or creates a new output file from the concatenated contents of given sequential files.
DIFFERENCES	Compares two files and reports the differences between the two.
SORT	Creates a file by rearranging the records in a given file based on the contents of key fields within the records.
OPEN	Opens a file for reading or writing at the command level.
CLOSE	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.
READ	Reads a single record from a specified input file and equates the record to a specified symbol name.
WRITE	Writes a record to a specified output file.
PRINT	Sends the contents of a given file or files to a spooled output device such as a line printer.
TYPE	Displays the contents of a given file or files on the device identified by the logical name SYS$OUTPUT: (default generally the user's terminal).
DUMP	Produces a printed listing of the contents of a file, ignoring any print formatting characters that may appear in the records.
UNLOCK	Permits access to a file that was improperly closed.
ANALYZE/ OBJECT	Provides a description of the contents of an object file or an executable image file.
PROGRAM DEVELOPMENT AND EXECUTION CONTROL	
MACRO	Assembles given assembly language source modules, producing an object module.
FORTRAN	Invokes the VAX-11 FORTRAN compiler to compile one or more source programs.
COBOL	Compiles given COBOL language source modules using VAX-11 COBOL compiler, producing an object module.
BASIC	Compiles given BASIC language source modules, producing an object module.
BLISS	Invokes the VAX-11 BLISS-32 compiler.
PL/I	Invokes the VAX-11 PL/I compiler to compile one or more source programs.
PASCAL	Invokes the VAX-11 PASCAL compiler to compile one or more source programs.
CORAL	Invokes the VAX-11 CORAL 66 compiler to compile one or more CORAL source programs.
LIBRARY	Creates, deletes, or maintains libraries of object modules, shareable images, or macro source modules.
LINK	Links modules to produce images.
RUN	Executes a program image in the current process context, or creates a detached process and executes a program image in that process context.
DEBUG	Starts interactive debugging session after interrupting program image execution by typing a Control C or Control Y.
EXAMINE	Displays the contents of a location in virtual memory.
DEPOSIT	Replaces the contents of a location in virtual memory with the given data.
CONTINUE	Resumes execution of a program interrupted by typing a Control C or Control Y.
STOP	Terminates the program currently interrupted by a Control C or Control Y.
SUBMIT	Enters a command procedure in the batch job queue.
SYNCHRONIZE	Places the process executing a command procedure in a wait state until a specified batch job completes execution.
WAIT	Places the current process in a wait state until a specified period of time has elapsed.
CANCEL	Cancels scheduled wakeup requests for a specified process. This includes wakeups scheduled with the run command and with the schedule wakeup ($SCHDWK) system service.

In addition to executing command procedures at a terminal, an interactive user can also submit batch jobs. Batch jobs execute under control of the system operator and leave the user's terminal free to continue interactive or command procedure processing. A batch job can be submitted as a deck of cards or as a batch command file. A batch command file is identical to a command procedure file, except that a batch command file submitted as a deck of cards begins with a $JOB card that provides job control information.

However, DCL is more than just a string of commands capable of standing alone; it possesses true high level programming language statements such as GOTO, IF, etc., and accepts a series of up to eight user-defined parameters 'P1' through 'P8'. DCL can be used to completely define and control a user environment tailored to a specific application.

The power and flexibility of command procedures and the DCL command language will be treated in greater detail in the Program Development and Support Facilities section.

RUN Command

The RUN command includes several qualifiers (DELAY, INTERVAL, and SCHEDULE) which are of particular importance to the real-time programmer.

Specifying any of the above qualifiers places a process in

hibernation, a wait state in which the process can be reactivated only when a particular time value occurs. The time value can be specified in delta time (/DELAY qualifier), in absolute time (/SCHEDULE qualifier) or at recurrent intervals (/INTERVAL qualifier). When the image completes execution, the process returns to a state of hibernation.

Programming Languages

The system includes the VAX-11 MACRO assembler for programming the machine using its native instruction set. A wide variety of language processors are optionally available to high-level language programmers: VAX-11 FORTRAN, VAX-11 COBOL, VAX-11 BASIC, VAX-11 PL/I, VAX-11 PASCAL, VAX-11 CORAL 66, and VAX-11 BLISS-32. In addition, VAX/VMS supports several optional language compilers that execute in compatibility mode. These include PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX and MACRO-11. These language processors, introduced below, are described fully in the Languages section.

The VAX-11 FORTRAN language processor is based on the American National Standard FORTRAN specification X3.9-1977 (commonly referred to as FORTRAN-77). The VAX-11 FORTRAN compiler supports this standard at the full language level. Additionally, however, VAX/VMS provides support for the industry-standard FORTRAN features based on FORTRAN-66 (an option that can be selected at compile time), the previous ANSI standard.

The VAX-11 FORTRAN compiler produces shareable, highly optimized VAX-11 native object code. The compiler takes advantage of the system's large virtual address space while utilizing the floating point and character string instructions. FORTRAN I/O processing is supported by the record management services (VAX-11 RMS). VAX-11 FORTRAN object modules can be linked with assembler-produced object modules and the system's run-time library, which is common to all native mode programs, to provide standard library functions. The VAX-11 FORTRAN language processor offers the programmer such features as:

- Full ANSI-77 FORTRAN language support
- Access to ISAM files as well as relative and sequential files
- Access to the VAX/VMS system services and the run time library procedures
- FORTRAN program can call external routines written in other VAX-supported high level languages
- The compiler itself is shareable

The VAX-11 COBOL language processor produces highly efficient shareable native mode code which utilizes the system's packed decimal and character instruction set and extended call facility. The VAX-11 COBOL language is based on the American National Standard Programming Language COBOL, X3.23-1974, the industry wide accepted standard for COBOL. Many features of the planned COBOL standard (anticipated in 1981) are also included. The VAX-11 COBOL language processor offers the programmer such features as:

- the ability to manipulate data strings via the INSPECT verb
- performing sorting and merging operations at the COBOL source language level
- complete file organization capability including sequential, relative, and indexed I/O
- structured programming
- support for the full range of data types including packed decimal and floating point
- COBOL programs can call external routines written in COBOL or other VAX-supported high level languages
- capability of writing shareable code for use by other native mode high level languages
- accepts source programs coded in either ANSI standard format or the shorter easy to read DIGITAL terminal format

VAX-11 BASIC is a native mode, shareable language processing system producing shareable VAX native object code. The language compiler utilizes VAX floating point and character string instructions while supporting a fast RUN command and immediate mode execution which makes it well suited for interactive use. VAX-11 BASIC is a superset of PDP-11 BASIC-PLUS-2, offering the VAX user major enhancements such as:

- access to VAX-11 RMS file and record processing
- long variable names (up to 31 alphanumeric characters)
- dynamic string handling
- CALL statement providing interface to common language environment
- shareable and re-entrant code

VAX-11 PL/I is an extended implementation of the proposed ANSI X3.74, American National Standard PL/I General Purpose Subset to full PL/I (ANSI X3.53-1976). VAX-11 PL/I extensions include some full language features and VAX/VMS system-specific extensions.

PL/I is a versatile language that is suited to commercial, scientific, and systems programming applications. Some of the features of VAX-11 PL/I include:

- block structured language
- full support for all VAX-11 hardware data types
- powerful I/O capabilities including ISAM support
- user control of storage allocation
- condition handling
- standard VAX-11 CALL interface, including access to VAX/VMS System Services and the run-time library
- fast, native-mode optimizing compiler
- shareable, position-independent code

VAX-11 PASCAL, a re-entrant native mode compiler, is an extended implementation of the PASCAL language as defined in the PASCAL User Manual and Report (Jensen and Wirth, 1974). Particularly suited to instructional use, PASCAL is gaining increasing popularity as a general purpose language. Major features of the VAX-11 PASCAL language include:

- block structuring via the BEGIN...END compound statement to allow easy logic flow
- data structuring including the ability to declare and use pointers, records, files and arrays

- predefined procedures and functions to deal with I/O handling and data manipulation

VAX-11 PASCAL takes advantage of the VAX hardware floating point and character instructions as well as the virtual memory capabilities of the VAX/VMS operating system. Many of the features common to other native languages 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

The VAX-11 CORAL 66 compiler executes 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. VAX-11 CORAL 66 is essentially a high level block-structured language whose compiler offers the user many features including:

- several numeric types (byte, long and double)
- generation of re-entrant code at the procedure level
- code optimization
- English text error messages
- INCLUDE keyword to incorporate CORAL 66 source code from user-defined files

VAX-11 BLISS-32 is a high-level systems implementation language for VAX-11, which runs in native mode under VAX/VMS. The BLISS language is specifically designed for building language compilers, real-time processors, utilities, and operating system software. BLISS contains many of the features of high-level languages (e.g., DO loops, IF-THEN-ELSE statements, automatic stack, and mechanisms for defining and calling routines), but it also provides the flexibility and access to hardware that one would expect from an assembly language. VAX-11 BLISS-32 can be used as an alternative to assembly language coding in all except the most machine-dependent systems programming applications. The VAX-11 BLISS-32 language processor offers the programmer such features as:

- program execution on architecturally different machines with little or no modification
- construction of complex expressions in which several different kinds of operations can be performed in a single program statement
- exploitation of high level language constructs

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

PDP-11 FORTRAN IV/VAX to RSX is an extended FORTRAN IV processor based on ANSI FORTRAN X3.9-1966. It supports mixed mode arithmetic, extended I/O facilities for data formatting, error condition transfer statements, bit manipulation, library usage, and several debugging facilities. The FORTRAN IV compiler (and its run time system) execute in the compatibility mode environment.

MACRO-11, the PDP-11 assembly language, is included in the compatibility mode environment. Programs written in MACRO-11 can be assembled to produce relocatable object modules and optional assembly listings.

Record Management Services

The record management services (RMS) are a collection of procedures that extend the programming languages by providing general purpose file and record handling capabilities. Programmers using RMS include in their programs statements that read, write, find, delete, and update records within files. Records can be fixed or variable length.

RMS enables the programmer to choose the file organization and record access method appropriate for the data processing application. The file organizations and record access methods are independent of the language in which they are programmed, although some languages support file organizations and access methods not provided in others. Every programming language uses RMS to process files organized to provide sequential, random or multi-keyed indexed record accessing.

For further information on RMS and the system's data management techniques, refer to the section on Data Management Facilities.

THE SYSTEM PROGRAMMER

The system programmer can use this system to design and build application systems for multiprogramming environments requiring fast response and a high degree of job interaction and data sharing.

Job and Process Structure

The user program environment consists of a job structure that can contain many processes. A process is the schedulable entity capable of performing computations in parallel with other processes. It consists of an address space and an execution state that define the context in which a program image executes. An executing program is associated with at least one process, but it can be associated with several processes.

A multiple process job structure allows one job to execute more than one program image at the same time. One process can wait for an event (such as I/O completion) to occur while another process continues its computations. The processes can communicate in several ways. They can coordinate their execution synchronously using event flags or asynchronously using software-simulated interrupts. They can send messages back and forth using virtual record-oriented devices called "mailboxes," and they can share code and data on disk and in memory.

Jobs can be grouped into application subsystems that share code and data protected from other applications. The processes within jobs in the same group can coordinate their activities using group interprocess communication facilities such as mailboxes and event flags, as well as those facilities local to the job. They can access files and data in memory that are protected from other groups in the system.

Multiprogramming Environment

The system supports multiprogrammed applications that require high performance by providing:

- event driven priority scheduling
- rapid process context switching
- minimum system service call overhead
- processor access mode memory protection
- memory management control

The system schedules processes for execution based on the occurrence of events such as I/O completion as well as time quantum expiration. When scheduled, the context switching and interrupt processing hardware and software ensure that processes are activated quickly. Real-time processes can be assigned high priorities to ensure that they receive processor time on demand. A process can schedule its execution at a given time of day or after an interval has elapsed, and an appropriately privileged process can modify its priority during execution.

The system's memory management hardware and software ensure that paging, swapping, and dynamic memory allocation are both efficient and transparent to the programmer. Where real-time applications require performance control, both paging and swapping can be reduced or eliminated by increasing the amount of memory allocated to a process and by locking a process in memory. Because memory management is transparent, programs can be written and later tuned for performance after they are tested. The system provides a utility program to aid system programmers in evaluating the effectiveness of the memory management system for their processes.

Program Development

This system provides the system programmer with tools that support highly modular program and applications development. By taking advantage of these tools, the programmer can build applications quickly, and easily modify and extend them later.

The system includes editors, compilers, librarians, linkers, and debuggers for both the native and compatibility programming environment. All program development utilities can be used either interactively or in batch mode, including the editors and debuggers. The native symbolic debugger recognizes a command language similar to the operating system command language and uses expressions similar to the language in which the program being debugged was written.

Executable program images can be built using extensive libraries. In the native programming environment, the programmer can create libraries of assembler macro definitions, of object modules, and of image modules. The system also includes the common run time procedure library, which provides library functions common to all native programming languages.

All program interfaces to the operating system and its utilities have uniform calling standards. System programmers can add new library procedures to the common run time procedure library and install them on-line without modifying existing programs and utilities, since all arguments are passed using standard data structures.

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

THE SYSTEM MANAGER

A job is normally associated with a user known to the system to have certain privileges, quotas, and resources. The system manager authorizes users, plans data access and protection, grants privileges, controls resource utilization, and analyzes the system's accounting and performance information.

User Authorization

The system manager controls use of the system primarily by creating user authorization information. This information is recorded in a specially maintained and protected file called the **user authorization file**. The system manager can create, examine, and update this file at any time.

The file contains one entry for each user authorized to access the system. Each entry:

- identifies the user
- supplies defaults
- specifies privileges
- limits resource usage

User identification consists of a unique user name, a password, a default account name, and a user identification code (UIC). When logging onto the system, people must always enter their user name and password to gain access to the system. The password is not displayed on the user terminal. Privileged users can change the passwords they are assigned as often as they desire.

This system's data protection scheme is based on the user identification code (UIC) that the system manager assigns. A UIC controls each user's access to the data structures protected by UICs, which include both files and the interprocess communication facilities such as mailboxes, shared areas of memory, and event flags.

A UIC consists of a group number and a member number. Every user is assigned a UIC, and every data structure is assigned both a UIC and a protection code. A protection code identifies what types of access are available to which users. There are four types of access (read, write, execute, and delete), and there are four types of user (owner, group, world, and system). The owner is any user that has the same UIC as that assigned the data, the group is any user that has the same group number as that assigned the data, the world is any user, and the system is any user with a group number of 1 through 10.

Using this protection scheme, a system can have files and interprocess communication facilities that are available for access only by users having the same UIC, or for access only by users in the same group, or for universal access. Furthermore, since each data structure has its own protection code, it is possible to protect each data structure assigned the same UIC on a different basis. The system UICs are generally reserved for system users and system pro-

grams and data structures. This arrangement enables a user to protect a file from access by anyone other than the owner or group, but still enables the system to access the file for operations such as backup.

In addition to identifying the user and the set of data structures the user can access, the user authorization file supplies the user with a default file protection, a default directory name, and a default device name. When the user creates a file, the system assigns the default file protection unless requested otherwise. An owner can modify a file's protection at any time.

Directory names are arbitrary character strings identifying a directory file. A directory is simply a file containing a list of file names and other identification information that is used to find files on a volume. The default directory name identifies the directory that lists the files the user normally accesses. The default device name is the name of the device on which the volume containing the files the user normally accesses is mounted.

When the user issues a command to the command interpreter that operates on a file, or runs a program that opens a file, the file system uses the default directory name and default device name to locate the file unless specifically requested to use some other directory name or some other device name. The user can change the default directory and device names for a given session.

For further information on directories and directory structures, refer to the section on Data Management Facilities.

Privileges

Each user's authorization file entry contains a list of the privileges that the user can invoke. They include interprocess communication and control privileges, performance control privileges, file and device access privileges, and system operational control privileges. The system manager can grant distinct privileges individually to each user. Table 3-2 lists some of the privileges.

Privileges are checked when the user executes program images. If a user runs an image that attempts to execute a function requiring a privilege the user is not granted, the image incurs a privilege violation. For example, diagnostic programs require the privilege to issue device level diagnostic functions and the privilege to send messages to the error logger. Users not granted these privileges will receive privilege violations if they attempt to run diagnostics.

In certain cases, however, it is desirable to let a user run an image that requires privileges the user is not granted. For example, the login program image requires the privilege to switch to a more protected processor access mode to set the user's initial context in a protected area of memory. To let a user run an image that requires special privileges, the system enables the system manager to install **known images**. When the user runs a known image, the user obtains the necessary privileges to execute the functions required by the image, but only for the duration of that image's execution.

Resource Quotas and Limits

The user authorization file also provides the limits on how many system resources a user can tie up while logged on the system, and quotas for how much of a resource a user can use up during an accounting period. The system manager can assign user quotas for the maximum amount of CPU time accumulated during a given accounting period, and can limit the amount of dynamic system memory a job can utilize for buffers. The system manager can set disk usage quotas via the disk quota utility on a per user, per volume or volume set basis. VAX/VMS will automatically record usage and enforce the assigned quotas during file operation. However, each user possessing a private volume controls the disk quotas on that volume. The limits

Table 3-2
Privileges Summary

INTERPROCESS CONTROL
- create event flag clusters
- create permanent common event flag clusters
- create temporary mailboxes
- create permanent mailboxes
- create global sections
- suspend, resume, wake, and delete processes within the same group
- suspend, resume, wake, and delete any process
- create detached processes
- create and delete shared memory sections
- map to physical pages

ACCESS TO FILES AND DEVICES
- insert logical names in group logical name table
- insert logical names in system logical name table
- allocate spooled devices
- obtain exclusive ownership of a shared device
- override volume protection
- issue mount requests

PERFORMANCE CONTROL
- execute time critical images
- lock process in memory

SYSTEM OPERATION CONTROL
- issue operator commands
- set any privilege bits
- set process priority

PROGRAM EXECUTION
- execute Change Mode to Executive system service
- execute Change Mode to Kernel system service
- bypass file protection
- issue diagnostic functions
- send messages to error logger
- suppress accounting messages
- issue logical and physical I/O functions

imposed by VAX/VMS include the maximum number of:

- outstanding open files
- CPU time
- outstanding subprocesses created
- pages in a process working set
- pages in system paging files
- outstanding entries in the timer queue
- outstanding system buffered I/O requests
- bytes in system buffered I/O request
- outstanding direct I/O requests

Resource Accounting Statistics

The system maintains an accounting information file for collecting cumulative resource usage statistics. The system updates the accounting information file with detail records each time a process terminates. The detail statistics include:

- elapsed CPU time
- login (connect) time
- number of volumes mounted
- number of pages printed
- largest process virtual size
- largest process working set size
- number of page faults
- number of system buffered I/O requests
- number of direct I/O requests

A detail record identifies the account name, user name, and user identification code (UIC) to which the statistic applies. The accounting information file can be used to calculate billing information and reporting by account name, user name, or UIC. Because the system collects all detail records, system managers can define their own algorithms for resource usage billing.

Performance Analysis Statistics

The system collects statistics on its activities to help system programmers and managers tune the system for maximum performance. The information collected includes:

- System and Job Statistics—indicate the current number of processes, interactive users, and batch jobs in the system, the date and time at which the system was booted, and the current date and time.
- Processor Access Mode Usage—indicates how much time is spent executing at each of the access modes as a measure of the type of code being executed and the computational workload.
- Page Fault Activity—indicates how many and what kind of page faults occurred as a measure of the effectiveness of memory management.
- I/O Activity—indicates how many and what kind of I/O operations are taking place.
- Network Activity—indicates network workload (current number of nodes in the network, number of bytes transmitted and received, number of messages transmitted and received, number of buffers currently in use, number of successful and failing attempts to obtain space for network buffers).
- Response Time Histograms—indicate the time it takes the system to initiate user requests.

Display Utility Program

The Display Utility Program (DISPLAY) provides a dynamic display of system performance measurement statistics on a VT100 or VT52 video display terminal. By typing appropriate commands, system users may list information regarding I/O system activity, paging, CPU usage, current process activity, and other relevant statistics. Figure 3-2 shows a typical screen display.

THE SYSTEM OPERATOR

An operator is any user given the privileges by the system manager to perform operator functions. A system does not require an operator, but it can have one or several operators, and they can use any terminal to issue commands or run programs. Operator functions include:

- system startup and shutdown
- job control (change process priorities, kill jobs, etc.)
- device allocation
- volume mount and dismount request servicing
- on-line disk and magnetic tape volume and file backup
- spool and batch queue control
- software maintenance update installation
- diagnostic execution

An operator uses the command language to control operations, check system status, and run utility programs.

A special system program, the Operator Communications Manager (OPCOM), is the primary operator aid. It collects and delivers the messages all users and user programs send to the operators. Any operator can respond to user requests, and the Operations Communications Manager will remind operators of any outstanding requests.

Spooling and Queue Control

The operators define the number and kind of input and output spool queues in the system. The operators can create output spool queues for any number of devices, including line printers, terminals, or even magnetic tape. The operators can also create input queues for spooling batch input from a card reader.

The operators can assign each queue a priority, merge or redirect queues to other devices, and modify the queue set-up at any time. It is possible to have more than one print queue for the same printer. For example, an operator can create a generic printer queue that will collect jobs that can be printed on any of a set of printers, and at the same time have a print queue for each individual printer. A user can issue a print request for a generic printer or a particular printer, and the operator can override the user's request.

A print job can contain one or more files to be printed together. Print jobs can be submitted by an interactive user, batch job, or any program. Print jobs are also automatically submitted at the end of a batch job. A print job can specify the forms type required, the number of copies of the job, the job priority, and a "hold until given time" request. Each file within a print job has its own copies count,

FREE LIST: 2016 I/O SYSTEM RATES MODIFY LIST: 35
16:17:09

NAME	VALUE	RATE /SEC	AVG RATE	NAME	VALUE	RATE /SEC	AVG RATE
DIRECT I/Os	32	7.30	1.50	PAGE FAULTS	65	14.84	1.83
BUFFERED I/Os	29	6.62	3.24	PAGES READ	4	0.91	0.11
MAILBOX WRITES	0	0.00	0.00	READ I/Os	2	0.45	0.07
WINDOW TURNS	3	0.68	0.14	PAGES WRITTEN	0	0.00	0.00
LOGNAME TRANS	39	8.90	0.98	WRITE I/Os	0	0.00	0.00
FILE OPENS	3	0.68	0.07	TOTAL INSWAPS	0	0.00	0.00

Figure 3-2

Display Utility Program (I/O System Rates)

and each can have these options: double space, inhibit form feed, print a flag page, label each page, or delete after printing. The operator can choose whether or not to print burst (job separator) pages, and can put jobs on indefinite hold, modify the priority of a job, or abort a job.

Batch Processing

The system supports multiple stream, multiple queue batch processing. The operators control how many batch job streams can run concurrently. Batch jobs can be submitted by an interactive user, another batch job, or any program. When the number of batch jobs submitted exceeds the number of streams, the remainder of the batch jobs are held in a batch input queue. As with the spool queues, the operators can control the batch job queue. They can change job priority, hold a job until after a given time, hold a job indefinitely, and kill a job.

Volume mount commands issued in a batch job can request a generic device, such as any disk, or a specific device, such as disk drive unit 2. The batch job waits until an operator satisfies the mount request, while other batch jobs proceed. Operators can find out which job has a given device.

On-line Software Maintenance

An operator can incorporate maintenance updates to the software without bringing down the system for stand-alone use. For example, VAX-11/780 maintenance patches are distributed on floppy disk and the operator simply loads the console floppy disk drive with the maintenance floppy to update the software on disk. Depending on the nature of the update, an operator may have to restart the system to activate the patched modules.

System Recovery

An operator can select manual or automatic system recovery following a power interruption or hardware or software failure.

On automatic system recovery after power interruption, the system determines whether the contents of memory are still valid, and if so, restarts all possible I/O in progress at the time of the power interruption and continues operations from the point of interruption. If the contents of memory are not valid, either because memory battery backup is not included in the configuration, or because the power failure lasted longer than the battery, the system automatically boots itself from disk and executes the start-up command procedures.

Error Logging and Reporting

The error logger is a job that runs continuously. It collects errors detected by both hardware and software, including:

- device errors
- interrupt timeouts
- interrupts received from nonexistent devices
- memory, translation buffer, and cache parity errors

In addition, system software sends complete recovery information to the error logger following a power interruption or hardware or software failure.

The error logger writes all messages it receives into an error log file, noting vital system statistics at the time of the message. The error logger also notes benign events when they occur, such as when volumes are mounted and dismounted, and periodic time stamps indicating that no entries have occurred for a specified period of time. The error logger can accept messages from the operators at any time, and from any programs privileged to send messages to the error logger.

The system includes an error report generating program that converts the information in the log file into a text file that can be printed for later study.

On-line Diagnostics

An operator can run diagnostics to check the operation of both hardware and software. An operator can run system exercisers and device verification diagnostics while normal operations proceed. System exercisers test general purpose software and compare the results with known answers, reporting any discrepancies to the error logger.

Operators can run device verification diagnostics either stand-alone or concurrent with other processes. Diagnostics check the peripheral functions, including disk head alignment. In addition, fault isolation diagnostics, which isolate problems to replaceable units, are available for stand-alone use.

Remote Diagnosis

If the system is equipped with the remote diagnosis option, an operator sets up the system for remote preventive maintenance or troubleshooting. When a hardware error is detected or suspected, the operator mounts a diagnostic disk pack, loads a diagnostic floppy disk in the console, sets a switch on the processor console, and calls the local DIGITAL service office. An operator does not need to be present at the installation once the call is made. The DIGITAL Diagnostic Center can then connect to the installation, run automated diagnostics, operate the diagnostic console manually, and check the error log file. If a problem is found, the Field Service engineer can bring the proper equipment and replacement modules to make the repairs.

THE USER ENVIRONMENT TEST PACKAGE

The User Environment Test Package (UETP), consists of a series of tests designed to demonstrate that the hardware and software components of a system are in working order. The UETP consists of six phases:

- The Initialization Phase—This phase verifies that I/O devices are operational via simple read/write operations. In addition, users are prompted to supply several parameters which define the scope of the test, e.g.,number of users to be simulated by UETP, amount of information to be displayed at the console, and number of consecutive runs to be made by the UETP.
- The I/O Device Test Phase—In this phase, I/O devices undergo comprehensive testing. Terminals and line printers generate pages or screens of output containing header information and a test pattern of ASCII characters. Disks and magnetic tapes are also exercised. Files are created on the mounted volumes. Data are then written to the files. The test then checks the written data for errors and erases the files.
- The Native Mode Phase—This phase includes three tests, each of which exercises software services provided explicitly for VAX/VMS. The first exercises VAX/VMS system services; the second exercises native mode utilities such as the Symbolic Debugger and Image File Patch Utility; and the third exercises VAX-11 RMS.
- The System Load Test Phase—This phase creates a number of detached processes which simulate the action of a group of users concurrently issuing commands from terminals; it tests the system's ability to handle various levels of utilization.
- The Compatibility Mode Test Phase—This phase tests most RSX-11M utilities running in compatibility mode on VAX/VMS.
- Termination Phase—In this phase, temporary files are deleted and other cleanup activities are performed. If miltiple runs were requested during the initialization phase, then the UETP is restarted and control is passed directly to the device test phase.

The UETP is invoked via command procedures. The entire package may be specified by executing the master procedure, UETP.COM, or tests may be executed individually by specifying particular command procedures as illustrated in Figure 3-3.

```
$ RUN  UETINIT00          Initialization Phase
$ RUN  UETINIT01
$ RUN  UNETPDEV01         I/O Device Test Phase
$ @UETCOMP00              Compatibility Mode Test Phase
$ RUN  UETNATV01
$ @UETNATV02              Native Mode Test Phase
$ @UETNRMS00
$ RUN  UETLOAD01          System Load Test Phase
$ RUN  UETTERM01          Termination Phase
```

Figure 3-3
UETP Command Procedures

APPLICATIONS EXAMPLES

To illustrate how the multiprogramming capabilities of the system can be effective in widely diverse applications, Figures 3-4 and 3-5 show two hypothetical application systems: a commercially oriented data processing system and a real-time flight training simulation system.

Commercial System Example

The commercial system diagram (Figure 3-4) begins with both a programming group and a data processing operations group. Within the programming group, jobs can be performing requests for programmers at terminals who are using the system's text editors, compilers, and linkers to write and test programs for both the VAX/VMS and an RSX-11M system in the manufacturing department. The programmers can execute command procedures and submit batch jobs to automate repetitive development steps.

Within the operations group, the system's operators can be managing batch and spool queues, backing up disks, and monitoring performance. They may be down-line loading tasks into the RSX-11M system. They may be running an accounting program that interprets and summarizes the accounting statistics collected by the operating system during the period and sends that data to the business data processing subsystem.

A billing process within the business data processing group can be suspended until it is activated by a process (such as the accounting process) that wants to send it bill-

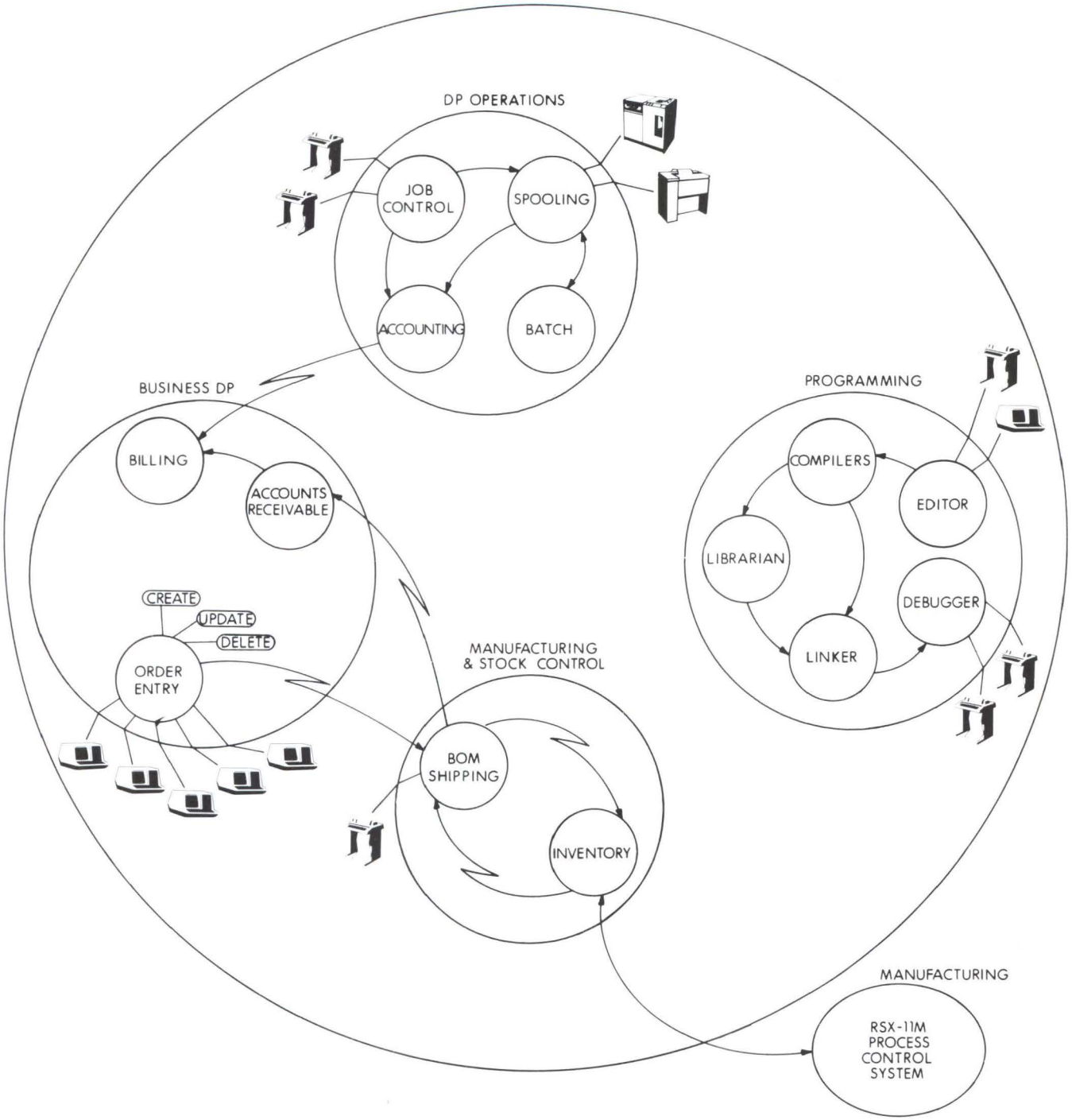


Figure 3-4

Commercial Data Processing System

ing information. The accounting process can send to the billing process the name of an account summary file that it created, using a permanent mailbox defined for that purpose.

The business data processing group may also run a job that handles order entry terminals. The job can consist of a controlling process that handles input from the terminals, and several subprocesses that perform the actual record processing functions. As users at the terminals enter their requests, such as create order record, update order record, etc., the controlling process collects the input from a terminal and sends the request to the appropriate subprocess's mailbox. The subprocess may be hibernating, having requested the operating system to activate it when

VAX-11/780 SAMPLE APPLICATION SYSTEM
FLIGHT TRAINING SIMULATION

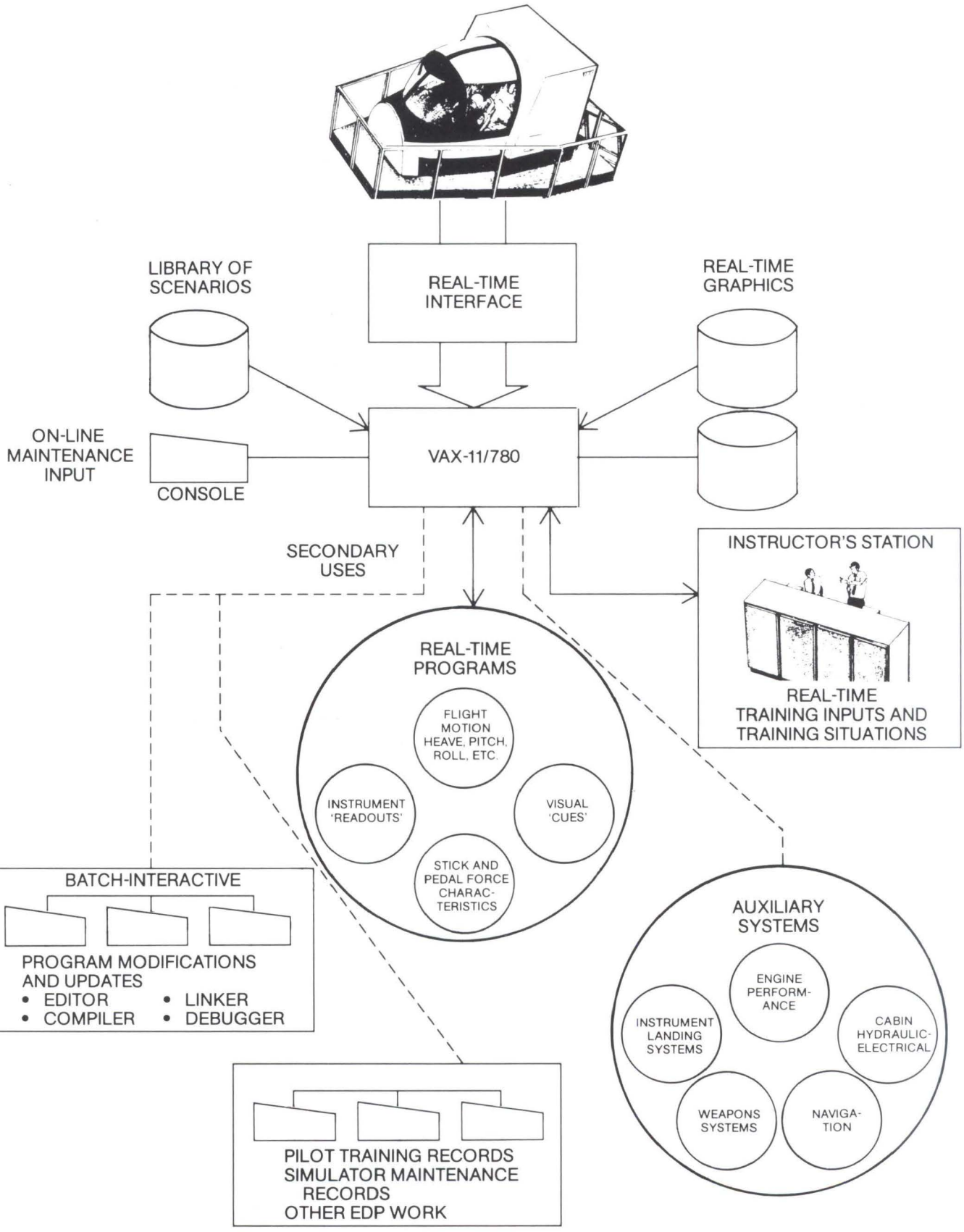


Figure 3-5
Real-time Flight Simulation System

anything is written to its mailbox. Or the controlling process can simply set an event flag to notify a subprocess that it has received a request for which the subprocess is waiting.

A process in the manufacturing application group may regularly collect orders from the order entry job to create materials parts lists, or notify stockroom clerks of high-priority orders. The stockroom clerks can keep inventory records up to date by using the shipping and inventory jobs, because they can collect manufacturing statistics from a background task in the RSX-11M process control system on the manufacturing floor. Once orders are shipped, the shipping process can notify an accounts receivable process in the business data processing application group, which in turn can activate the billing process.

Real-Time Flight Simulation Example

To illustrate how VAX's facilities can be as extended to the real-time environment, Figure 3-5 shows a sample flight training simulation system. Flight simulation is a particularly good application for the VAX system, since in addition to fast real-time response, such systems must also be capable of solving large and complex equation systems. In addition, such systems also require general program development facilities such as FORTRAN compilation, assembly, editing, debug, library facilities, and fast-access file management.

The illustrated system shows how the multiprogramming capabilities of VAX allow it to handle the basic real-time tasks of data acquisition and transmission, while also performing a wide range of other activities:

Looking from top to bottom, the diagram begins with a representation of an aircraft fuselage containing the cockpit throttle levers and control panel which the student operates to simulate flight. The signals and control movements of the student go through an analog to digital conversion, and are then passed through a real-time interface device into VAX memory. Before the system can respond to the data generated by the student, complex flight-motion equations must be called and combined with current aircraft data. This, in turn, produces a new set of circumstances with which the student must deal.

Additional inputs to the system are also provided by an instructor at a timesharing terminal (shown in the right central part of the diagram). By typing commands at the terminal, the instructor can control the situations which the student must face—for example, by injecting an engine failure, weather change, or some other complication. The instructor can also introduce additional variables into the system by inputting predefined "scenarios" which have been stored on libraries. The student's flight environment can include "visual cues" (e.g., terrain, runway approaches, other aircraft, etc.) produced by sophisticated, real-time graphics modules.

In addition to performing basic flight simulation functions, the system also performs a number of auxiliary functions which are less time-critical in nature. These would include such activities as monitoring of engine performance, testing of navigation and instrument landing systems, testing of weapons systems, and monitoring cabin, hydraulic, and electrical systems. The system also generates a file of student performance statistics which can be analyzed at a later time.

The system also provides facilities for program development. As shown in the lower left-hand side of the diagram, programmers at terminals can write and test new applications programs (in either batch or interactive mode) using text editors, compilers, linkers, and debuggers. Because of the way the VAX architecture handles real-time vs. normal processes (described below), program development can take place simultaneously with the running of the flight simulator; no significant reduction in real-time processing speed will result.

Design Considerations

Systems such as that shown in Figure 3-5 must be designed to operate at extremely high speeds, both in terms of real-time I/O and computation. Many simulators require turnaround times (data sampling, computation, and output) in the 25-50 millisecond range.

There are several elements of the VAX hardware and the VAX/VMS operating system which aid in achieving such speeds. Most important is VAX's context switching ability—a result of special processor instructions which relieve the operating system software of having to individually load or save the hardware registers which define the hardware context. Another element is the design and operation of VAX/VMS device drivers. VAX/VMS drivers are forked processes which are created dynamically in response to a user I/O request or unsolicited device interrupt. They operate with minimal context, execute to completion when invoked, and remain memory resident throughout execution. (VAX/VMS device drivers can be written for user-developed devices which interface to the VAX UNIBUS.)

In addition, system designers can utilize the VAX/VMS scheduling and system service facilities to yield still higher processing speeds. For example, the following design schema could be employed.

Those processes which perform the basic flight simulation functions (i.e., flight motion equations, visual graphics modules, etc.) can be designed as a group of hibernating processes capable of being immediately reawakened as required by the student's control activities. This could be accomplished via the use of system service calls such as ($HIBERNATE/$WAKE) or ($SUSPEND/$RESUME) or by using interprocess control structures such as mailboxes and common event blocks.

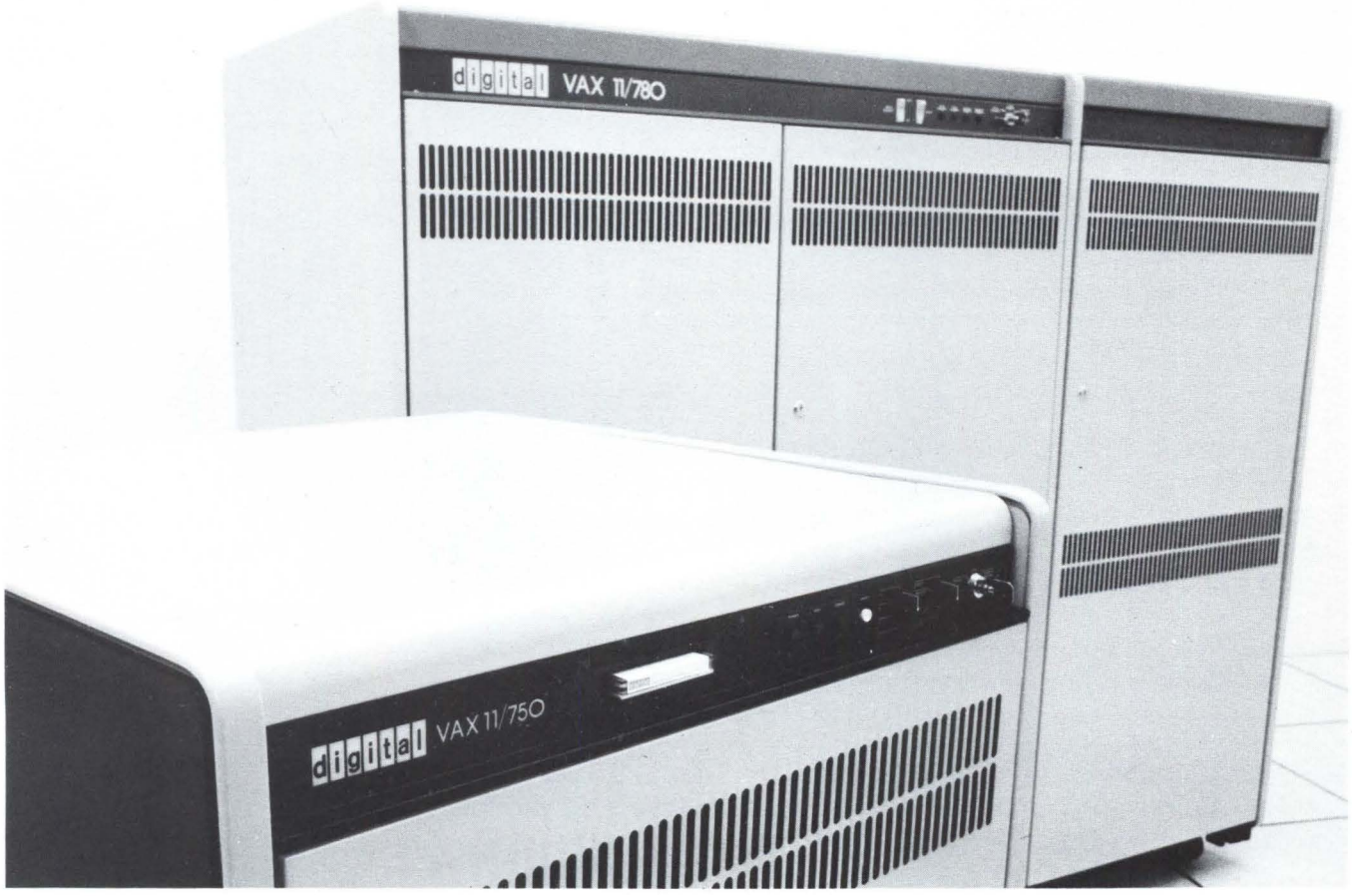
To further ensure that basic simulation functions will operate at the fastest possible speed, real-time processes can be granted the highest scheduling priorities. Such processes can also be given special privileges which allow them to eliminate paging and swapping and thus assure memory residency. (Note that when a VAX process runs at real-time priority its priority level will actually be higher than system processes such as the Swapper, Linker, and Symbionts).

Processes that are not time-critical can be assigned normal scheduling priorities. Those processes which perform what is essentially a monitoring function (e.g., engine performance, electrical system, etc.) can also be hibernated, then monitored periodically via the issuance of a programmed system timer service. Other activities such as

program development and statistical processing can take full advantage of the VAX/VMS system with minimum impact upon the basic real-time core of the simulation system.

For more information on the concepts of jobs, processes and program images, refer to the Operating System section.

4
The VAX Processors



A VAX processor executes variable-length instructions in native mode and non-privileged PDP-11 instructions in compatibility mode. The VAX processor includes integral memory management, sixteen 32-bit registers, 32 interrupt priority levels, an intelligent console, a programmable real-time clock, and a time-of-day and date clock.

The VAX native instruction set provides 32-bit addressing enabling the processor to address up to 4 billion (10^9) bytes of virtual address space. The processor's memory management hardware includes mapping registers used by the operating system, page protection by access mode, and an address translation buffer that eliminates extra memory accesses during virtual to physical address translation.

VAX also provides sixteen 32-bit general registers that can be used for temporary storage, as accumulators, index registers, and base registers. Four registers have special significance: the Program Counter and three registers that are used to provide an extensive procedure CALL facility. The processor offers a variety of addressing modes that use the general registers to identify instruction operand locations, including an indexed addressing mode that provides a true post-indexing capability.

The native instruction set is highly bit efficient. It includes integral decimal, character string, and floating point instructions, as well as integer, logical, and bit field instructions. Instructions and data are variable length and can start at any arbitrary byte boundary or, in the case of bit fields, at any arbitrary bit in memory. Floating point instruction execution can be enhanced by an optional floating point accelerator.

The I/O subsystem consists of the processor's internal bus and the UNIBUS and MASSBUS interfaces.

INTRODUCTION

This section is divided into two parts. The first discusses the architecture of a VAX processor, and the second discusses VAX processor implementation details.

VAX ARCHITECTURE

The following sections discuss the architecture, i.e., the programming characteristics of the processing system as seen from the general user's and the operating system's viewpoint.

PROCESSING CONCEPTS FOR USER PROGRAMMING

A program is a stream of instructions and data that a user can request the operating system to translate, link, and execute. An executable program is called an **image** to distinguish it from source and object programs. When a user runs an image, the context in which the image is executed is called a **process**. A process is the complete unit of execution in this computer system and typically runs several images, one after another. Process context includes the state of the image it is currently executing and includes the image's allowable limitations, which primarily depend on the privileges of the user executing the image.

The next few pages introduce some of the concepts that concern assembly language programmers in general, including addressing, data types, instruction sets, and other programming aspects of the processor. Further details on these programming characteristics follow this introduction.

Process Virtual Address Space

Most data are located in memory using the address of an 8-bit byte. The programmer uses a 32-bit virtual address to identify a byte location. This address is called a *virtual address* because it is not the real address of a physical memory location. It is translated into a real address by the processor under operating system control. A virtual address is not a unique address of a location in memory, as are physical memory addresses. Two programs using the same virtual address might refer to two different physical memory locations, or refer to the same physical memory location using different virtual addresses.

The set of all possible 32-bit virtual addresses is called virtual address space. Conceptually, virtual address space can be viewed as an array of byte "locations" labelled from 0 to $2^{32}-1$, an array that is approximately four billion bytes in length. This address space is divided into sets of virtual addresses designated for certain uses. The set of virtual addresses designated for use by a process, including an image it executes, is one half of the total virtual address space. Addresses in the remaining half of virtual address space are used to refer to locations maintained and protected by the operating system.

Instruction Sets

At any one time, the processor's instruction interpretation hardware can be set to either of two modes: native mode or compatibility mode. In native mode the processor executes a large set of variable-length instructions, recognizes a variety of data types, and uses sixteen 32-bit general purpose registers. In compatibility mode the processor executes a set of PDP-11 instructions, recognizes integer data, and uses eight 16-bit general purpose registers. While native mode is the primary instruction execution state of the machine and compatibility mode the secondary state, their instruction sets are closely related, and their programming characteristics are very similar. A user process can execute both native mode images and compatibility mode images.

A native instruction consists of an operation code (opcode) and zero or more operands, which are described by data type and addressing mode. The native instruction set is based on over 200 different kinds of operations, of each operand of which can be addressed in any one of several ways. Thus, the native instruction set offers a tremendous number of instructions from which to choose.

In spite of the large number of instructions, the native instruction set is a natural programming language that is very easy to learn. Many of the instructions correspond directly to high-level language statements, and the assembler mnemonics are readily associated with the instruction function.

To choose the appropriate instruction, it is only necessary to become familiar with the operations, data types, and addressing modes. For example, the ADD operation can be applied to any of several sizes of integer, floating point, or packed decimal operands, and each operand can be addressed directly in a register, directly in memory, or indirectly through pointers stored in registers or memory locations.

Registers and Addressing Modes

A register is a location within the processor that can be used for temporary data storage and addressing. The assembly language programmer has sixteen 32-bit general registers available for use with the native instruction set. Some of these registers have special significance. For example, one register is designated as the Program Counter, and it contains the address of the next instruction to be executed. Three general registers are designated for use with procedure linkages: the Stack Pointer, the Argument Pointer, and the Frame Pointer.

An instruction operand can be located in main memory, in a general register, or in the instruction stream itself. The way in which an operand location is specified is called the operand *addressing mode*. The processor offers a variety of addressing modes and addressing mode optimizations. There is one addressing mode that locates an operand in a register. There are six addressing modes that locate an operand in memory using a register to:

- point to the operand
- point to a table of operands
- point to a table of operand addresses

Additionally, there are six addressing modes that are indexed modifications of the addressing modes that locate an operand in memory. Finally, there are two addressing modes that identify the location of the operand in the instruction stream: one for constant data, and one for branch instruction addresses.

Data Types

The data type of an instruction operand identifies how many bits of storage are to be treated as a unit, and what

the interpretation of that unit is. The processor's native instruction set recognizes four primary data types: integer, floating point, packed decimal, and character string. For each of these data types, the selection of operation immediately tells the processor the size of the data and its interpretation. The processor can also manipulate a fifth data type, the bit field, in which the user defines the size of the field and its relative position. In addition, the processor supports two types of linked-list queue structures.

There are several variations on the four primary data types. Table 4-1 provides a summary of the data types available, each of which are illustrated in Figure 4-1. Integer data are stored as a binary value in either byte, word, longword, or, in some cases, in quadword or octaword format. A byte is eight bits, a word is two bytes, a longword is four bytes, a quadword is eight bytes, and an octaword is sixteen bytes. The processor can interpret an integer as either a signed (2's complement) value or an unsigned value. The sign is determined by the high-order bit.

Floating point values are stored using a signed exponent and a binary normalized fraction. The normalization bit is not represented. Four types of floating point data formats are provided. The two PDP-11 compatible formats (F_floating and D_floating) are standard on all VAX family processors. Two extended range formats (G_floating and H_floating) are available as options on VAX family processors. F_floating and D_floating are 4 and 8 bytes long respectively, with an 8-bit excess 128 exponent. The effective 24-bit fraction of F_floating yields approximately 7 decimal digits of precision. The 56-bit fraction of D_floating yields approximately 16 decimal digits of precision. G_floating is also 8 bytes in length, but has an 11-bit excess 1024 exponent and effectively 53 bits of fraction. Its precision is approximately 15 decimal digits. H_floating is 16 bytes in length with a 15-bit excess 16384 exponent and 113-bit fraction. Its precision is approximately 33 decimal digits.

Packed decimal data are stored in a string of bytes. Each byte is divided into two 4-bit nibbles. One decimal digit is stored in each nibble. The first, or high-order, digit is stored in the high-order nibble of the first byte, the second digit is stored in the low-order nibble of the first byte, the third digit is stored in the high-order nibble of the second byte, and so on. The sign of the number is stored in the low-order nibble of the last byte of the string.

Character data are simply a string of bytes containing any binary data, for example, ASCII codes. The first character

Table 4-1
Data Types

DATA TYPE	SIZE	RANGE (decimal)	
Integer		Signed	Unsigned
Byte	8 bits	-128 to $+127$	0 to 255
Word	16 bits	-32768 to $+32767$	0 to 65535
Longword	32 bits	-2^{31} to $+2^{31}-1$	0 to $2^{32}-1$
Quadword	64 bits	-2^{63} to $+2^{63}-1$	0 to $2^{64}-1$
Octaword	128 bits	-2^{127} to $+2^{127}-1$	0 to $2^{128}-1$
F_and D_floating point		$\pm 2.9 \times 10^{-37}$ to 1.7×10^{38}	
F_floating point	32 bits	approximately seven decimal digits precision	
D_floating	64 bits	approximately sixteen decimal digits precision	
G_floating point		$\pm 0.56 \times 10^{-308}$ to 0.9×10^{308}	
G_floating	64 bits	approximately fifteen decimal digits precision	
H_floating point		$\pm 0.84 \times 10^{-4932}$ to $\pm 0.59 \times 10^{4932}$	
H_floating	128 bits	approximately thirty-three decimal digits precision	
Packed Decimal String	0 to 16 bytes (31 digits)	numeric, two digits per byte sign in low half of last byte	
Character String	0 to 65535 bytes	one character per byte	
Variable-length Bit Field	0 to 32 bits	dependent on interpretation	
Numeric String	0 to 31 bytes (DIGITS)	$-10^{31}-1$ to $+10^{31}-1$	
Queue	$\geq$ 2 longwords/ queue entry	Queue entries at arbitrary displacement in memory	

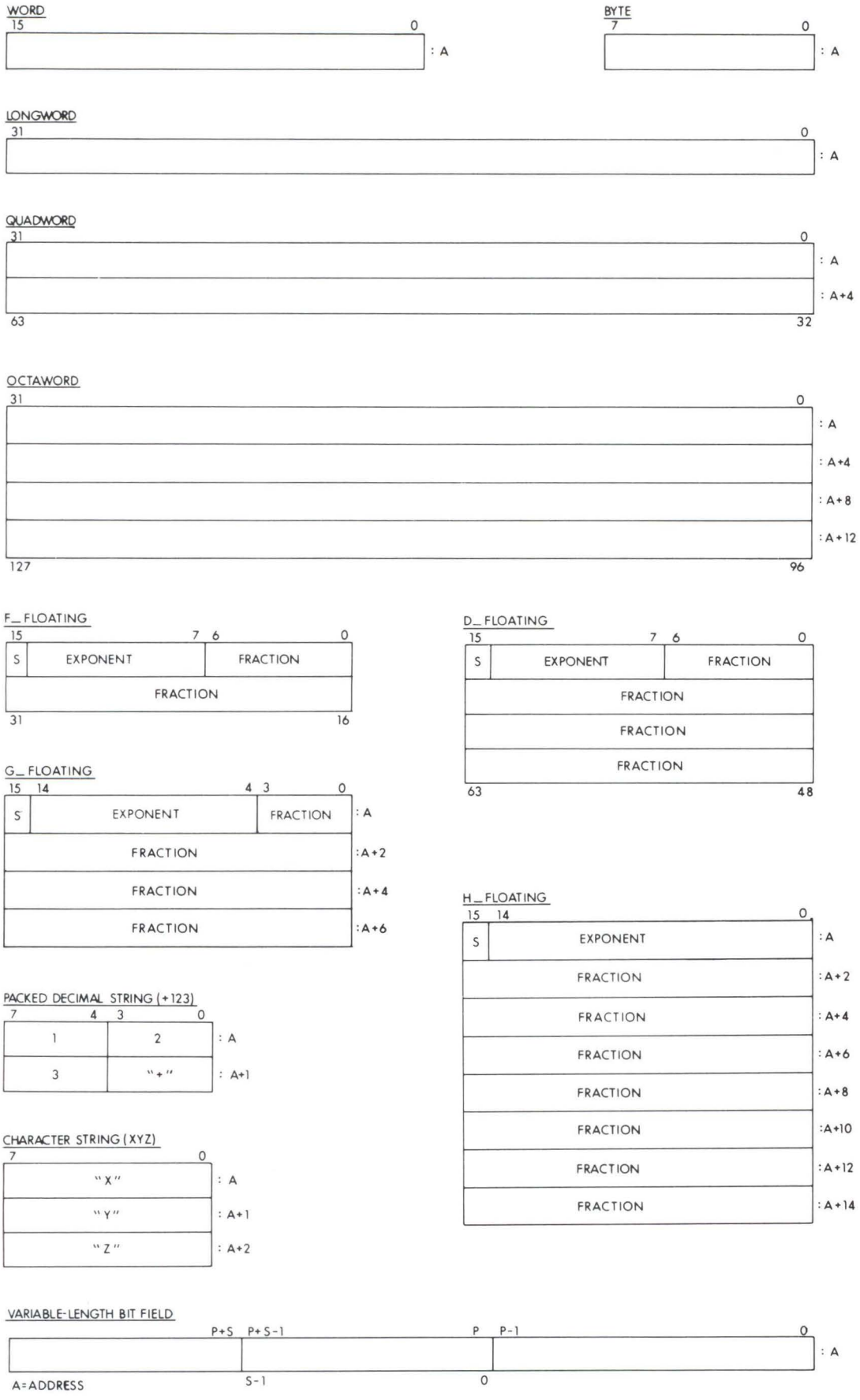


Figure 4-1
Data Type Representations

in the string is stored in the first byte, the second character is stored in the second byte, and so on. A character string that contains ASCII codes for decimal digits is called a numeric string.

The address of any data item is the address of the first byte in which the item resides. All integer, floating point, packed decimal, and character data can be stored starting on an arbitrary byte boundary. A bit field, however, does not necessarily start on a byte boundary. A field is simply a set of contiguous bits (0-32) whose starting bit location is identified relative to a given byte address or register. The native instruction set can interpret a bit field as a signed or unsigned integer.

The VAX queue data types consist of circular, doubly linked lists. A queue entry is specified by its address. Each queue entry is linked to the next entry via a pair of longwords. The first longword is the forward link; it specifies the location of the succeeding entry. The second longword is the backward link; it specifies the location of the preceding entry. VAX processors support two queue types according to the nature of the links: absolute and self-relative. An absolute link contains the absolute address of the entry that it points to. A self-relative link contains a displacement from the address of the queue entry.

Stacks, Subroutines, and Procedures

A stack is an array of consecutively addressed data items that are referenced on a last-in, first-out basis using a general register. Data items are added to and removed from the low address end of the stack. A stack grows toward lower addresses as items are added, and shrinks toward higher addresses as items are removed.

A stack can be created anywhere in the user's program address space. Any register can be used to point to the current item on the stack. The operating system, however, automatically reserves portions of each process address space for stack data structures. User software references its stack data structure, called the **user stack**, through a general register designated as the Stack Pointer. When the user runs a program image, the operating system automatically provides the address of the area designated for the user stack.

A stack is an extremely powerful data structure because it can be used to pass arguments to routines efficiently. In particular, the stack structure supports re-entrant routines because the processor can handle routine linkages automatically using the Stack Pointer. Routines can also be recursive because arguments can be saved on the stack for each successive call of the same routine.

The processor provides two kinds of routine call instructions: those for **subroutines**, and those for **procedures**. In general, a subroutine is a routine entered using a Jump to Subroutine or Branch to Subroutine instruction, while a procedure is a routine entered using a Call instruction.

The processor provides more elaborate routine linkages for procedures than for subroutines. The processor automatically saves and restores the contents of registers to be preserved across procedure calls. The processor provides two methods for passing argument lists to called procedures: by passing the arguments on the stack, or by passing the address of the arguments elsewhere in memory. The processor also constructs a "journal" of procedure call nesting by using a general register as a pointer to the place on the stack where a procedure has its linkage data. This record of each procedure's stack data, known as its **stack frame**, enables proper returns from procedures even when a procedure leaves data on the stack. In addition, user and operating system software can use the stack frame to trace back through nested calls to handle errors or debug programs.

Condition Codes

A user program can test the outcome of an arithmetic or logical operation. The processor provides a set of condition codes and branch instructions for this purpose. The condition codes indicate whether the previous arithmetic or logical operation produced a negative or zero result, or whether there was a carry or borrow, or an overflow. There are a variety of branch on condition instructions: those for overflow and carry or borrow, and those for signed and unsigned relational tests.

Exceptions

Certain situations may require that the results of an operation be tested either by the user or by the processor directly. The processor recognizes many events for which it must test directly, and automatically changes the normal flow of the user program when they occur. These events, called **exceptions**, are the direct result of executing a specific instruction. Exceptions also include errors automatically detected by the processor, such as improperly formed instructions.

All exceptions trap to operating system software. There are essentially no fatal exceptions. All exceptions either wait for the instruction that caused them to complete before trapping or they restore the processor to the state it was in just prior to executing the instruction that caused the exception. In either case, the instruction can be retried after the cause of the exception is cleared. Depending on the exception, it may be desirable to correct the situation and continue. If not, the operating system issues an appropriate message and aborts the instruction stream in progress. To continue, the user can request the operating system software to start execution of a condition handler automatically when an exception occurs.

USER PROGRAMMING ENVIRONMENT

A process context includes the definition of the virtual address space in which it executes an image. An image executing within a process context controls its execution through the use of one of the instruction sets, the general purpose registers, and the Processor Status Word. These hardware resources are discussed in detail in the following sections.

Process Virtual Address Space Structure

As mentioned earlier, certain sets of virtual addresses in virtual address space are designated for particular uses. The processor and operating system provide a multiprogramming environment by dividing virtual address space into two halves: one half for addressing context-dependent code and data, the other half for addressing context-independent code and data.

The first half is called **per-process space** (or more simply, "process space"), which is capable of addressing approxi-

mately two billion bytes. An image executing in the context of a process and the operating system on behalf of the process use addresses in process space to refer to code and data particular to that process context. A process cannot represent virtual addresses in any process space but its own. Thus, code and data belonging to a process are automatically protected from other processes in the system.

The second half of virtual address space is called **system space**. The operating system assigns specific meanings to addresses in system space. The significance of any address in system space is the same for every process, independent of process context. Although most locations referred to by system space addresses are protected from access by user images, if two images executing in different process contexts do use an address in system space, the address always refers to the same physical location in memory.

Process space is further subdivided into two equal regions. Addresses in the first region, called the **program region**, are used to identify the location of image code and data. Addresses in the second region, called the **control region**, are used to refer to stacks and other temporary program image and permanent process control information maintained by the operating system on behalf of the process. Program region addresses are allocated from address 0 up, and control region addresses are allocated from address $2^{31}-1$ down.

System space is also subdivided into two equal regions. The operating system assigns the **system region** addresses for linkages to its service procedures, for memory management data, and for I/O processing routines. The second region is presently unused.

General Registers

Instruction operands are often either stored in the processor's general registers or accessed through them. The sixteen 32-bit programmable general registers are labelled R0 through R15 (in decimal). Registers can be used for temporary storage, accumulators, base registers, and index registers. A base register contains the address of the base of a software data structure such as a table, and an index register contains a logical offset into a data structure.

Whenever a register is used to contain data, the data are stored in the register in the same format as would appear in memory. If a quadword or double floating datum is stored in a register, it is actually stored in two adjacent registers. For example, storing a double floating number in register R7 loads both R7 and R8.

Some registers have special significance depending on the instruction being executed. Registers R12 through R15 have special significance for many instructions, and therefore have special labels. These special registers are:

- The Program Counter (PC or R15), which contains the address of the next byte to be processed in the instruction stream.
- The Stack Pointer (SP or R14), which contains the address of the top of a stack maintained for subroutine and procedure calls.
- The Frame Pointer (FP or R13), which contains the address of the base of a software data structure stored on the stack called the stack frame, maintained for procedure calls.
- The Argument Pointer (AP or R12), which contains the address of the base of a software data structure called the argument list, maintained for procedure calls.

In addition, the first six registers, R0 through R5, have special significance for character and packed decimal string instructions, and the Cyclic Redundancy Check and Polynomial Evaluation instructions. These instructions use these registers to store temporary results and, upon completion, leave results in the registers that a program can use as the operands of subsequent instructions.

A register's special significance does not preclude its use for other purposes, except for the Program Counter. The Program Counter can not be used as an accumulator, as a temporary register, or as an index register. In general, however, most users do not use the Stack Pointer, Argument Pointer, or Frame Pointer for purposes other than those designated.

Addressing Modes

The processor's addressing modes allow almost any operand to be stored in a register or in memory, or as an immediate constant. Table 4-2 summarizes the addressing modes.

There are seven basic addressing modes that use the general registers to identify the operand location. They include:

- Register mode, in which the register contains the operand.
- Register Deferred mode, in which the register contains the address of the operand.
- Autodecrement mode, in which the contents of the register are first decremented by the size of the operand, and then used as the address of the operand. The size of the operand (in bytes) is given by the data type of the instruction operand, and depends on the instruction. For example, the Clear Word instruction uses a size of two, since there are two bytes per word.
- Autoincrement mode, in which the contents of the register are used as the address of the operand, and then incremented by the size of the operand. If the Program Counter is the specified register, the mode is called Immediate mode.
- Autoincrement Deferred mode, in which the contents of the register are used as the address of a location in memory containing the address of the operand, and then are incremented by four (the size of an address). If the Program Counter is the specified register, the mode is called Absolute mode.
- Displacement mode, in which the value stored in the register is used as a base address. A byte, word, or longword signed constant is added to the base address, and the resulting sum is the effective address of the operand.

Table 4-2
Addressing Modes: Assembler Syntax

Literal (Immediate)	{S^ I^} *#constant*	
Register	**R***n*	
Register Deferred	(**R***n*)	Indexed [**Rx**]
Autodecrement	–(**R***n*)	
Autoincrement	(**R***n*)+	
Autoincrement Deferred	@(**R***n*)+	
(Absolute)	@*#address*	
Displacement	{B^ W^ L^} *displacement* (**R***n*) *address*	
Displacement Deferred	@{B^ W^ L^} *displacement* (**R***n*) *address*	

n = 0 through 15
x = 0 through 14

- Displacement Deferred mode, in which the value stored in the register is used as the base address of a table of addresses. A byte, word, or longword signed constant is added to the base address, and the resulting sum is the address of the location that contains the actual address of the operand.

Of these seven basic modes, all except register mode can be modified by an index register. When an index register is used with a basic mode to identify an operand, the addressing mode is the name of the basic mode with the suffix "indexed." For example, the indexed addressing mode for register deferred is called "register deferred indexed" mode. In addition to the seven basic addressing modes that use registers, the processor recognizes six indexed addressing modes.

In an indexed addressing mode, one register is used to compute the base address of a data structure, and the other register is used to compute an index offset into the data structure. To obtain the operand's effective address in an indexed addressing mode, the processor: 1) computes the base operand address provided by one of the basic addressing modes (except register mode), 2) takes the value stored in the index register and multiplies it by the given operand size, and 3) adds the resultant value to the operand address. The index register contents are not affected and can be used for subsequent processing operations.

The processor also provides literal mode addressing, in which an unsigned 6-bit field in the instruction is interpreted as an integer or floating point constant.

The variety of addressing modes enables the assembly language programmer to write, and high-level language compilers to produce, very compact code. For example, literal mode is a very efficient way to specify small constants. The 6-bit field is interpreted as an integer when used with integer operations, and can therefore express the constants 0 through 63 (decimal). The 6-bit field is interpreted as a floating point constant when used in floating point operations, where three bits express an exponent and three a fraction.

The autoincrement and autodecrement modes enable automatic stepping through tables. Displacement mode enables generation of offsets into a table, with a choice of either short or long displacements. The deferred modes enable the user to maintain tables of operand addresses instead of the operands themselves.

The indexed addressing modes allow indexing into tables with a step size automatically determined by the operand. As in autoincrement and autodecrement addressing, the index is calculated in the context of the operand data type. This means that the user can easily access several tables of differing data types using the same index key.

Furthermore, because the indexed addressing modes enable specification of the base operand address using any mode that generates an actual address (that is, any mode except register or literal), the user has the ability to construct double indexing. The base address can be selected from a table of base addresses using displacement deferred mode, and then use an index register to provide the offset into the particular table selected.

Thus the processor's addressing modes allow considerable flexibility in the arrangement and processing of data structures. A data structure's design does not have to be tied to its processing method for efficiency.

Program Counter

A native mode instruction has a variable-length format, and instructions are thought of as byte aligned. A variable-length format not only makes code more compact, it

means that the instruction set can be extended easily. The opcode for the operation is either one or two bytes, and is followed by zero to six operand specifiers, depending on the instruction. An operand specifier can be one or several bytes long, depending on the addressing mode. Figure 4-2 illustrates the representation of an instruction as a string of bytes. Just before the processor begins to execute an instruction, the Program Counter contains the address of the first byte of the next instruction. The way in which the Program Counter is updated is totally transparent to the programmer.

The Program Counter itself can be used to identify operands. The assembler translates many types of operand references into addressing modes using the Program Counter. Autoincrement mode using the Program Counter, or **immediate mode,** is used to specify in-line constants other than those available with literal mode addressing. Autoincrement deferred mode using the Program Counter, or **absolute mode,** is used to reference an absolute address. Displacement and displacement deferred modes using the Program Counter are used to specify an operand using an offset from the current location.

Program Counter addressing enables the user to write position-independent code. Position-independent code can be executed anywhere in virtual address space after it has been linked, since program linkages can be identified as absolute locations in virtual address space and all other addresses can be identified relative to the current instruction.

Stack Pointer, Argument Pointer and Frame Pointer

The Stack Pointer is a register specifically designated for use with stack structures. Autodecrement mode addressing using the Stack Pointer can be used to place items on the stack. Auoincrement mode can be used to remove items from the stack. To reference and modify the top element on a stack without removing it, use register deferred mode, and to reference other elements of the stack use displacement mode addressing.

The processor designates Register 14 as the Stack Pointer for use with both the subroutine Branch or Jump instructions, and the procedure Call instructions. On routine entry, the processor automatically saves the address of the instruction that follows the routine call on the stack. It uses the Program Counter and the Stack Pointer to perform the operation. Before entering the subroutine, the Program Counter contains the address of the instruction following the Branch, Jump, or Call instruction; the Stack Pointer contains the address of the last item on the stack. The processor pushes the contents of the Program Counter on the stack. Returning from a subroutine, the processor automatically restores the Program Counter by popping the return address off the stack.

Also for the procedure Call instructions, the processor designates Register 12 as an Argument Pointer, and Register 13 as a Frame Pointer. The Argument Pointer is used to pass the address of the argument list to a called procedure, and the Frame Pointer is used to keep track of nested Call instructions.

An argument list is a formal data structure containing the arguments required by the procedure being called. Arguments may be actual values, addresses of data structures, or addresses of other procedures. An argument list can be passed in either of two ways: by passing only its address, or by passing the entire list on the user stack. The first method is used to pass long argument lists, or lists that are to be preserved. The second method is generally used when calling procedures that do not require arguments, or when building an argument list dynamically.

When issuing a procedure Call instruction, the processor uses the Argument Pointer to pass arguments to the procedure. If arguments were passed on the stack, the proc-

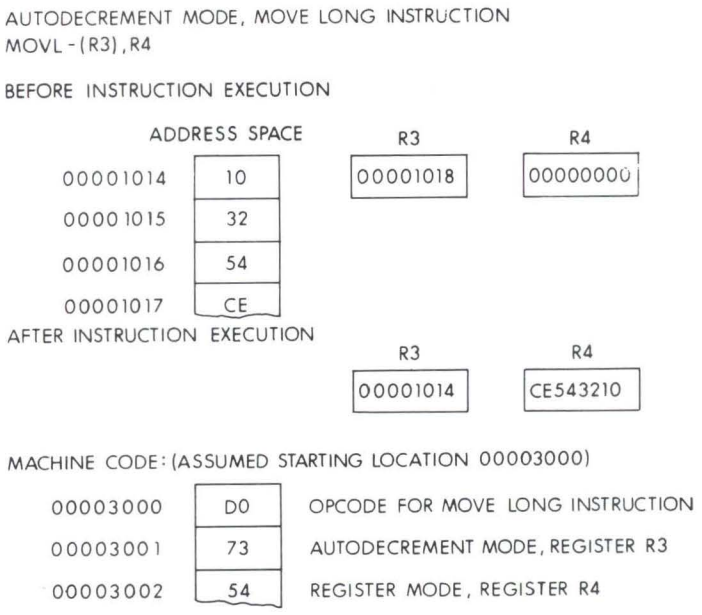


Figure 4-2
Instruction Representation

essor automatically pops the arguments off on return from the procedure.

The importance of the way the Call instructions work is that nested calls can be traced back to any previous level. The Call instructions always keep track of nested calls by using the Frame Pointer register. The Frame Pointer contains the address on the stack of the items pushed on the stack during the procedure call. The set of items pushed on the stack during a procedure call is known as a **call frame** or **stack frame**. Since the previous contents of the Current Frame register are saved in each call frame, the nested frames form a linked data structure which can be unwound to any level when an error or exception condition occurs in any procedure.

Processor Status Word

The Processor Status Word (a portion of the Processor Status Longword) is a special processor register that a program uses to check its status and to control synchronous error conditions. The Processor Status Word, illustrated in Figure 4-3, contains two sets of bit fields:

- the condition codes
- the trap enable flags

The condition codes indicate the outcome of a particular logical or arithmetic operation. For example, the Subtract instruction sets the Negative bit if the result of the subtraction operation produced a negative number, and it sets the Zero bit if the result produced zero. The Branch on Condition instructions can be used to transfer control to a code sequence that handles the condition.

There are two kinds of exceptions that concern the user process: trace faults and arithmetic exceptions. The trace fault is used by debugging programs or performance evaluators. Arithmetic exceptions include:

- integer, floating point, or decimal string overflow, in which the result was too large to be stored in the given format
- integer, floating point, or decimal string divide by zero, in which the divisor supplied was zero
- floating point underflow, in which the result was too small to be expressed in the given format

Of the arithmetic exceptions, integer overflow, floating underflow, and decimal string overflow may be handled in one of two ways. By clearing the exception enable bits in the Processor Status Word, the processor can be directed to ignore integer and decimal string overflow and floating underflow. The user may check for these conditions either by testing the condition codes (except for underflow) using the Branch on Condition instructions or by enabling the exception bits. By enabling the exception bits, the processor treats integer and decimal string overflow and floating underflow as exceptions. In any case, floating overflow and divide by zero exceptions are always enabled.

Handling Exceptions

When an exception occurs, the processor immediately saves the current state of execution and traps to the operating system. The operating system automatically searches for a procedure that wants to handle the exception. Procedures that respond to exceptions are called **condition handlers**. The user can declare a condition handler for an entire image and for each individual procedure called. In addition, because the processor keeps track of nested calls using the Frame Pointer register, it is possible to declare condition handlers for procedures that call other procedures in which exceptions might occur. The operating system automatically traces back through call frames to find a condition handler that wants to handle an exception that occurs.

NATIVE INSTRUCTION SET

The instruction set that the processor executes is selected under operating system control to either native mode or compatibility mode. The native mode instruction set is based on over 200 different opcodes. The opcodes can be grouped into classes based on their function and use. Instructions used to manipulate the general data types include:

- integer and logical instructions
- floating point instructions
- packed decimal instructions
- character string instructions
- bit field instructions

Instructions that are used to manipulate special kinds of data include:

- queue manipulation instructions

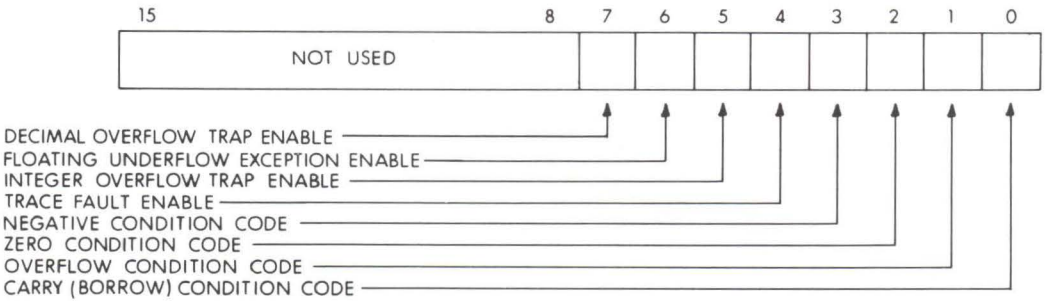


Figure 4-3
Processor Status Word

- address manipulation instructions
- user-programmed general register control instructions

Instructions that provide basic program flow control, and enable the user to call procedures are:

- branch, jump and case instructions
- subroutine call instructions
- procedure call instructions

Table 4-3 lists the basic instruction operations in order by these classifications. Instructions that enable operating system procedures to provide user mode processes with services requiring privilege are listed in the table, but discussed in the system programming environment section. Instructions that are singular in the functions they provide are listed last. The following paragraphs describe the functions of most of the instructions within each class. For further information on the instruction set, refer to the VAX-11 Architecture Handbook.

Table 4-3
Instruction Set Summary

Integer and Floating Point Logical Instructions

MOV_*	Move (B,W,L,F,D,G,H,Q,O)**
MNEG_	Move Negated (B,W,L,F,D,G,H)
MCOM_	Move Complemented (B,W,L)
MOVZ_	Move Zero-Extended (BW,BL,WL)
CLR_	Clear (B,W,L=F,Q=D=G,O=H)
CVT_	Convert (B,W,L,F,D,G,H)(B,W,L,F,D,G,H) except BB,WW,LL,FF,DD,GG,HH,DG, and GD
CVTR_L	Convert Rounded (F,D,G,H) to Longword
CMP_	Compare (B,W,L,F,D,G,H)
TST_	Test (B,W,L,F,D,G,H)
BIS_2	Bit Set (B,W,L) 2-Operand
BIS_3	Bit Set (B,W,L) 3-Operand
BIC_2	Bit Clear (B,W,L) 2-Operand
BIC_3	Bit Clear (B,W,L) 3-Operand
BIT_	Bit Test (B,W,L)
XOR_2	Exclusive OR (B,W,L) 2-Operand
XOR_3	Exclusive OR (B,W,L) 3-Operand
ROTL	Rotate Longword

Integer and Floating Point Arithmetic Instructions

INC_	Increment (B,W,L)
DEC_	Decrement (B,W,L)
ASH_	Arithmetic Shift (L,Q)
ADD_2	Add (B,W,L,F,D,G,H) 2-Operand
ADD_3	Add (B,W,L,F,D,G,H) 3-Operand
ADWC	Add with Carry
ADAWI	Add Aligned Word Interlocked
SUB_2	Subtract (B,W,L,F,D,G,H) 2-Operand
SUB_3	Subtract (B,W,L,F,D,G,H) 3-Operand
SBWC	Subtract with Carry
MUL_2	Multiply (B,W,L,F,D,G,H) 2-Operand
MUL_3	Multiply (B,W,L,F,D,G,H) 3-Operand
EMUL	Extended Multiply
DIV_2	Divide (B,W,L,F,D,G,H) 2-Operand
DIV_3	Divide (B,W,L,F,D,G,H) 3-Operand
EDIV	Extended Divide
EMOD_	Extended Modulus (F,D,G,H)
POLY_	Polynomial Evaluation (F,D,G,H)

Packed Decimal Instructions

MOVP	Move Packed
CMPP3	Compare Packed 3-Operand
CMPP4	Compare Packed 4-Operand
ASHP	Arithmetic Shift Packed and Round
ADDP4	Add Packed 4-Operand
ADDP6	Add Packed 6-Operand
SUBP4	Subtract Packed 4-Operand
SUBP6	Subtract Packed 6-Operand
MULP	Multiply Packed
DIVP	Divide Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing
CVTTP	Convert Trailing to Packed
CVTPS	Convert Packed to Separate
CVTSP	Convert Separate to Packed
EDITPC	Edit Packed to Character String

Character String Instructions

MOVC3	Move Character 3-Operand
MOVC5	Move Character 5-Operand
MOVTC	Move Translated Characters
MOVTUC	Move Translated Until Character
CMPC3	Compare Characters 3-Operand
CMPC5	Compare Characters 5-Operand
LOCC	Locate Character
SKPC	Skip Character
SCANC	Scan Characters
SPANC	Span Characters
MATCHC	Match Characters

Variable-Length Bit Field Instructions

EXTV	Extract Field
EXTZV	Extract Zero-Extended Field
INSV	Insert Field
CMPV	Compare Field
CMPZV	Compare Zero-Extended Field
FFS	Find First Set
FFC	Find First Clear

Table 4-3 (Cont.)
Instruction Set Summary

Index Instruction

INDEX	Compute Index

Queue Instructions

INSQUE	Insert Entry in Queue
INSQHI	Insert Entry into Queue at Head, Interlocked
INSQTI	Insert Entry into Queue at Tail, Interlocked
REMQUE	Remove Entry from Queue
REMQHI	Remove Entry from Queue at Head, Interlocked
REMQTI	Remove Entry from Queue at Tail, Interlocked

Address Manipulation Instructions

MOVA_	Move Address (B,W,L=F,Q=D=G,O=H)
PUSHA_	Push Address (B,W,L=F,Q=D=G,O=H) on Stack

General Register Manipulation Instructions

PUSHL	Push Longword on Stack
PUSHR	Push Registers on Stack
POPR	Pop Registers from Stack
MOVPSL	Move from Processor Status Longword
BISPSW	Bit Set Processor Status Word
BICPSW	Bit Clear Processor Status Word

Unconditional Branch and Jump Instructions

BR_	Branch with (B, W) Displacement
JMP	Jump

Branch on Condition Code

BLSS	Less Than
BLSSU	Less than Unsigned
BLEQ	Less than or Equal
BLEQU	Less than or Equal Unsigned
BEQL	Equal
(BEQLU)	(Equal Unsigned)
BNEQ	Not Equal
(BNEQU)	(Not Equal Unsigned)
BGTR	Greater than
BGTRU	Greater than Unsigned
BGEQ	Greater than or Equal
BGEQU	Greater than or Equal Unsigned
(BCC)	(Carry Cleared)
(BCS)	(Carry Set)
BVS	Overflow Set
BVC	Overflow Clear

Branch on Bit

BLB_	Branch on Low Bit (Set, Clear)
BB_	Branch on Bit (Set, Clear)
BBS_	Branch on Bit Set and (Set, Clear) Bit
BBC_	Branch on Bit Clear and (Set, Clear) Bit
BBSSI	Branch on Bit Set and Set Bit Interlocked
BBCCI	Branch on Bit Clear and Clear Bit Interlocked

Loop and Case Branch

ACB_	Add, Compare and Branch (B,W,L,F,D,G,H)
AOBLEQ	Add One and Branch Less Than or Equal
AOBLSS	Add One and Branch Less Than
SOBGEQ	Subtract One and Branch Greater Than or Equal
SOBGTR	Subtract One and Branch Greater Than
CASE_	Case on (B,W,L)

Subroutine Call and Return Instructions

BSB_	Branch to Subroutine with (B,W) Displacement
JSB	Jump to Subroutine
RSB	Return from Subroutine

Procedure Call and Return Instructions

CALLG	Call Procedure with General Argument List
CALLS	Call Procedure with Stack Argument List
RET	Return from Procedure

Protected Procedure Call and Return Instructions

CHM_	Change Mode to (Kernel, Executive, Supervisor, User)
REI	Return from Exception or Interrupt
PROBER	Probe Read
PROBEW	Probe Write

Privileged Processor Register Control Instructions

SVPCTX	Save Process Context
LDPCTX	Load Process Context
MTPR	Move to Process Register
MFPR	Move from Processor Register

Special Function Instructions

CRC	Cyclic Redundancy Check
BPT	Breakpoint Fault
XFC	Extended Function Call
NOP	No Operation
HALT	Halt

* The underscore following the instruction name implies that the instruction will operate upon any data type contained in the parentheses following that instruction.

** B = byte
W = word
L = longword
Q = quadword
O = octaword
F = F_floating
D = D_floating
G = G_floating
H = H_floating

Instructions that operate on G, H, and O formats are only available on VAX family processors equipped with the extended range floating point option.

Integer and Floating Point Instructions

The logical and integer arithmetic instructions illustrate how the opcodes, data types, and addressing modes can be combined in an instruction. Most of the operations provided for integer data are also provided for floating point and packed decimal data. Exceptions are the strictly logical operations for integer data (such as bit clear, bit set, complement), the multiword arithmetic instructions for integer data (such as Add/Subtract with Carry and Extended Multiply and Extended Divide), and the Extended Modulus and Polynomial instructions for floating point data.

The arithmetic instructions include both 2-operand and 3-operand forms that eliminate the need to move data to and from temporary operands. The 2-operand instructions store the result in one of the two operands, as in "Set A equal to A plus B." The 3-operand instructions effectively implement the high-level language statements in which two different variables are used to calculate a third, such as "Set C equal to A plus B." The 3-operand instructions are applicable to both integer and floating point data, and equivalent instructions exist for packed decimal data.

To illustrate the instruction set and addressing modes, consider the FORTRAN language statement:

A(I) = B(I) * C(I)

where A, B, and C are statically allocated REAL*4 arrays and I is INTEGER*4. A code sequence that performs this operation is:

```
MOVL    I,R0                    ;Move the longword I
                                ;to a register
MULF3   B[R0],C[R0],A[R0]       ;3-operand floating
                                ;multiply
```

The same code applies if A, B, and C are REAL*8, INTEGER*4, INTEGER*2, or even INTEGER*1 data types: the MULF3 instruction is simply changed to MULD3, MULL3, MULW3, or MULB3, respectively.

If arrays A, B, and C are dynamically allocated arrays, the code sequence could be:

```
MOVL    I,R0
MULF3   Bdisp(FP)[R0],Cdisp(FP)[R0],Adisp(FP)[R0]
```

If A, B, and C are arguments to a procedure, the code could be:

```
MOVL    I,R0
MULF3   @Bargptr(AP)[R0],@Cargptr(AP)[R0],
        ;@Aargptr[R0]
```

In fact, the locations of A, B, and C can be arbitrarily selected. For example, combining the above, if A is statically allocated, B dynamically allocated, and C an argument, then the code sequence could be:

```
MOVL    I,R0
MULF3   Bdisp(FP)[R0],@Cargptr(AP)[R0],A[R0]
```

Some of the arithmetic instructions are used for extending the accuracy of repeated computations. The Extended Multiply (EMUL) instruction takes longword integer arguments and produces a quadword result. The instruction effectively implements a high-level language statement such as "Set D equal to (A times B) plus C." The Extended Divide (EDIV) instruction divides a quadword integer by a longword and produces a longword quotient and a longword remainder.

The Extended Modulus (EMOD) instructions multiply a floating point number with an extended precision floating point number (extended by eight bits for F_floating and D_floating for an effective 9 or 19 digits of accuracy) and returns the integer portion and the fractional portion separately. This instruction is particularly useful for the reduction of the argument of trigonometric and exponential functions to a standard interval.

The Polynomial Evaluation (POLY) instructions evaluate a polynomial from a table of coefficients using Horner's method. This instruction is used extensively in the high-level languages' math library for operations such as sine and cosine.

Packed Decimal Instructions

Many of the operations for integer and floating point data also apply to packed decimal strings. They include:

- Move Packed (MOVP) for copying a packed decimal string from one location to another, and Arithmetic Shift Packed (ASHP) for scaling a packed decimal up or down by a given power of 10 while moving it, and optionally rounding the value.
- Compare Packed (CMPP) for comparing two packed decimal strings. Compare Packed has two variations: a 3-operand (CMPP3) instruction for strings of equal length, and a 4-operand instruction (CMPP4) for strings of differing lengths.
- Convert Instructions, including Convert Long to Packed (CVTLP), Convert Packed to Long (CVTPL), Convert Packed to Numeric with Trailing sign (CVTPT), Convert Numeric with Trailing sign to Packed (CVTTP), Convert Packed to Numeric with Separate overpunched sign (CVTPS), and Convert Numeric with Separate overpunched sign to Packed (CVTSP). These instructions enable the conversion of our packed decimal format to commonly used numeric formats. Numeric with trailing sign allows various sign encodings including zoned and overpunched.
- Add Packed (ADDP) and Subtract Packed (SUBP) for adding or subtracting two packed decimal strings, with the option of replacing the addend or subtrahend with the result (ADDP4 and SUBP4), or storing the result in a third string (ADDP6 or SUBP6).
- Multiply Packed (MULP) and Divide Packed (DIVP) for multiplying or dividing two packed decimal strings and storing the result in a third string.

In addition, the packed decimal instructions include a special packed decimal string to character string conversion instruction that provides output formatting: the Edit instruction.

Edit Instruction

The Edit Packed to Character String (EDITPC) instruction supplies formatted numeric output functions. The instruction converts a given packed decimal string to a character string using selected pattern operators. The pattern operators enable the creation of numeric output fields with any of the following characteristics:

- leading zero fill
- leading zero protection
- leading asterisk fill protection

- a floating sign
- a floating currency symbol
- special sign representations
- insertion characters
- blank when zero

Character String Instructions

The character string instructions operate on strings of bytes. They include:

- move string instructions, with translation options
- string compare instructions
- single character search instructions
- substring search instructions

There are two basic forms of Move instructions for character strings. The Move Character instructions (MOVC3 and MOVC5) simply copy character strings from one location to another. They are optimized for block transfer operations. The 5-operand variation provides for a fill character (user-supplied) that the instruction uses to pad out the destination location to a given size.

The Move Translated Characters (MOVTC) and Move Translated Until Character (MOVTUC) instructions actually create new character strings. The user supplies a string which the instruction uses as a list of offsets into a translation table. The instruction selects characters from the table in the order that the offset list points to the table. The MOVTC instruction allows the user to supply a fill character that the instruction uses to pad out the resultant string to a given size with an arbitrary character. The MOVTUC instruction allows the user to supply any number of escape characters. When the next offset points to an escape character in the table, translation stops.

The Compare Characters (CMPC) instructions provide character-by-character byte string compares. CMPC has a 3-operand form and a 5-operand form. Both instructions compare two strings from beginning to end, informing the user when they reach the first character that is different between the strings, or when they get to the end of either string. The 5-operand variation provides for a fill character which it uses to effectively pad out a string when comparing it with a longer one.

The Locate Character (LOCC) and Skip Character (SKPC) instructions are search instructions for single characters within a string. LOCC searches a given string for a character that matches the search character supplied by the user. This is useful, for example, when searching for the delimiter at the end of a variable-length string. SKPC, on the other hand, finds the first character in the string that is *different* from the search character supplied. This is useful for skipping through fill characters at the end of a field to find the beginning of the next field.

The Match Characters (MATCHC) instruction is similar to the Locate Character instruction, but it locates multiple-character substrings. MATCHC searches a string for the first occurrence of a substring supplied by the user.

The Span Characters (SPANC) and Scan Characters (SCANC) instructions are search instructions that look for members of character classes. For these instructions the user supplies a character string, a mask, and the address of a 256-byte table of character type definitions. For each character in the given string, the instruction looks up the type code in the table for that character, and then AND the given mask with the character's type code. SPANC finds the first character in the string which is of the type indicated by its mask. SCANC finds the first character in the string which is of any type other the one indicated by its mask.

The Index Instruction

The Index instruction (INDEX) calculates an index for an array of fixed length data types (integer and floating) and for arrays of bit fields, character strings, and decimal strings. It accepts as arguments: a subscript, lower and upper subscript bounds, an array element size, a given index, and a destination for the calculated index. It incorporates range checking within the calculation for high-level languages using subscript bounds, and it allows index calculation optimization by removing invariant expressions.

The COBOL statements:

```
01  A-ARRAY
    02      A PIC X(10) OCCURS 15 TIMES.
01  B PIC X(10).
    MOVE A(I) TO B.
```

are equivalent to:

```
INDEX I, #1, #15, #10, #0, R0   ; 1 less than or equal I
                                ; I less than or equal 15
                                ; (0 + I) * 10 is
                                ; stored in R0
MOVC3 #10, A-10[R0],B
```

The FORTRAN statements:

```
INTEGER*4 A(L1:U1, L2:U2), I, J
A(I,J) = 1
```

are equivalent to:

```
INDEX J, #L2, #U2, #M1, #0, R0      ; M1 = U1 - L1
INDEX I, #L1, #U1, #1, R0, R0
MOVL    #1, A-a[R0]                 ; a = ((L2*M1)+L1)*4
```

Variable-Length Bit Field Instructions

The bit field instructions enable the user to define, access, and modify those fields whose size and location were user-specified. Location is determined from a base address or a register and a signed bit offset. If the field is in memory, the offset can reach bits located up to 2^{31} bits (approximately 256 million bytes) away in either direction. If the field is in a register, the offset can be large as 31. Fields of arbitrary lengths (0 to 32 bits) may be used for storing data structure header information compactly, for status codes, or for creating user data types. The field instructions enable manipulation of fields easily.

The Insert Field and Extract Field instructions store data in and retrieve data from fields. Insert Field (INSV) stores data in a field by taking a specified number of bits of a longword (starting from the low-order bit) and writing them into a field, which may start at any bit relative to a given base address. The Extract Field instruction retrieves data from a field by copying the bit field and storing it in the low-order

bits of a longword. The field can either be signed (EXTV) or unsigned (EXTZV).

The Compare Field and Find First instructions enable the user to test the contents of a field. Compare Field extracts a field and then compares it with a given longword. The field can be interpreted as signed (CMPV), or as unsigned (CMPZV). The Find First instructions locate the first bit in a field that is clear (FFC) or set (FFS), scanning from low-order bit to high-order bit. These instructions are particularly useful for scanning a status control longword. For example, the longword may represent a set of queues processed in order by priority 0 (high) to 31 (low). Each set bit represents an active queue. The Find First Set instruction quickly returns the highest priority queue that is active. Together with the SKPC instructions, the Find First instructions are also useful for scanning an allocation table (bit map) of arbitrary length.

Queue Instructions

The processor has six instructions that allow easy construction and maintenance of queue data structures. Queues manipulated using the queue instructions are circular, doubly linked lists of data items.

The first longword of a queue entry contains the forward pointer to the next entry in the queue, and the next longword contains the backward pointer to the preceding entry in the queue.

Two types of queues are provided: absolute and self-relative. Absolute queues use pointers that are virtual addresses, whereas self-relative queues use pointers that are relative displacements.

Two instructions are provided for manipulating absolute queues: INSQUE, and REMQUE. INSQUE inserts an entry specified by an entry operand into the queue following the entry specified by the predecessor operand. REMQUE removes the entry specified by the entry operand. Queue entries can be on arbitrary byte boundaries. Both INSQUE and REMQUE are implemented as non-interruptible instructions.

Four operations can be performed on self-relative queues: insert at head (INSQHI), remove from head (REMQHI), insert at tail (INSQTI), and remove from tail (REMQTI). Furthermore, these operations are interlocked to allow cooperating processes in a multiprocessor system to access a shared queue without additional synchronization. Queue entries must be quadword aligned.

Address Manipulation Instructions

Because the processor offers a variety of addressing modes enabling access to data structures easily via base addresses and indices in registers, addresses are often manipulated. The processor provides two instructions enabling an address to be fetched without actually accessing the data at that location:

- The Move Address (MOVA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum in a specified register or location in memory.
- The Push Address (PUSHA) instruction, which stores the address of a byte, word, longword (and floating), or quadword (and double floating) datum on the stack.

The Push Address instruction is useful for computing an address to be passed to a called subroutine or procedure. Move Address is useful for loading a base register and performing run time position-independent address computation. It has some interesting uses because it is effectively an ADD instruction:

```
MOVAB disp(R1)[R2],X    ; sets X=R1+R2+displacement
                        ; (two adds in one instruction)

MOVA_disp(Rn)[Rn],Rn    ; multiplies Rn by 3
                        ; (for MOVAW),
                        ; 5 (for MOVAL),
                        ; or 9 (for MOVAQ)
                        ; and adds displacement to it.
```

General Register Manipulation Instructions

The general register manipulation instructions enable any user program to save or load the general purpose registers in one operation, examine the Processor Status Longword, and set or clear status bits in the Processor Status Word. (Processor register control instructions primarily used by operating system software are covered later.)

The Push Longword (PUSHL) instruction pushes a longword on the stack. This instruction is the same as a Move Longword using the Stack Pointer in register deferred mode, but is a byte shorter. It is a consistent and convenient way to move data to the stack.

The Push Registers (PUSHR) instruction pushes a set of registers on the stack in one operation. The user supplies a mask word in which each set bit (0-14) represents a register (R0-R14) to be saved on the stack. (The only general register that cannot be saved using this instruction is R15, the Program Counter.) Pop Registers (POPR) reverses the operation, loading each register from successive longwords on the stack according to the given mask word. The PUSHR and POPR instructions replace the need to write a sequence of Move instructions to save and restore registers upon entry and exit from a subroutine.

The Move from Processor Status Longword (MOVPSL) instruction allows examination of the contents of the processor's status register by loading its contents into a specified location. The Bit Set (BISPSW) and Bit Clear (BICPSW) Processor Status Word instructions enable the user to set or clear the PSW condition codes and trap enable bits. The mask bits represent the bits to be set or cleared.

Branch, Jump and Case Instructions

The two basic types of control transfer instructions are branch and jump instructions. Both branch and jump load new addresses in the Program Counter. With branch instructions, the user supplied displacement (offset) is added to the current contents of the Program Counter to obtain the new address. The jump instructions allow the user specified address to be loaded, using one of the normal addressing modes.

Because most transfers are to locations relatively close to the current instruction, and branch instructions are more efficient than jump instructions, the processor offers a variety of branch instructions to choose from. There are two unconditional branch instructions and many conditional branch instructions.

The unconditional branch instructions allow specification of either a byte-size (BRB) or a word-size displacement (BRW), thereby permitting displacements as far away from the current location as 32,767 bytes in either direction. The Jump instruction (JMP) should be used for transfer of control to locations greater in displacement than 32,767 bytes.

Most conditional branches allow only byte displacements, although some of the more powerful, such as the Add Compare and Branch instruction, allow word displacements. Conditional branch instructions include:

- branch on bit instructions
- set and clear bit instructions with a branch if it is already set or cleared
- loop instructions that increment or decrement a counter, compare it with a limit value, and branch on a relational condition
- computed branch instruction in which a branch may take place to one of several locations depending on a computed value

The Branch on Condition (B) instructions enable transfer of control to another location depending on the status of one or more of the condition codes in the Processor Status Word (PSW). There are three groups of Branch on Condition instructions:

- The signed relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as signed integers, floating point data types, and decimal strings.
- The unsigned relational branches, which are used to test the outcome of instructions operating on integer and field data types being treated as unsigned integers, character strings, and addresses.
- The overflow and carry test branches, which are used for checking overflow when traps are not enabled, for multiprecision arithmetic, and for the results of special instructions.

The instruction mnemonics clearly indicate the choice between a signed and unsigned integer data type interpretation for relational testing. The relational tests determine if the result of the previous operation is less than, less than or equal, equal, not equal, greater than or equal, or greater than zero. For example, the Branch on Less than or Equal Unsigned (BLEQU) instruction branches if either the Carry or Zero bit is set. The Branch on Greater Than (BGTR) instruction branches if neither the Negative nor the Zero bit is set.

General purpose Branch on Bit instructions similar to Branch on Condition also exist. The Branch on Low Bit Set (BLBS) and Branch on Low Bit Clear (BLBC) instructions test bit 0 of an operand, which is useful for testing Boolean values. The Branch on Bit Set (BBS) and Branch on Bit Clear (BBC) instructions test any selected bit.

There are special kinds of Branch on Bit instructions that are actually bit set/clear instructions. The Branch on Bit Set and Set (BBSS) is an example. The instruction branches if the indicated bit is set, otherwise it falls through. In either case, the instruction sets the given bit. The BBSS instruction can thus be thought of as a Bit Set instruction with a branch side-effect if the bit was already set. There are four permutations:

- Branch on Bit Set and Set (BBSS)
- Branch on Bit Clear and Clear (BBCC)
- Branch on Bit Set and Clear (BBSC)
- Branch on Bit Clear and Set (BBCS)

These instructions are particularly useful for keeping track of procedure completion or initialization, and for signaling the completion or initialization of a procedure to a cooperating process. In addition, there are two Branch on Bit Interlocked instructions that provide control variable protection:

- Branch on Bit Set and Set Interlocked (BBSSI)
- Branch on Bit Clear and Clear Interlocked (BBCCI)

The memory interconnect bus provides a memory interlock on these instructions. No other BBSSI or BBCCI operation can interrupt these instructions to gain access to the byte containing the control variable between the testing of the bit and the setting or clearing of the bit.

The processor offers three types of branch instructions that can be used to write efficient loops. The first type provides the basic subtract-one-and-branch loop. A counter variable (user-supplied) is decremented each time the loop is executed. In the Subtract One and Branch Greater Than (SOBGTR) instruction, the loop repeats until the counter equals zero. In the Subtract One and Branch Greater Than or Equal (SOBGEQ) instruction, the loop repeats until the counter becomes negative.

The counterpart to subtract-one-and-branch is add-one-and-branch. A counter and a limit must be supplied by the user. The counter is incremented at the end of the loop. In the Add One and Branch Less Than (AOBLSS) instruction, the loop repeats until the counter equals the user-defined limit. In the Add One and Branch Less Than or Equal (AOBLEQ) instruction, the loop repeats until the counter exceeds the user defined limit.

The third type of loop instruction efficiently implements the FORTRAN language DO statement and the BASIC language FOR statement: Add Compare and Branch (ACB). In this case, the user must supply a limit, a counter, and a step value. For each execution of the loop, the instruction adds the step value to the counter and compares the counter to the limit. The sign of the step value determines the logical relation of the comparison: the instruction loops on a less than or equal comparison if the step value is positive, on a greater than or equal comparison if the step value is negative.

The processor provides a branch instruction that implements higher-level language computed GO TO statements: the CASE instruction. To execute the CASE instruction, the user must supply a list of displacements that generate different branch addresses indexed by the value obtained as a selector. The branch falls through if the selector does not fall within the limits of the list.

Subroutine Branch, Jump, and Return Instructions

Two special types of branch and jump instruction are provided for calling subroutines: the Branch to Subroutine (BSB) and Jump to Subroutine (JSB) instructions. Both BSB and JSB instructions save the contents of the Program Counter on the stack before loading the Program Counter with the new address. With Branch to Subroutine,

the user supplies either a byte (BSBB) or word (BSBW) displacement. With Jump to Subroutine, regular addressing is used.

The subroutine call instructions are complemented by the Return from Subroutine (RSB) instruction. RSB pops the first longword off the stack and loads it into the Program Counter. Since the Branch to Subroutine instruction is either two or three bytes long, and the Return from Subroutine instruction is one byte long, it is possible to write extremely efficient programs using subroutines.

Procedure Call and Return Instructions

Procedures are general purpose routines that use argument lists passed automatically by the processor. The procedure Call instructions enable language processors and the operating system to provide a standard calling interface. They:

- save all the registers that the procedure uses, and only those registers, before entering the procedure
- pass an argument list to a procedure
- maintain the Stack, Frame, and Argument Pointer registers
- initialize the arithmetic trap enables to a given state

When issuing a Call procedure instruction, the address of the procedure being called, must be included. The first word of a procedure contains an entry mask that is used in the same way as the entry mask defined for the Push Registers instruction. Each set bit of the 12 low-order bits in the word represents one of the general registers, R0 through R11, that the procedure uses. The Call instruction examines this word and saves the indicated registers on the stack. In addition, the Call instruction also automatically saves the contents of the Frame Pointer, Argument Pointer, and Program Counter registers. This is an extremely efficient way to ensure that registers are saved across procedure calls. No general register is saved that does not have to be saved.

The Call Procedure with General Argument List (CALLG) instruction accepts the address of an argument list and passes the address to the procedure in the Argument Pointer register. The Call Procedure with Stack Argument List (CALLS) passes the argument list, (placed on the stack by the user) by loading the Argument Pointer register with its stack address.

When a procedure completes execution, it issues the Return from Procedure instruction (RET). Return uses the Frame Pointer register to find the saved registers that it restores, and to clean up any data left on the stack, including nested routine linkages. A procedure can return values using the argument list or other registers.

Miscellaneous Special Purpose Instructions

The processor has a number of special purpose instructions. They include:

- Cyclic Redundancy Check (CRC)
- Breakpoint Fault (BPT)
- Extended Function Call (XFC)
- No Operation (NOP)
- Halt

The Cyclic Redundancy Check (CRC) instruction calculates a cyclic redundancy check for a given string using any CRC polynomial up to 32 bits long. The user supplies the string for which the CRC is to be performed, and a table for the CRC function. The operating system library includes tables for standard CRC functions, such as CRC-16.

The Breakpoint Fault (BPT) instruction makes the processor execute the kernel mode condition handler associated with the Breakpoint Fault exception vector. BPT is used by the operating system debugging utilities, but can also be used by any process that sets up a Breakpoint Fault condition handler.

The Extended Function Call (XFC) instruction allows escapes to customer-defined instructions in writable control store. The NOP instruction is useful for debugging. The HALT instruction is a privileged instruction issued only by the operating system to halt the processor when bringing the system down by operator request.

COMPATIBILITY MODE

Under control of the operating system, the processor can execute PDP-11 instruction streams within the context of any process. When executing in compatibility mode, the processor interprets the instruction stream executing in the context of the current process as a subset of the PDP-11 instruction set.

In general, compatibility mode enables the operating system to provide an environment for executing most user mode programs written for a PDP-11 except stand-alone software. The processor expects all compatibility mode software to rely on the services of the native operating system for I/O processing, interrupt and exception handling, and memory management. There are some restrictions, however, on the environment that the native operating system can provide a PDP-11 program. For example, the PDP-11 memory management instructions Move To/From Previous Instruction/Data Space can not be simulated by the operating system since they do not trap to native mode software.

PDP-11 Program Environment

PDP-11 addresses are 16-bit byte addresses. There is a one-to-one correspondence between compatibility mode virtual addresses and the first 64K bytes of virtual address space available to native mode processes. As in the PDP-11, a compatibility mode program is restricted to referencing only these addresses. It is possible for the operating system to provide most of the PDP-11 memory management mechanisms. For example, compatibility mode automatically supports PDP-11 memory segment protection, but in 512-byte rather than 64-byte segments.

All of the PDP-11 general registers and addressing modes are available in compatibility mode. Compatibility mode registers R0 through R6 are the low-order 16 bits of native mode registers R0 through R6. Compatibility mode R7 (the Program Counter) is the low-order bits of native mode register 15 (the Program Counter). Native mode registers 8 through 14 are not affected by compatibility mode. Note that the compatibility mode register R6 acts as the Stack Pointer for program-local temporary data storage, but that the program-local stack is allocated address space in the program region, not the control region.

A subset of the PDP-11 Processor Status Word is defined for compatibility mode. Only the condition codes and the trace trap bit are relevant for the PDP-11 instruction stream.

All interrupts and exceptions that occur when the processor is executing in compatibility mode cause the processor to enter native mode. As in native mode, it is the operating system's responsibility to handle interrupts and exceptions. There are a few types of exceptions that apply only to compatibility mode. They include illegal instruction exceptions and odd address trap.

PDP-11 Instruction Set

The compatibility mode instruction set is that of the PDP-11 with the following exceptions:

- The privileged instructions (HALT, WAIT, RESET, SPL, and MARK) are illegal.
- The trap instructions (BPT, IOT EMT, and TRAP) cause the processor to enter native mode, where either the trap may be serviced, or the instruction simulated
- The Move From/To Previous Instruction/Data space instructions (MFPI, MTPI, MFPD, and MTPD) execute exactly as they would on a PDP-11 in user mode with instruction and data space overmapped. They ignore the previous access level and act as PUSH and POP instructions referencing the current stack.
- PDP-11 floating point instructions are emulated through software.

All other instructions execute as they would on a PDP-11/70 processor running in user mode.

PROCESSING CONCEPTS FOR SYSTEM PROGRAMMING

The processor is specifically designed to support a high-performance multiprogramming environment. The chief advantage of a multiprogramming system is its ability to get the most out of a computer that is being used for several different purposes concurrently. For example, multiprogramming enables the simultaneous execution of two or more application systems, such as process control and order entry. It is also possible to execute several application systems while simultaneously developing application programs. The characteristics of the hardware system that support multiprogramming are:

- rapid context switching
- priority dispatching
- virtual addressing and memory management

As a multiprogramming system, VAX not only provides the ability to share the processor among processes, but also protects processes from one another while enabling them to communicate with each other and share code and data.

Context Switching

In a multiprogramming environment, several individual streams of code can be ready to execute at any one time. Instead of allowing each stream to execute to completion serially (as in a batch-only system), the operating system can intervene and switch between the streams of code which are ready to execute.

To support multiprogramming for a high-performance system, the processor enables the operating system to switch rapidly between individual streams of code. The stream of code the processor is executing at any one time is determined by its *hardware context*. Hardware context includes the information loaded in the processor's registers that identifies:

- where the stream's instructions and data are located
- which instruction to execute next
- what the processor status is during execution

A process is a stream of instructions and data defined by a hardware context. Each process has a unique identification in the system. The operating system switches between processes by requesting the processor to save one process hardware context and load another. Context switching occurs rapidly because the processor instruction set includes save hardware context and load hardware context instructions. The operating system's context switching software does not have to individually save or load the processor registers which define the hardware context.

The actual scheduling mechanism for arbitrating among processes competing for processor time is left to the operating system software itself to give the system flexibility.

Priority Dispatching

While running in the context of one process, the processor executes instructions and controls data flow to and from peripherals and main memory. To share processor, memory and peripheral resources among many processes, the processor provides two arbitration mechanisms that support high-performance multiprogramming: exceptions and interrupts. Exceptions are events that occur synchronously with respect to instruction execution, while interrupts are external events that occur asynchronously.

The flow of execution can change at any time, and the processor distinguishes between changes in flow that are local to a process and those that are system-wide. Process-local changes occur as the result of a user software error or when user software calls operating system services. Process-local changes in program flow are handled through the processor's exception detection mechanism and the operating system's exception dispatcher.

System-wide changes in flow generally occur as the result of interrupts from devices or interrupts generated by the operating system software. Interrupts are handled by the processor's interrupt detection mechanism and the operating system's interrupt service routines. (System-wide changes in flow may also occur as the result of severe hardware errors, in which case they are handled either as special exceptions or high-priority interrupts.)

System-wide changes in flow take priority over process-local changes in flow. Furthermore, the processor uses a priority system for servicing interrupts. To arbitrate between all possible interrupts, each kind of interrupt is assigned a priority, and the processor responds to the highest priority interrupt pending. For example, interrupts from real-time I/O devices would take precedence over interrupts from mass-storage devices, terminals, line printers and other less time-critical devices.

The processor services interrupts between instructions, or at well-defined points during the execution of long, itera-

tive instructions. When the processor acknowledges an interrupt, it switches rapidly to a special system-wide context to enable the operating system to service the interrupt. System-wide changes in the flow of execution are handled in such a way as to be totally transparent to individual processes.

Virtual Addressing and Virtual Memory

The processor's memory management hardware enables the operating system to provide an execution environment that allows users to write programs without having to know where the programs are loaded in physical memory, and to write programs that are too large to fit in the physical memory they are allocated.

The processor provides the operating system with the ability to provide virtual addressing. A virtual address is a 32-bit integer that a program uses to identify storage locations in virtual memory. Virtual memory is the set of all physical memory locations in the system plus the set of disk blocks that the operating system designates as extensions to physical memory.

A physical address is an address that the processor uses to identify physical memory storage locations and peripheral controller registers. It is the physical address that the processor sends to the memory and peripheral adaptors.

The processor must be capable of translating virtual addresses provided by the executing program into the physical addresses recognized by the memory and peripherals. To provide virtual to physical address mapping, the processor has address mapping registers controlled by the operating system and an integrated address translation buffer.

The mapping registers enable the operating system to relocate programs in physical memory, to protect programs from each other, and share instructions and data between programs transparently or at their request. The address translation buffer ensures that the virtual address to physical address translation takes place rapidly.

SYSTEM PROGRAMMING ENVIRONMENT

Within the context of one process, user-level software controls its execution using the instruction sets, the general registers and the Processor Status Word. Within the multiprogramming environment, the operating system controls the system's execution using a set of special instructions, the Processor Status Longword, and the internal processor registers.

Processor Status Longword

A processor register called the Processor Status Longword (PSL) determines the execution state of the processor at any time. The low-order 16 bits of the Processor Status Longword is the Processor Status Word available to the user process. The high-order 16 bits provide privileged control of the system. Figure 4-4 illustrates the Processor Status Longword.

The fields can be grouped together by functions that control:

- the instruction set the processor is executing
- the access mode of the current instruction
- interrupt processing

The instruction set the processor executes is controlled by the compatibility mode bit in the Processor Status Longword. This bit is normally set or cleared by the operating system. For further information on compatibility mode, refer to the Operating System section.

The following paragraphs discuss access modes, the native instructions primarily used by the operating system, memory management, and interrupt processing.

Processor Access Modes

In a high-performance multiprogramming system, the processor must provide the basis for protection and sharing among the processes competing for the system's resources. The basis for protection in this system is the processor's access mode. The access mode in which the processor executes determines:

- instruction execution privileges: what instructions the processor will execute
- memory access privileges: which locations in memory the current instruction can access

At any one time, the processor is executing code in the context of a particular process, or it is executing in the system-wide interrupt service context. In the context of a process, the processor recognizes four access modes: kernel, executive, supervisor, and user. Kernel is the most privileged mode and user the least privileged.

The processor spends most of its time executing in user mode in the context of one process or another. When user software needs the services of the operating system, whether for acquisition of a resource, for I/O processing, or for information, it calls those services.

The processor executes those services in the same or one

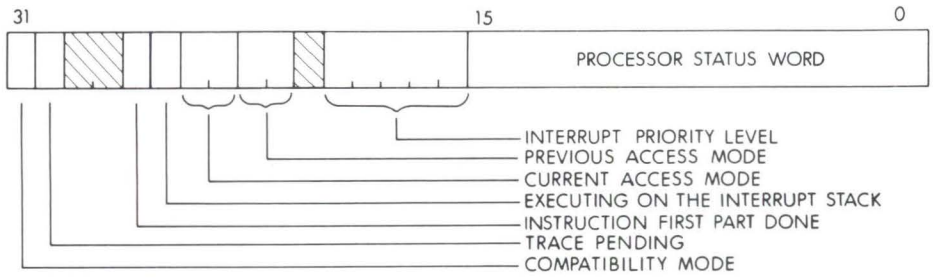


Figure 4-4
Processor Status Longword

of the more privileged access modes within the context of that process. That is, all four access modes exist within the same virtual address space. Each access mode has its own stack in the control region of per-process space, and therefore each process has four stacks: one for each access mode. Note that this makes it easy for the operating system to context switch a process even when it is executing an operating system service procedure.

In any mode except kernel, the processor will not execute the instructions that:

- halt the processor
- load and save process context
- access the internal processor registers that control memory management, interrupt processing, the processor console, or the processor clock

These instructions are privileged instructions that are generally reserved to the operating system.

In any mode, the processor will not allow the current instruction to access memory unless the mode is privileged to do so. The ability to execute code in one of the more privileged modes is granted by the system manager and controlled by the operating system. The memory protection the privilege affords is enforced by the processor. In general, code executing in one mode can protect itself and any portion of its data structures from read and/or write access by code executing in any less privileged mode. For example, code executing in executive mode can protect its data structures from code executing in supervisor or user mode. Code executing in supervisor mode can protect its data structures from access by code executing in user mode. This memory protection mechanism provides the basis for system data structure integrity.

Protected and Privileged Instructions

The processor provides three types of instructions that enable user mode software to obtain operating system services without jeopardizing the integrity of the system. They include:

- the Change Mode instructions
- the PROBE instructions
- the Return from Exception or Interrupt instruction

User mode software can obtain privileged services by calling operating system service procedures with a standard CALL instruction. The operating system's service dispatcher issues an appropriate Change Mode instruction before actually entering the procedure. Change Mode allows access mode transitions to take place from one mode to the same or more privileged mode only. When the mode transition takes place, the previous mode is saved in the Previous Mode field of the Processor Status Longword, allowing the more privileged code to determine the privilege of its caller.

A Change Mode instruction is simply a special trap instruction that can be thought of as an operating system service call instruction. User mode software can explicitly issue Change Mode instructions, but since the operating system receives the trap, non-privileged users can not write any code to execute in any of the privileged access modes. User mode software can include a condition handler for Change Mode to User traps, however, and this instruction is useful for providing general purpose services for user mode software. The system manager ultimately grants the privilege to write any code that handles Change Mode traps to more privileged access modes.

For service procedures written to execute in privileged access modes (kernel, executive, and supervisor), the processor provides address access privilege validation instructions. The PROBE instructions enable a procedure to check the read (PROBER) and write (PROBEW) access protection of pages in memory against the privileges of the caller who requested to access a particular location. This enables the operating system to provide services that execute in privileged modes to less privileged callers and still prevent the caller from accessing protected areas of memory.

The operating system's privileged service procedures and interrupt and exception service routines exit using the Return from Exception or Interrupt (REI) instruction. REI is the only way in which the privilege of the processor's access mode can be decreased. Like the procedure and subroutine return instructions, REI restores the Program Counter and the processor state to resume the process at the point where it was interrupted.

REI performs special services, however, that normal return instructions do not. For example, REI checks to see if any asynchronous system traps have been queued for the currently executing process while the interrupt or exception service routine was executing, and ensures that the process will receive them. Furthermore, REI checks to ensure that the mode to which it is returning control is the same as or less privileged than the mode in which the processor was executing when the exception or interrupt occurred. Thus REI is available to all software including user-written trap handling routines, but a program cannot increase its privilege by altering the processor state to be restored.

When the operating system schedules a context switching operation, the context switching procedure uses the Save Process Context (SVPCTX) and Load Process Context (LDPCTX) instructions to save the current process context and load another. The operating system's context switching procedure identifies the location of the hardware context to be loaded by updating an internal processor register.

Internal processor registers not only include those that identify the process currently executing, but also the memory management and other registers, such as the console and clock control registers. The Move to Processor Register (MTPR) and Move from Processor Register (MFPR) instructions are the only instructions that can explicitly access the internal processor registers. MTPR and MFPR are privileged instructions that can be issued only in kernel mode.

Memory Management

The processor is responsible for enforcing memory protection between access modes. Memory protection, however, is only a part of the processor's memory management function. In particular, the memory management hardware enables the operating system to provide an extremely flexible and efficient virtual memory programming environment. Virtual and physical address space defi-

nitions provide the basis for the virtual memory available on a system.

Virtual address space consists of all possible 32-bit addresses that can be exchanged between a program and the processor to identify a byte location in physical memory. The memory management hardware translates a virtual address into a physical address. A physical address can be up to 30 bits in length as in the case of the VAX-11/780. Other processor implementations may choose a smaller physical address. A physical address is the address exchanged between the processor and the memory and peripheral adaptors. Physical address space is the set of all possible physical addresses the processor can use to express unique memory locations and peripheral control registers. Figure 4-5 compares the structure of the common virtual address space with that of the VAX-11/750 and VAX-11/780 physical address spaces.

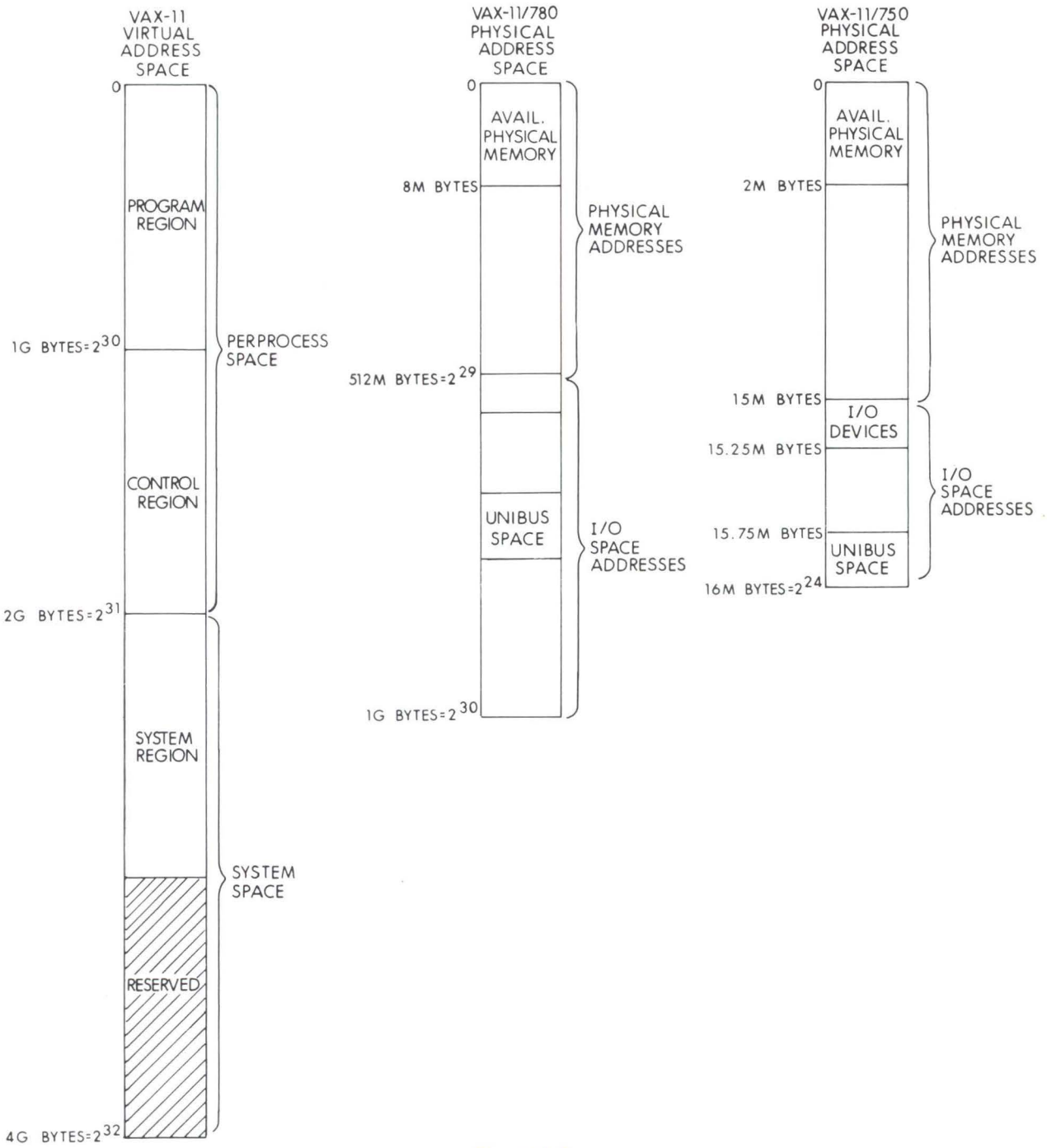


Figure 4-5

Virtual and Physical Address Space

On the VAX-11/780, physical address space is an array of addresses which can be used to represent 2^{30} byte locations, or approximately one billion bytes. Half of the addresses in VAX-11/780 physical address space can be used to refer to real memory locations and the other half can be used to refer to peripheral device control and data registers. The lowest-addressed half of physical address space is called **memory space**, and the highest-addressed half **I/O space**. On the VAX-11/750, physical address space is an array of addresses which can be used to represent 2^{24} byte locations, or approximately 16 million bytes. The first 15M bytes are dedicated to physical memory addresses while the last 1M byte is dedicated to I/O space.

The following section describes the way in which the memory management hardware enables the operating system to map virtual addresses into physical addresses to provide the virtual memory available to a process.

Virtual to Physical Page Mapping

Virtual address space is divided into pages, where a page represents 512 bytes of contiguously addressed memory. The first page begins at byte zero and continues to byte 511. The next page begins at byte 512 and continues to byte 1023, and so forth. For example, decimal and hexadecimal addresses of the first eight pages of virtual addresss space are:

PAGE	ADDRESS(10) decimal	ADDRESS(16) hexadecimal
0	0000-0511	0000-01FF
1	0512-1023	0200-03FF
2	1024-1535	0400-05FF
3	1536-2047	0600-07FF
4	2048-2559	0800-09FF
5	2560-3071	0A00-0BFF
6	3072-3583	0C00-0DFF
7	3584-4095	0E00-0FFF

The size of a virtual page exactly corresponds to the size of a physical page of memory, and the size of a block on disk.

To make memory mapping efficient, the processor must be capable of translating virtual addresses to physical addresses rapidly. Two features providing rapid address translation are the processor's internal address translation buffer and the translation algorithm itself.

Figure 4-6 compares the virtual address format to the physical address formats of the VAX-11/780 and VAX-11/750 processors. The high-order two bits of a virtual address immediately identify the region to which the virtual address refers. Whether the address is physical (processor specific) or virtual, the byte within the page is the same. Thus, the processor has to know only which virtual pages correspond to which physical pages.

The processor has three pairs of page mapping registers, one pair for each of the three regions actively used. The operating system's memory management software loads each pair of registers with the base address and length of data structures it sets up called **page tables**. The page tables provide the mapping information for each virtual page in the system. There is one page table for each of the three regions.

A page table is a virtually contiguous array of page table entries. Each page table entry is a longword representing the physical mapping for one virtual page. To translate a virtual address to a physical address, therefore, the processor simply uses the virtual page number as an index into the page table from the given page table base address. Each translation is good for 512 virtual addresses since the byte within the virtual page corresponds to the byte within the physical page.

Figure 4-7 shows the format of a page table entry. The high-order bits are used to indicate the page's status and protection. The page's protection can be set to prevent read and/or write access by any mode (kernel, executive, supervisor, or user). The page's status indicates what the remainder of the page table entry means. It may be, for example, a page address in physical address space, a disk sector, or a temporary pointer to a page shared by two or more processes. The system's **virtual memory** is a dynamic memory that is defined by the physical memory and disk pages that are virtually mapped by page table entries.

The operating system's memory management software maintains the page table entry protection and status bits, with the exception of the modified page bit. The processor sets the modified page bit to indicate that it has written into a physical page in memory. This is used to keep disk I/O to a minimum when paging a process.

The processor uses the page table base registers to locate the page tables, and uses the length registers as a validity check to ensure that any given virtual page is in the range of defined page table entries. Figure 4-8 summarizes and compares the page table structures.

All process page tables have virtual addresses in the system region of virtual address space, but the system region page table is located by its address in physical memory. That is, the system region page table base register contains the *physical address* of the page table base, while the process page table base registers contain the *virtual addresses* of their page table bases. Because a per-process page table entry is referred to by a virtual address in the system region, the hardware translates its virtual address using the system page table.

There are two advantages to using a virtual address as the base address of a per-process page table. The first advantage is that all page tables do not have to reside in physical memory. The system region page table is the only page table that needs to be resident in physical memory. All process page tables can reside on disk; that is, process page tables can themselves be paged and swapped as necessary.

The second advantage is that the operating system's memory management software can allocate per-process page tables dynamically, because the per-process page tables do not need to be mapped into contiguous physical pages. And although the system region page table must be mapped into contiguous physical pages, this requirement does not restrict physical memory allocation. The region is shared among processes, and therefore does not require redefinition from context to context.

To illustrate the efficiency of this memory mapping scheme, suppose that 16 processes, each of which is us-

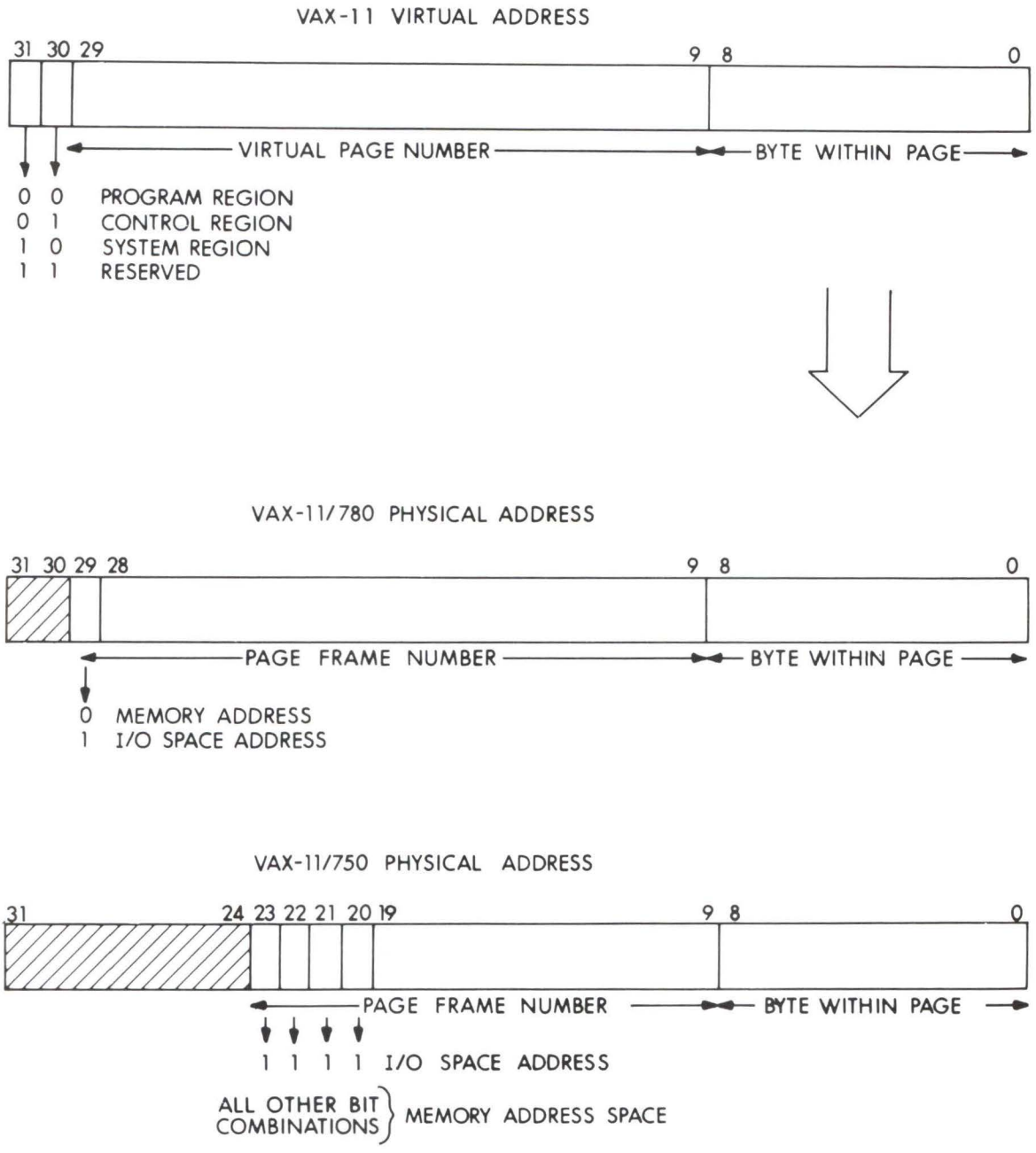


Figure 4-6

Virtual and Physical Address Formats

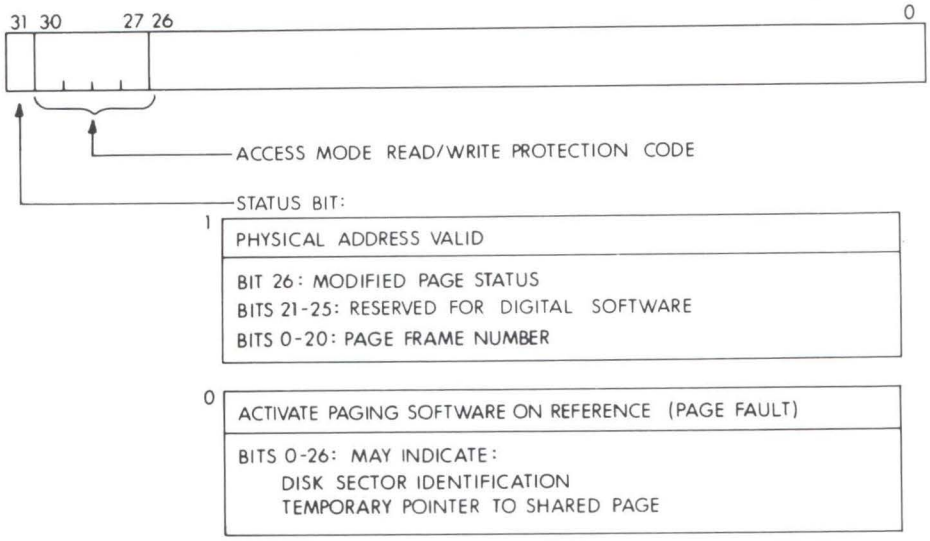


Figure 4-7

Page Table Entry

ing 4 million bytes of virtual address space, are known to the system at the same time (for a total of 64 Mb of virtual address space). One system page table entry maps one page of per-process page table entries, and one page of per-process page table entries maps 65,536 (64K) bytes of virtual address space (since it is possible to store 128 page table entries in a single page of memory). Therefore one page of system page table maps 128 pages of per-process page tables, which in turn maps 8 Mb of process virtual address space. Thus the system region page table needed to map these 16 processes requires approximately 8 physical pages (4K bytes) of memory.

Exception and Interrupt Vectors

The processor can automatically initiate changes in the normal flow of program execution. The processor recognizes two kinds of events that cause it to invoke conditional software: exceptions and interrupts. Some exceptions affect an individual process only, such as arithmetic traps, while others affect the system as a whole, for example, machine check. Interrupts include both device interrupts, such as those signaling I/O completion, and software-requested interrupts, such as those signaling the need for a context switch operation.

The processor knows which software to invoke when an exception or interrupt occurs because it references specific locations, called vectors, to obtain the starting address of the exception or interrupt dispatcher. The processor has one internal register, the System Control Block Base Register, which the operating system loads with the physical address of the base of the System Control Block, which contains the exception and interrupt vectors. The processor locates each vector by using a specific offset into the System Control Block. Figure 4-9 illustrates the vectors in the System Control Block. Each vector tells the processor how to service the event, and contains the system region virtual address of the routine to execute. Note that vector 14 (hex) can be used as a trap to writable control store to execute user-defined instructions, and the vector contains information passed to microcode.

Interrupt Priority Levels

Exceptions do not require arbitration since they occur synchronously with respect to instruction execution. Interrupts, on the other hand, can occur at any time. To arbitrate between interrupt requests that may occur simultaneously, the processor recognizes 31 interrupt priority levels.

The highest 16 interrupt priority levels are reserved for interrupts generated by hardware, and the lowest 16 interrupt priority levels are reserved for interrupts requested by software. Table 4-4 lists the assignment of each level, from highest to lowest priority. Normal user software runs at process level, which is interrupt priority level zero.

To handle interrupt requests, the processor enters a special system-wide context. In the system-wide context, the processor executes in kernel mode using a special stack called the interrupt stack. The interrupt stack cannot be referenced by any user mode software because the processor only selects the interrupt stack after an interrupt, and all interrupts are trapped through system vectors.

The interrupt service routine executes at the interrupt priority level of the interrupt request. When the processor receives an interrupt request at a level higher than that of the currently executing software, the processor honors the request and services the new interrupt at its priority level. When the interrupt service routine issues the REI (Return from Exception or Interrupt) instruction, the processor returns control to the previous level.

System Base Register
(contains the physical address of the first entry of the page table)

System Length Register
(contains the number of page table entries, N)

Program Region Base Register
(contains the virtual address of the first entry in the page table)

Program Region Length Register
(contains the number of page table entries, N)

Control Region Base Register
(contains the virtual address of base of the page table)

Control Region Length Register
(contains the virtual address of the first entry in the page table for virtual page number $2^{22}-N$, where N is the number of page table entries.)

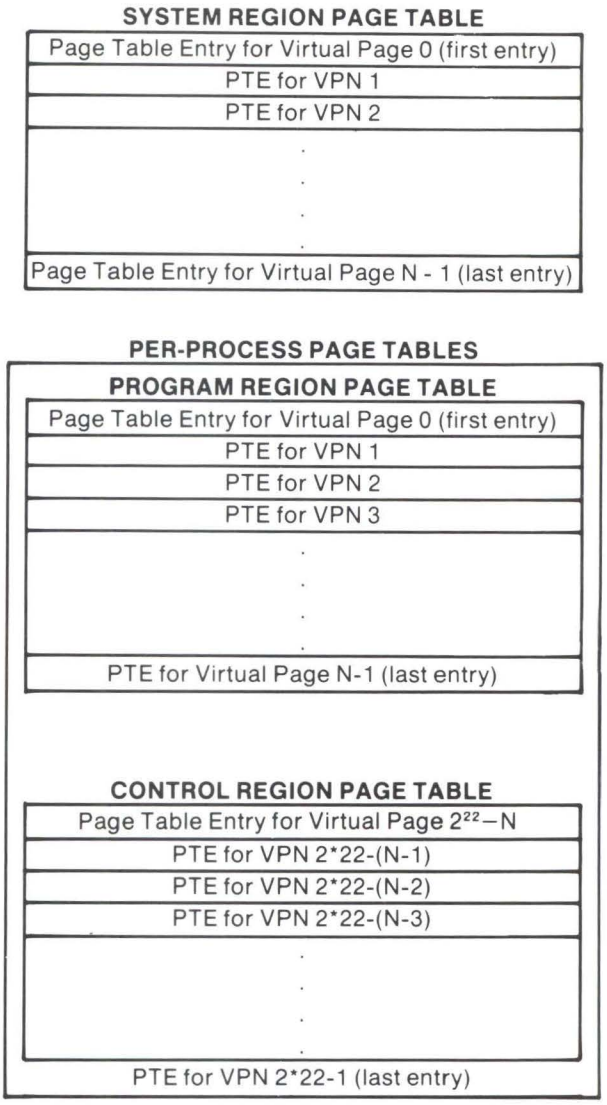


Figure 4-8
Page Tables

I/O Space and I/O Processing

An I/O device controller has a set of control/status and data registers. The registers are assigned addresses in physical address space, and their physical addresses are mapped, and thus protected, by the operating system's memory management software. That portion of physical address space in which device controller registers are located is called I/O space.

No special processor instructions are needed to reference I/O space. The registers are simply treated as locations containing integer data. An I/O device driver issues commands to the peripheral controller by writing to the controller's registers as if they were physical memory locations. The software reads the registers to obtain the controller status. The driver controls interrupt enabling and disabling on the set of controllers for which it is responsible. When interrupts are enabled, an interrupt occurs when the controller requests it. The processor accepts the interrupt request and executes the driver's interrupt service routine if it is not currently executing on a higher-priority interrupt level.

Process Context

For each process eligible to execute, the operating system creates a data structure called the **software process control block**. Within the software process control block is a pointer to a data structure called the **hardware process control block**. The hardware process control block is illustrated in Figure 4-10. It contains the hardware process context, that is, all the data needed to load the processor's

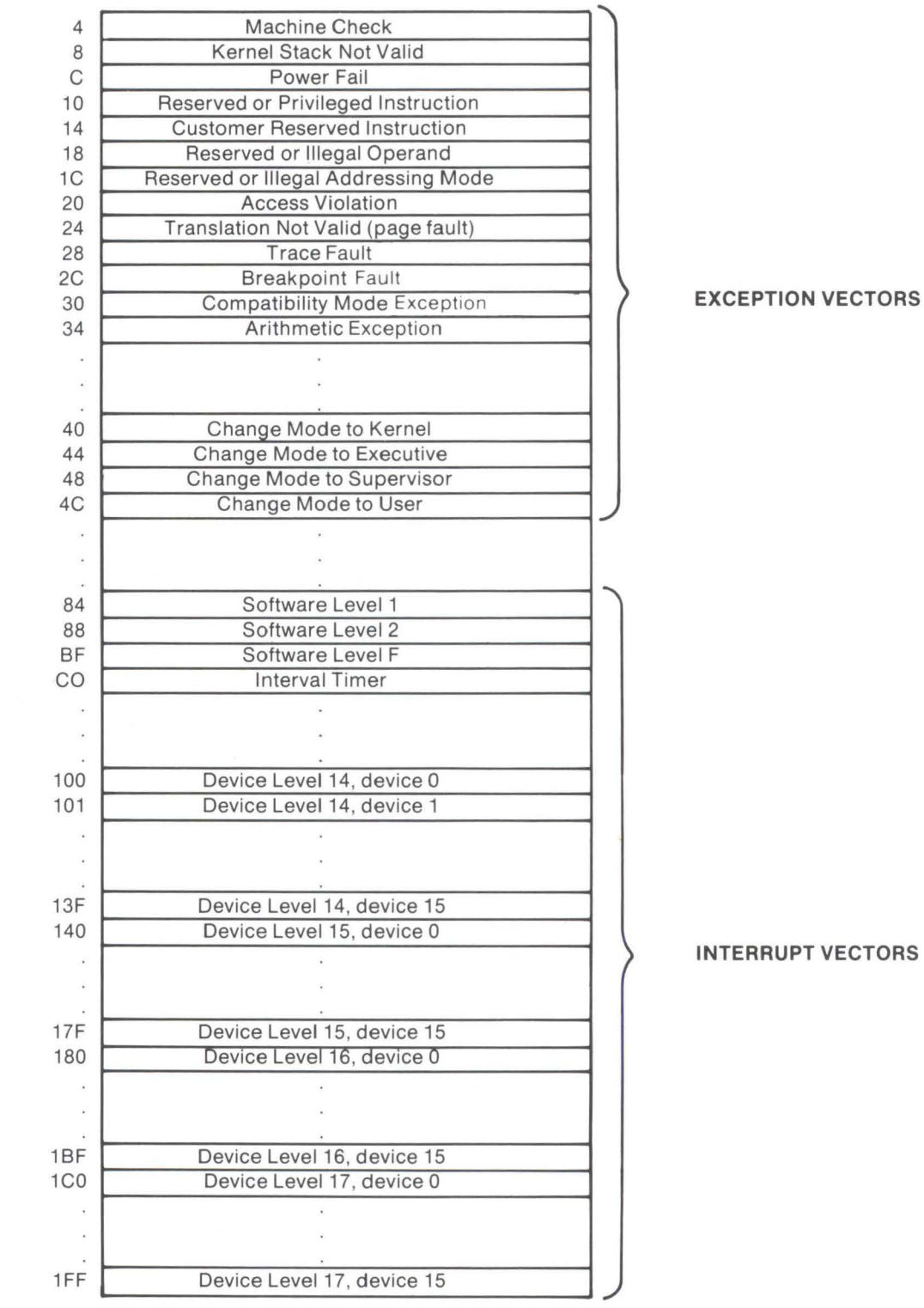


Figure 4-9

System Control Block

Table 4-4
Interrupt Priority Levels

PRIORITY Hex	PRIORITY Decimal	HARDWARE EVENT
1F	31	Machine Check, Kernel Stack Not Valid
1E	30	Power Fail
1D	29	Processor, Memory, or Bus Error
1C	28	
1B	27	
1A	26	
19	25	
18	24	Clock
17	23	UNIBUS BR7
16	22	UNIBUS BR6
15	21	UNIBUS BR6
14	20	UNIBUS BR4
13	19	Device Interrupt
12	18	
11	17	
10	16	
PRIORITY		**SOFTWARE EVENT**
0F	15	Reserved for DIGITAL
0E	14	
0D	13	
0C	12	
0B	11	Device Drivers
0A	10	
09	09	
08	08	
07	07	Timer Process
06	06	Queue Asynchronous System Trap (AST)
05	05	Reserved for DIGITAL
04	04	I/O Post
03	03	Process Scheduler
02	02	AST Delivery
01	01	Reserved for DIGITAL
00	00	User Process Level

process-specific registers when a context switch occurs. To give control of the processor to a process, the operating system loads the processor's Process Control Block Base register with the physical address of a hardware process control block and issues the Load Process Context instruction. The processor loads the process context in one operation and is ready to execute code within that context.

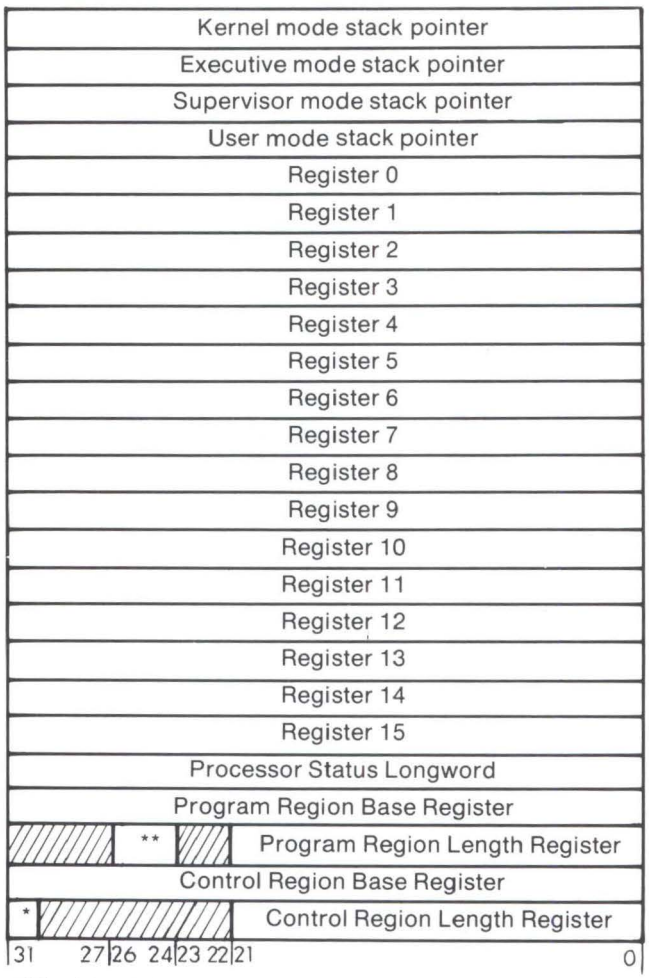
As can be seen from the illustration, a process control block not only contains the state of the programmable registers, it also contains the definition of the process virtual address space. Thus, the mapping of the process is automatically context-switched.

Furthermore, the process control block provides the mechanism for triggering asynchronous system traps to user processes. The Asynchronous System Trap field enables the processor to schedule a software interrupt to initiate an AST routine and ensure that they are delivered to the proper access mode for the process.

CONSOLE

The console is the operator's interface to the central processor. Using the console terminal, the operator can examine and deposit data in memory locations or the processor registers, halt the processor, step through instruction streams, and boot the operating system.

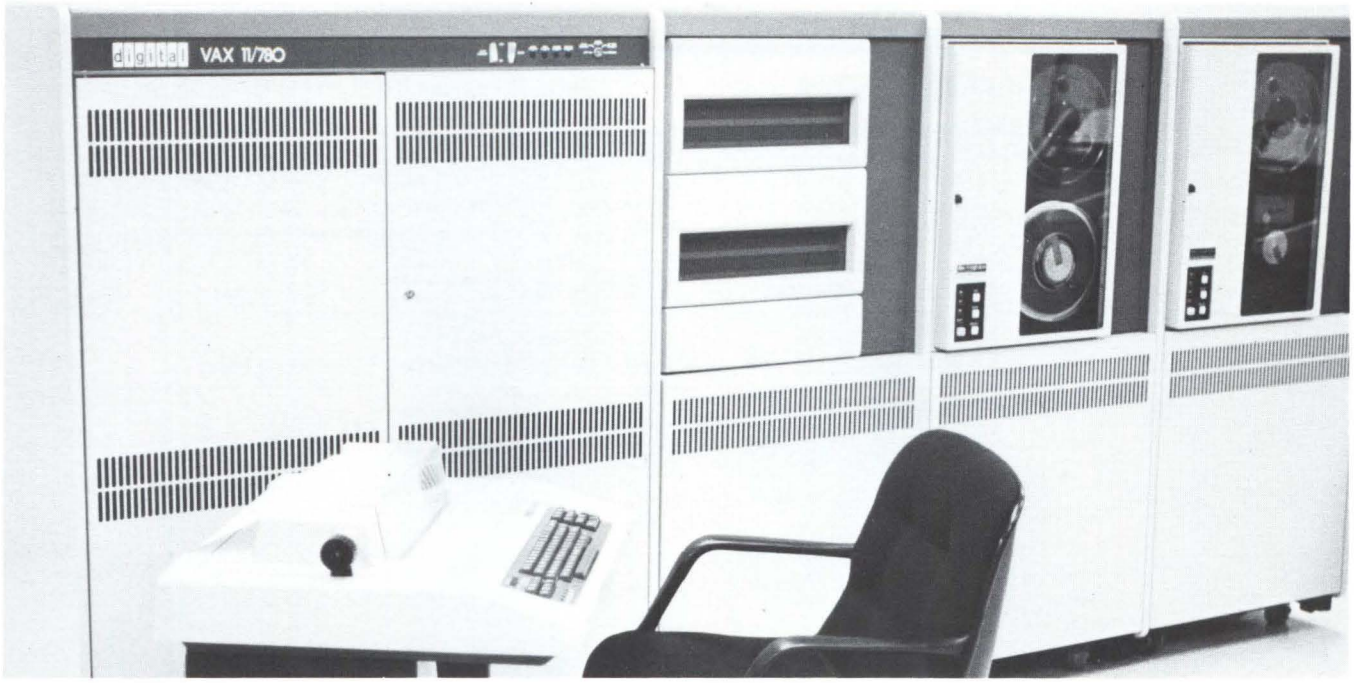
HARDWARE PROCESS CONTROL BLOCK



*Enable performance monitor
**Asynchronous System Trap pending

Figure 4-10

Hardware Process Control Block



THE VAX-11/780 PROCESSOR

INTRODUCTION

A VAX-11 processor is a specific set of hardware logic that performs the operations of the computer system according to the VAX-11 architecture.

This section describes the implementation-specific details of the VAX-11/780 processor. Its integrated components are:

- The Central Processing Unit (CPU) itself, including its cache, writable diagnostic control store, optional floating point accelerator, clocks and console.
- Main memory and main memory controllers.
- Input/output bus adaptors.
- Optional multiport memory.
- Optional high performance 32-bit interface.

These components communicate over a high-speed internal bus called the memory interconnect. Figure 4-11 illustrates the major processor components.

The Central Processing Unit performs the logical and arithmetic operations requested of the computer system. Its user programmable registers include sixteen 32-bit general purpose registers for data manipulation, and the Processor Status Longword for controlling the execution states of the CPU. The processor's instruction set is interpreted by the microcode contained in its control store.

The processor includes 12K bytes of writable diagnostic control store for updating the instruction set microcode. The writable diagnostic control is also used for storing microcode diagnostics, which can be loaded from the console's floppy disk.

The processor will also support 12K bytes of user writable control store (WCS). WCS is optionally available to the customer for augmenting the speed and power of the basic machine with customized functions.

The console enables the computer system operator to control the processor operation directly. The console actually consists of an LSI-11 microcomputer with 24K bytes of memory, a floppy disk system, and a terminal. A serial line interface is optionally available for remote diagnosis.

Two memory controllers can be be connected to the memory interconnect. Each controller handles up to 4096K bytes of semiconductor memory, for a system total of 8192K bytes of memory. The memory controllers employ an error detecting and correcting technique that ensures correction of all single-bit errors and detection of all double-bit errors.

In addition, VAX-11/780 supports the multiported memory option. Multiported memory supports very high throughput interprocessor communications. Multiported memory is discussed more fully in the Peripherals section.

Three I/O bus adaptors can be interfaced to the memory interconnect: an adaptor for the MASSBUS, which connects high-speed disk and magnetic tape devices to the processor; an adaptor for the UNIBUS, which connects lower-speed devices to the processor, including disks, communications lines, and I/O peripherals such as terminals, line printers, and card readers; and an optional adaptor for the high performance 32-bit interface. The high per-

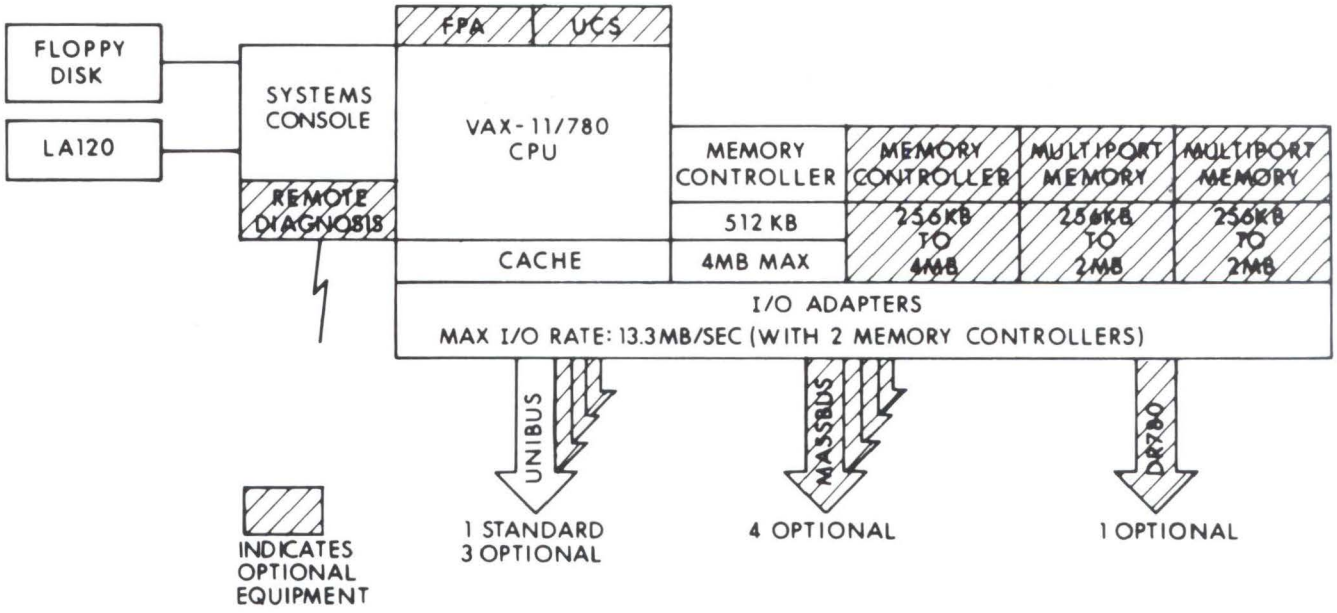


Figure 4-11
VAX-11/780 Processor

formance interface enables the user to interface custom devices directly to the memory interconnect, or to connect two VAX-11/780 systems together. The high performance interface will be discussed more thoroughly in the Peripherals section.

VAX-11/780 PROCESSOR COMPONENTS

Described below are the major hardware components of the VAX-11/780 processor.

VAX-11/780 Console

The VAX-11/780's integrated console consists of an LSI-11 microcomputer with 16K bytes of read/write memory and 8K bytes of ROM (used to store the LSI diagnostic, the LSI bootstrap, and fundemental console routines), a floppy disk system (for the storage of basic diagnostic programs and software updates), a hard-copy terminal, and an optional remote diagnosis port.

The console is further used for updating the software with maintenance releases and for loading optional software products distributed on floppy disk.

The operator communicates with the VAX-11/780 console via a set of user-oriented, English-like commands known as the console command language (CCL).

An EIA serial line interface and modem can be added to the console to provide remote diagnosis.

VAX-11/780 MEMORY INTERCONNECT

The memory interconnect is the system's internal bus, conveying addresses, data, and control information between the processor and memory, and between memory and the I/O controllers. The memory interconnect has a cycle time of 200 nanoseconds and can transfer 32 bits each cycle. Data transfers use two consecutive cycles to transfer 64 bits at a time. The maximum memory interconnect transfer rate is 13.3 million bytes per second. The memory interconnect provides an unusual degree of throughput and reliability because it uses:

- time-division multiplexing
- distributed priority arbitration
- parity and protocol checking on every transfer
- transaction history recording

The protocol, or sequence in which operations occur on the memory interconnect, is time-division multiplexed to increase the effective bus bandwidth. Time-division multiplexing means that the transactions which constitute one transfer operation are interleaved with the transactions which constitute another transfer operation. Thus, several operations can be in progress over the same period of time. For example, the CPU can ask a memory controller to read some data; the same memory controller might then transfer previously requested data to an I/O device before it transfers the requested data to the CPU.

In some systems, the processor bus can be tied up for each transfer because a requester acquires the bus to send an address and then keeps the bus while it waits for the requested data. In VAX-11/780, the bus is not held inactive during the data access time because bus ownership is relinquished after every cycle. A requester acquires the bus to specify an operation and send an address, and then relinquishes the bus. At some time later the responder acquires the bus to send back the requested data. In the interim, any number of other transactions can be initiated or completed. This and the fact that transactions are buffered

make it possible for the bus to operate at its full bandwidth.

Arbitration on the memory interconnect is distributed, which ensures that no unit is critical to bus operation. Every unit on the memory interconnect has its own arbitration line. Arbitration lines are ordered by priority and every unit monitors all the arbitration lines each cycle to determine if it will get the next cycle. Unlike some bus systems, any unit on the memory interconnect (except the CPU clock) can fail without causing a failure of the entire bus.

To ensure the integrity of the signals transmitted, the memory interconnect includes several error checking and diagnostic mechanisms, such as:

- parity checking on data, addresses, and commands
- protocol checking in each interface
- a history silo of the last 16 memory interconnect cycles

VAX-11/780 MAIN MEMORY AND CACHE SYSTEMS

The processor includes both main memory systems and cache memory systems. Transactions between main memory and the processor take place over the memory interconnect. The cache memory systems are internal to the processor.

Main Memory

Main memory consists of arrays of MOS RAM integrated circuits with a cycle time of 600 nanoseconds. A memory controller can access a maximum of 4,194,304 bytes (4M bytes). Two memory controllers can be connected to the memory interconnect, yielding a maximum of 8M bytes of physical memory that can be available on the system. The maximum total physical address space is 2^{29} or approximately 512 million bytes. However, the minimum required memory is 256K bytes, which is then expandable in increments of 256K bytes.

A memory controller will buffer one command while it processes another to increase system throughput. Main memory can also be interleaved (where two memory controllers are each addressing the same amount of memory) to increase the available memory bandwidth. The memory system employs error checking and correction (ECC) that corrects all single bit errors and detects all double bit errors.

When the system is powered down, an ac standby current is normally used to retain the contents of memory. In case of temporary AC power interruption, an optional backup battery is also available to provide 10 minutes of power for up to 4M bytes of memory so that the contents of main memory are not destroyed. Two backup batteries provide power for up to 8M bytes of memory.

Data are fetched from main memory 64 bits at a time (two memory interconnect cycles) and cached in the processor's internal memory systems. The internal memory systems include a main memory cache, an address translation buffer, and an instruction lookahead buffer.

Memory Cache

The memory cache is the primary cache system for all data coming from memory, including addresses, address translations, and instructions. The memory cache is an 8K byte, two-way set associative, write-through cache.

Write-through provides reliability because the contents of main memory are updated immediately after the processor performs a write. Most write-through cache systems tie up the processor while main memory is updated. However, the VAX-11/780 processor buffers data to be written to memory to avoid waiting while main memory is updated from the cache. Therefore, while providing the reliability of a write-through cache, this system also provides much the same performance as a write-back cache.

Memory cache significantly reduces processor wait time since practically all of the time, (greater than 95%), the data are in the cache. The cache memory system carries byte parity for both data and addresses for increased integrity.

Address Translation Buffer

The address translation buffer is a cache of virtual to physical address translations. It significantly reduces the amount of time spent by the CPU on the repetitive task of dynamic address translation. The cache contains 128 virtual-to-physical page address translations which are divided into equal sections: 64 system space page translations and 64 process space page translations. Each of these sections is two-way associative. There is byte parity on each entry for increased integrity.

Instruction Buffer

The 8-byte instruction buffer improves CPU performance by prefetching data in the instruction stream. The control logic continuously fetches information from memory or cache, where possible, to keep the 8-byte buffer full. It effectively eliminates the time spent by the CPU waiting for two memory cycles where bytes of the instruction stream cross 32-bit longword boundaries. In addition, the instruction buffer processes operand specifiers in advance of execution and subsequently routes them to the CPU.

I/O CONTROLLER INTERFACES

Peripherals can be connected to the processor's memory interconnect bus in either of two ways: through the MASSBUS, for high-speed disk and/or magnetic tape devices, or through the UNIBUS, for a variety of I/O devices, including line printers, disks, card readers, terminals, and interprocessor communication links.

VAX-11 MASSBUS Interface

The processor interface for a MASSBUS peripheral is the **MASSBUS adaptor**. The MASSBUS adaptor performs control, arbitration, and buffering functions. Up to four MASSBUS adaptors can be connected to the memory interconnect. The MASSBUS is typically used to attach high-speed disk or magnetic tape devices.

Each MASSBUS adaptor includes its own address translation map that permits scatter/gather disk transfers. In scatter/gather transfers, physically contiguous disk blocks can be read into or written from discontiguous blocks of memory. The translation map contains the addresses of the pages, which may be scattered throughout memory, from or to which the contiguous disk transfer takes place.

Each MASSBUS adaptor includes a 32-byte silo data buffer. Data are assembled in 64-bit quadwords (plus parity) to make efficient use of the memory interconnect bandwidth. On transfers from memory to a MASSBUS peripheral, the

MASSBUS adaptor anticipates upcoming MASSBUS data transfers by fetching the next 64 bits from memory before all of the previous data have been transferred to the peripheral.

On-line diagnostics and built-in loop-back testing enable fault isolation of the MASSBUS adaptor for any of its function circuits without a drive on the MASSBUS.

VAX-11/780 UNIBUS Interface

All devices other than the high-speed disk drives and magnetic tape transports are connected to the UNIBUS, an asynchronous bidirectional bus. These include all DIGITAL- and user-developed real-time peripherals. The UNIBUS is connected to the memory interconnect through the **UNIBUS adaptor**. The UNIBUS adaptor does priority arbitration among devices on the UNIBUS. Up to four UNIBUS adapters can be placed on the memory interconnect.

The UNIBUS adaptor provides access from the VAX-11/780 processor to the UNIBUS peripheral device registers by translating UNIBUS addresses, data transfer requests, and interrupt requests to their memory interconnect equivalents, and vice versa. The UNIBUS adaptor address translation map translates an 18-bit UNIBUS address to a 30-bit memory interconnect address. The map provides direct access to system memory for nonprocessor request UNIBUS peripheral devices and permits scatter/gather disk transfers.

The UNIBUS adaptor enables the processor to read and/or write the peripheral controller registers. In some cases this constitutes the transfer.

To make the most efficient use of the memory interconnect bandwidth, the UNIBUS adaptor provides buffered direct memory access data paths for up to 15 nonprocessor request (NPR) devices. Each of these channels has a 64-bit buffer (plus byte parity) for holding four 16-bit transfers to and from UNIBUS devices. The result is that only one memory interconnect transfer (64 bits) is required for every four UNIBUS transfers. The maximum aggregate transfer rate through the buffered data paths is 1.35 million bytes per second. On memory interconnect-to-UNIBUS transfers, the UNIBUS adaptor anticipates upcoming UNIBUS requests by pre-fetching the next 64-bit quadword from memory as the last 16-bit word is transferred from the buffer to the UNIBUS. By the time the UNIBUS device requests the next word, the UNIBUS adaptor has it ready to transfer.

Any number of unbuffered direct memory access transfers are handled by one Direct Data Path. Every 8- or 16-bit transfer requires one 32-bit transfer on the memory interconnect. The maximum transfer rate through the Direct Data Path is 500,000 bytes per second.

The UNIBUS adaptor permits concurrent program interrupt, unbuffered, and buffered data transfers. The aggregate throughput rate of the Direct Data Path, plus the 15 buffered data paths, is 1.35 million bytes per second.

Data Throughput

VAX-11/780 includes many features that support high data throughput, including silo data buffers for MASSBUS peripheral controllers, buffered direct memory access for the UNIBUS peripherals, and 64-bit data transfers and prefetching.

Memory bandwidth matches that of the processor's internal bus — 13.33 million bytes per second, including time for refresh cycles. This is primarily because of the memory controller request buffers, which substantially increase memory throughput and overall system throughput, and decrease the need for interleaving for most configurations. Memory interleaving, which is enabled and disabled under program control, can be used effectively when more than two MASSBUS peripheral controllers are connected and the MASSBUS and UNIBUS devices are transferring at very high rates — greater than one million bytes per second.

The operating system supports the hardware throughput in its I/O request processing software. The software uses the processor's multiple hardware priority levels to increase I/O response time, and keeps each disk controller as busy as possible by overlapping seek requests with I/O transfers.

VAX-11/780 FLOATING POINT ACCELERATOR

The floating point accelerator (FPA) is an optional high-speed processor enhancement. When included in the processor, the floating point accelerator accelerates the execution of the addition, subtraction, multiplication, and division instructions that operate on single- and double-precision floating point operands. This includes the special EMOD and POLY instructions in both single- and double-precision formats. Additionally, the floating point accelerator enhances the performance of the 32-bit integer multiply instruction _MUL.

The processor does not have to include the floating point accelerator to execute floating point operand instructions; the FPA increases the execution speed of floating point instructions. The floating point accelerator can be added or removed without changing any existing software.

When the floating point accelerator is included in the processor, a floating point register-to-register add instruction takes as little as 800 nanoseconds to execute. A register-to-register multiply instruction takes as little as one microsecond. The inner loop of the POLY instruction takes approximately one microsecond per degree of polynomial.

THE VAX-11/750 PROCESSOR

INTRODUCTION

A VAX-11 processor is a specific set of hardware logic that performs the operations requested of the computer system according to the VAX-11 architecture.

This section describes the implementation specific details of the VAX-11/750 processor. Its integrated components are:

- the Central Processing Unit (CPU) itself, including its cache, optional user control store, clocks and console
- main memory and main memory controllers
- peripheral bus adaptors

Figure 4-12 illustrates the major VAX-11/750 processor components.

The central processing unit performs the logical and arithmetic operations requested of the computer system. Its user programmable registers include sixteen 32-bit general purpose registers for data manipulation, and the Processor Status Word for controlling the execution states of the CPU. The processors instruction set is defined by the microcode contained in its control store.

The optional User Control Store includes 10K bytes (1Kbytes of 80 bit microwords) of writeable storage. This allows customers to augment the speed and power of the basic machine with customized microcode functions. Digital offers a loadable microcode package for extended precision floating point arithmetic operations (G- and H-floating point data types) on the 11/750.

The console enables the computer system operator to control the processor operation directly. The console subsystem consists of the console terminal (LA38 DECwriter), the front panel, the user oriented console command language, and a TU58 Tape Cartridge Drive. Also optionally available for the console is the remote diagnosis interface.

The main memory subsystem consists of ECC MOS memory, which is interfaced to the system via the memory controller. MOS memory may be added to the system in increments of 256K bytes to a maximum of 2M bytes.

The I/O subsystem consists of the UNIBUS and MASSBUS devices connected to the system via special buffered interfaces called adaptors. Each VAX-11/750 system contains one UNIBUS adapter for standard peripherals and up to a maximum of three MASSBUS adapters for high speed peripherals.

VAX-11/750 PROCESSOR COMPONENTS

Described below are the major hardware components of the VAX-11/750 processor.

VAX-11/750 Console

The console enables the computer system operator to control the processor operation directly. The console subsystem consists of the console terminal (LA38), the front panel, the user oriented console command language, and a TU58 Tape Cartridge Drive. Simple console commands, entered through the console terminal, replace the traditional toggle switches and provide operational control (i.e., bootstrapping, initialization, self testing, examining and depositing data in memory,etc.). When not performing op-

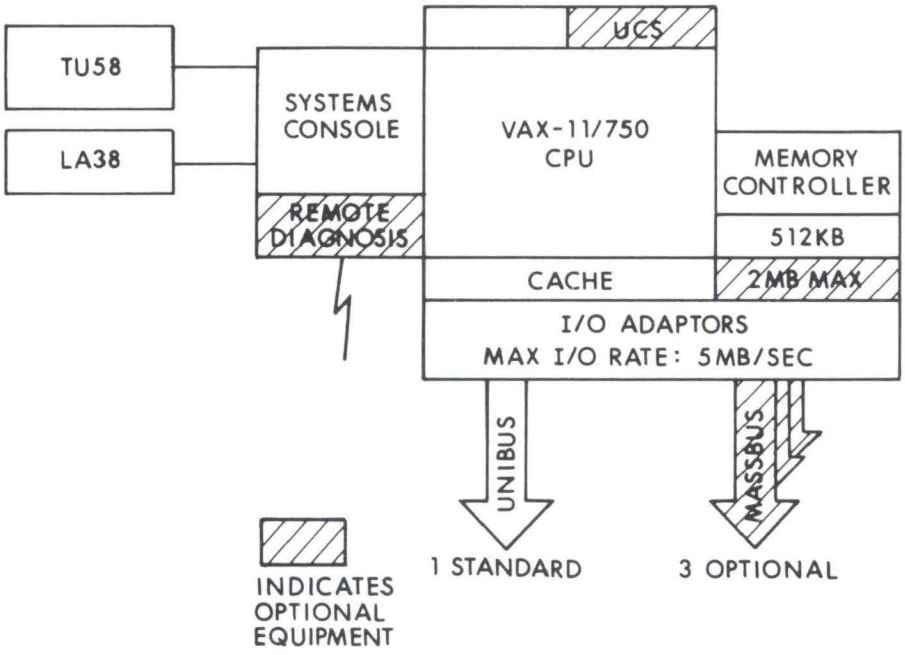


Figure 4-12
VAX-11/750 Processor

erator functions or error logging, the same terminal can be available to authorized users for normal system operations.

The VAX-11/750 console subsystem and the console command language also facilitate the loading of diagnostics and software updates from the TU58 Tape Cartridge. For those customers subscribing to a DIGITAL maintenance contract, the console subsystem may also be equipped with a remote diagnosis module (RDM) allowing the VAX-11/750 to interface to a host computer at a DIGITAL Diagnostic Center for remote fault detection or preventive maintenance procedures.

VAX-11/750 Main Memory

The VAX-11/750 main memory is built using 16K MOS RAM (random access memory) LSI chips. Physical memory is organized into an array of 32-bit longwords plus an additional 7 bits per longword dedicated to ECC (error correcting code). ECC allows the correction of all single-bit errors and the detection of all double bit errors to insure data integrity. Main memory is interfaced to the VAX system via the memory controller. The VAX-11/750 can be easily field upgraded to 2M bytes of main memory by simply adding 256K byte expansion modules.

VAX-11/750 Cache Systems

The VAX-11/750 CPU provides three cache systems: the main memory cache, the address translation buffer, and the instruction buffer.

- Main Memory Cache

Memory cache (typically 90% hit rate) provides the central processor with high-speed data access by storing frequently referenced addresses, data and instruction items. The memory cache typically reduces memory access time in half.

The VAX-11/750 memory cache is a 4K byte, direct mapped, write-through cache. It is used for all data coming from memory, including addresses and instructions. The write-through feature protects the integrity of memory because memory contents are updated immediately after the processor performs a write. For increased integrity, the cache memory system carries byte parity for both data and addresses. Cache locations are allocated when data is read from memory or when a longword is written to memory. Memory cache also watches I/O transfers and updates itself appropriately. Therefore, no operating system overhead is needed to synchronize the cache with I/O operations, i.e., memory cache is transparent to all software.

- Instruction Buffer

The instruction buffer is an 8 byte buffer that enables the CPU to fetch and decode the next instruction while the current instruction completes execution. The instruction buffer in combination with the parallel data paths (which can perform integer arithmetic and shifting operations simultaneously) significantly enhances the VAX-11/750's performance because the CPU is not held in a wait state.

- Address Translation Buffer

The address translation buffer is a cache of the most frequently used 512 physical address translations. It significantly reduces the amount of time the CPU spends on the repetitive task of dynamic address translation. The cache contains 512 virtual-to-physical page address translations which are divided into equal sections: 256 system space page translations and 256 process space page translations. Each of these sections is two-way associative and has parity on each entry for increased integrity.

Peripheral Controller Interfaces

Peripherals can be connected to the processor in either of two ways: through the MASSBUS, which conveys signals

to and from high-speed disks or magnetic tape devices, or through the UNIBUS, which conveys signals to and from a variety of I/O devices, including line printers, disks, card readers, tapes, terminals, and interprocessor communication links.

VAX-11 MASSBUS Interface

The processor interface for a MASSBUS peripheral is the **MASSBUS adaptor**. The MASSBUS adaptor performs control, arbitration, and buffering functions. There may be a total of three MASSBUS adapters on each VAX-11/750 system.

Each 11/750 MASSBUS adaptor includes its own address translation map that permits scatter/gather disk transfers. In scatter/gather transfers, physically contiguous disk blocks can be read into or written from discontiguous blocks of memory. The translation map contains the addresses of the pages, which may be scattered throughout memory, from or to which the contiguous disk transfer takes place.

Each 11/750 MASSBUS adaptor includes a 32-byte silo data buffer. Data are assembled in 32-bit longwords (plus parity) to make efficient use of the system bus. On transfers from memory to a MASSBUS peripheral, the MASSBUS adaptor anticipates upcoming MASSBUS data transfers by fetching the next 32 bits from memory before all of the previous data are transferred to the peripheral.

On-line diagnostics and loop-back enable adaptor fault isolation without requiring the use of a drive on the MASSBUS.

VAX-11/750 UNIBUS Interface

General purpose peripherals and customer developed devices are connected to the VAX-11/750 system via the UNIBUS. Since the 11/750 memory deals in 24-bit physical addresses (16M byte physical address space), 18-bit UNIBUS addresses must be translated to 24 bit memory addresses. This mapping function is performed by the UNIBUS adapter (a special hardware interface between memory and the UNIBUS) which translates UNIBUS addresses to their memory equivalents, and vice versa.

The UNIBUS adapter performs priority arbitration among devices on the UNIBUS, a function handled by the central processor in PDP-11 systems. The address translation map permits contiguous disk transfers to and from non-contiguous pages of memory (these are called scatter/gather operations). Interrupts on the VAX-11/750 UNIBUS are directly vectored into the appropriate process handler.

The UNIBUS adapter allows two kinds of data transfers; program interrupt and direct memory access (DMA). To make the most efficient use of the memory bandwidth, the UNIBUS adapter facilitates high-speed DMA transfers by providing buffered DMA data paths for up to 3 high-speed devices at one time. Each of these channels has a 32-bit buffer (plus byte parity) for holding two 16-bit transfers to or from UNIBUS devices. The result is that only one memory transfer (32 bits) is required for every two UNIBUS transfers. The maximum aggregate transfer rate through the buffered data paths is 1.5M bytes per second.

Any number of unbuffered DMA transfers are handled by one direct DMA data path. Every 8- or 16-bit transfer on the UNIBUS requires a 32-bit memory transfer (although only 16 bits are used). The maximum transfer rate through the direct data path is 1M bytes per second.

It should be noted that the UNIBUS adapter permits program interrupts, unbuffered and buffered data transfers to occur concurrently.

5
The Peripherals

The VAX system supports high-performance mass storage devices for on-line data retrieval, unit record equipment for data processing, terminals and line interfaces for the interactive user, direct memory access interfaces for real-time users and a line interface for interprocessor communications.

The mass storage systems provide large capacity and high throughput. Each MASSBUS adapter can support up to eight disk drives or seven disk drives and one magnetic tape controller. In addition, up to eight medium-capacity disk drives can be connected to the system's UNIBUS. VAX/VMS overlaps seeks on all multiple-drive disk configurations, performs multiple-block I/O transfers, and allows the user to control buffering, positioning, and blocking.

Card readers and line printers can be spooled input and output devices managed by operator-controlled queues. The LP11 and LA11 series line printers provide a range of high-speed and low-cost printer models. Up to four LP11 printers and up to 16 LA11 printers can be used on the system.

The system supports full-duplex handling for both hard copy and video terminals. The LA120 is a hard-copy terminal which offers moderate throughput and advanced print features; the VT100 video terminal offers a variety of controllable character and screen attributes including 24 lines by 80 columns or 14 lines by 132 columns screen sizes, smooth scrolling, and split screen. The system can support up to 96 terminals.

The DMC11 serial synchronous communications line provides high-performance point-to-point interprocessor connection using the DIGITAL Data Communications Message Protocol (DDCMP). The DMC11 ensures reliable data transmission and relieves the host processor of the details of protocol operation. For very high-performance interprocessor communications, the VAX-11/780 offers both multiport memory (MA780) and a high-speed channel interface (DR780). The DR780 can also be used for interfacing customer devices which require transfer rates of up to 6.67M bytes/second.

For real-time applications, VAX supports the LPA11-K and DR11-B direct memory access (DMA) interfaces. These devices reduce CPU involvement in I/O operations and speed the transfer of data between external devices and computer memory. The LPA11-K is an intelligent (dual-microprocessor) controller which provides high speed data sampling, operates in both dedicated and multirequest mode, and supports a number of peripheral devices. The DR11-B is a general purpose interface which performs high speed block data transfers between the VAX memory and user peripheral devices.

All equipment is integrated with the software system, and is supported by both on-line error logging and diagnostics. Each component includes extensive error checking and correction features. The software provides power failure and error recovery algorithms.

COMPONENTS

VAX supports four types of peripheral subsystems:

- Mass storage peripherals such as disk and magnetic tape
- Unit record peripherals such as line printers and card readers
- Terminals and terminal line interfaces
- Interprocessor communications links

All peripheral device control/status registers (CSRs) are assigned addresses in physical I/O space. No special processor instructions are needed for I/O control. In addition, all device interrupt lines are associated with locations that identify each device's interrupt service routine. When the processor is interrupted on function request completion, it immediately starts executing the appropriate interrupt service routine. There is no need to poll devices to determine which device needs service.

Devices use either one of two types of data transfer techniques: direct memory access or programmed interrupt request. The mass storage disk and magnetic tape devices and the interprocessor communications link are capable of direct memory access (DMA) data transfers. The DMA devices are also called **non-processor request** (NPR) devices because they can transfer large blocks of data to or from memory without processor intervention until the entire block is transferred.

The unit record peripherals and terminal interfaces are called **programmed interrupt request** devices. These devices transfer one or two bytes at a time to or from assigned locations in physical address space. Software then transfers the data to or from a buffer in physical memory.

MASS STORAGE PERIPHERALS

The mass storage peripherals include various capacity moving head disk drives and various speed magnetic tape transports:

- the high speed, large capacity RP06 and RM05 disk drives
- the high speed, medium capacity RM03 disk drive
- the medium speed, smaller capacity RL02, and RK07 disk drives
- the RX02 floppy disk
- the TE16, TU45, and TU77 magnetic tape transports
- the TS11 magnetic tape transport

The RM03, RP06, and RM05 disks and the TE16, TU45, and TU77 magnetic tape controllers are MASSBUS peripheral devices. The RX02 floppy disk, the RL02, and RK07 disk drives, and the TS11 tape subsystem are UNIBUS peripheral devices. Each MASSBUS can support up to eight device controllers; eight disk controllers with one drive each or seven disk drives and one magnetic tape formatter with up to eight tape transports.

To support the performance and reliability features of the system's disk and magnetic tape devices, the operating system's disk and magnetic tape device drivers provide:

- overlapped seeks for increased throughput on controllers with multiple disk drives
- overlapped magtape operations (write on one transport while another rewinds, for example)
- multiple block non-contiguous I/O transfers for file-structured devices
- read and write checks on a per-request, per-file, and/or volume basis
- extensive error recovery algorithms (e.g., ECC and offset recovery for disk, NRZI error correction for magnetic tape)
- logging of all device errors
- dynamic bad block support for file-structured disk devices
- volume mount verification after a change in drive status (off/on-line, powerfail)
- powerfail recovery for on-line drives, including repositioning of magnetic tape transports

Table 5-1
Disk Devices

DISK	RX02	RL02	RK07	RM03	RP06	RM05
Pack capacity:	512 Kbytes	10.4 Mbytes	28 Mbytes	67 Mbytes	176 Mbytes	300 Mbytes
Peak transfer rate (/sec):	55 Kbytes	512 Kbytes	538 Kbytes	1200 Kbytes	806 Kbytes	1200 Kbytes
Ave. seek time:	263ms	55ms	36.5ms	30ms	30ms	30ms
Ave. rotational latency:	83ms	12.5ms	12.5ms	8.3ms	6ms	8.3ms

For applications requiring special data reliability checks, the programmer can implement user written error recovery procedures without having to write unique device driver routines. The operating system driver's normal error recovery retry and error logging operations can be inhibited. If any error occurs when the recovery functions are inhibited, the driver immediately terminates the I/O operation and returns a failure status. User software can then perform its own recovery or logging procedures, since all the hardware diagnostic operations are available to jobs granted the diagnostic privilege by the system manager.

Disks

The disk subsystems provide high performance and reliability. They feature accurate servo positioning, error correction, and offset positioning recovery. Table 5-1 summarizes the capacities and speeds of the disk devices.

All disk drives use top-loading removable media. The RM03, RP06, and RM05 disk drives can be mixed on the same MASSBUS.

The UNIBUS accepts RK06, RK07, and RL02 disk drives and the RX02 floppy disk. The RX02 is the smallest capacity disk available, while the RK07 is the largest capacity disk. Up to eight RX02, RL02, RK06, and RK07 disk drives can be mixed in any combination on the same controller. In small system configurations where the RK06 or RK07 is used as the systems device, two drives are required in the configuration.

To decrease the effective access time and increase throughput, the operating system's disk device drivers provide overlapped seeks for all disk units on a controller. All I/O transfers, including write checks, are preceded by a seek, except when the seek is explicitly inhibited by diagnostic software. On MASSBUS devices, seeks to any unit can be initiated at any time and do not require controller intervention. During seeks, the controller is free to perform a transfer on any unit other than the one on which the seek is active. If a data transfer was in progress at the time of completion, the driver processes the attention interrupts caused by seek completion when the controller is free.

The device unit notifies the driver when it detects a read error that can be recovered using its error correction code (ECC). It provides the position and pattern of any error burst of up to 11 bits within the data field. The driver applies the error correction to the data in memory. The transfer continues as if the error had not occurred.

In addition to overlapping seeks with data transfers, the driver also overlaps offset error recovery with normal controller operation. Offset recovery enables the driver to reposition the head on the track to pick up a stronger signal on a sector during a read operation. Provided that retry is not inhibited, the driver performs offset recovery automatically when a read error occurs that can not be corrected using the hardware ECC.

The driver logs all errors, including those from which it successfully recovers. The driver also supplies dynamic bad block handling for virtual I/O (Files-11 file-structured) operations. When a bad block is detected, the information is stored in the file header. The bad block is recorded in the bad block file when the file is deleted.

In addition to the driver's dynamic bad block handling, the system includes an on-line static bad block utility and on-line diagnostics for verifying drive level functions.

Magnetic Tape

The TE16, TU45, and TU77 are high performance MASSBUS tape storage subsystems, which share the following characteristics:

- program-selectable 1600 or 800 bpi, 9-track data storage
- industry compatible data formats
- reading in reverse (as well as forward)
- parity, longitudinal, and cyclic redundancy checking
- NRZI error correction

The TU45 and TE16 are identical in capacity: both allow up to 40 million bytes per tape reel and both allow up to 8 tape drives per formatter. However, the TU45 offers substantially higher speed and throughput. Read/write speeds for the TU45 and TE16 are 75 and 45 inches/second respectively; their data transfer speeds are 120K and 72K bytes per second.

The TU77 tape storage system can perform read/writes of data at the rate of 125 inches/second, while allowing a peak transfer rate of 200K bytes/second. These features make the TU77 ideally suited for heavy duty cycle applications such as disk to tape backup and transaction processing.

The TS11 is a medium performance UNIBUS magnetic tape subsystem containing 9 tracks with a density of 1600 bpi. It performs read/writes at a speed of 45 inches/second. The TS11 subsystem can handle a maximum data transfer rate of up to 72K bytes/second.

The operating system's magnetic tape device driver supports the read reverse operation, which enables a program to request a sequential read of the block preceding the block at which the tape is positioned. Writing occurs only while the tape is moving forward.

The operating system's file system can read and write file-structured magnetic tape volumes using the current ANS magnetic tape standard. The system also supports multivolume files, program-controlled blocking factors, and unlabeled magnetic tapes.

UNIT RECORD PERIPHERALS

The operating system normally treats line printers and card readers as spooled shareable devices managed by multiple operator-controlled queues. The devices can also be allocated to individual programs.

The operating system's line printer handling includes line and page counting for job accounting. The user can specify carriage control as: one line per record, FORTRAN conventions, contained within the record itself, or general pre- and post-spacing (within the limits of the hardware capabilities).

The operating system's card reader driver interprets the encoded punched information using the American National Standard 8-bit card code. The driver uses a special punch outside the data representation to indicate end-of-file.

LP11 Line Printers

LP11 series line printers can be connected to the VAX system. The LP11 series printers are impact-type, rotating drum, serial interface line printers. They feature full line buffering, a static eliminator, and a self-test capability.

All models are 132-column printers that can accept paper 4 to 16-3/4 inches wide with up to 6-part forms. They print 10 characters per inch horizontally, and 6 or 8 lines per inch vertically (switch selectable). They include a vernier adjustment for horizontal and vertical paper position. All models are available with either upper (64) or upper/lower (95) character sets (including numbers and symbols). Most models have optional scientific or EDP character sets.

The low-cost models print one line every two revolutions (300 lines per minute with the 64-character set, 230 lines per minute with the 95-character set), or one line every revolution (600 lines per minute with the 64-character set, 460 lines per minute with the 95-character set). A higher-speed version that includes a noise-reduction cabinet, ribbon guide, and a high-speed paper puller offers 900 line-per-minute printing with the 64-character set, or 660 lines per minute with the 95-character set.

For systems requiring even greater printer throughput, LP11 models are available that print up to 1200 lines per minute with the 64-character set or 800 lines per minute with the 95-character set.

LA11 Line Printer

The LA11 is an extremely low-cost, highly-reliable parallel interface printer. The LA11 prints at speeds up to 180 characters per second. The print set consists of the ASCII characters, including 95 upper and lower case letters, numbers, and symbols. Characters are printed using a 7 × 7 matrix with horizontal spacing of 10 characters per inch and vertical spacing of six lines per inch.

Adjustable pin-feed tractors allow for a variable-form width of 3 to 14-7/8 inches (up to 132 columns). A forms length switch sets the top-of-form to any of 11 common lengths, with fine adjustment for accurate forms placement. The printer can accommodate multipart forms (with or without carbons) of up to six parts.

CR11 Card Reader

CR11 card readers can be connected to the system as programmed interrupt request devices. The CR11 reads up to 285 80-column punched cards per minute. The card reader has a high tolerance for cards that have been nicked, warped, bent, or subjected to high humidity. The card reader uses a short card path, with only one card in the track at a time. It uses a vacuum pick mechanism and keeps cards from sticking together by blowing a stream of air through the bottom half-inch of cards in the input hopper. The input hopper holds up to 400 cards, and cards can be loaded and unloaded while the reader is operating.

TERMINALS AND INTERFACES

Interactive terminals can be connected to the VAX system. The operating system's terminal driver provides full duplex handling for both hard copy and video terminals.

Programs can control terminal operations through the terminal driver. The terminal driver supports many special operating modes for terminal lines. A program can enable or disable the following modes by calling a system service:

- SLAVE All unsolicited data are discarded. This mode is used to establish application-controlled terminals.
- NO ECHO Data entered on the terminal keyboard are not printed or displayed on the terminal. This mode is used, for example, to read passwords typed on the terminal.
- PASS ALL All data entered on the terminal are transmitted to the program as 8-bit binary information without any interpretation, except where a line terminator or terminators are specified. This mode enables programs to perform their own interpretation of control characters instead of using the VAX/VMS interpretation.
- ESCAPE Escape sequences entered on the terminal are recognized as read terminators, validated, and passed to a program for interpretation.
- TERMINAL/HOST SYNCHRONIZATION Data sent to the terminal are controlled by terminal-generated XOFF and XON. These functions are generated by typing CTRL/S and CTRL/Q on command terminals and are interpreted as requests to stop and resume output to the terminal.
- HOST/TERMINAL SYNCHRONIZATION All read operations are explicitly solicited with XON and terminated with XOFF. XON and XOFF are also used to keep the type-ahead buffer from filling.

Input from a command terminal is always independent of concurrent output. This capability is called *type-ahead*. Data typed at the terminal are retained in a type-ahead buffer until a program issues a read request. At that time the data are transferred to the program buffer and echoed

on the terminal (provided that echoing is not disabled). If a read is already in progress, the echo and data transfer are immediate. Deferring the echo until a read operation is active allows the program to specify the mode of the terminal, such as No Echo or Convert Lower Case to Upper, to modify the read operation.

A line entered on a command terminal is terminated by any of several special characters, for example, the RETURN key. A program reading from a terminal can optionally specify a particular line terminator or class of line terminators for read requests (including read PASS ALL requests).

Terminal characteristics are initially established during system generation. Users operating command terminals can modify the characteristics of the particular terminal being used. For example, the user can set the baud rate (transmission speed) or change the terminal line width.

LA120 Hard Copy Terminal

The LA120 is a hard copy terminal which offers exceptional throughput and a number of advanced keyboard-selectable formatting and communication features. It uses a contoured typewriter-styled keyboard and includes an additional numeric keypad and a prompting LED display for infrequently used features.

The LA120 achieves high throughput owing to several features:

- 180 character per second print speed
- 14 data transmission speeds ranging up to 9600 baud
- 1K character buffer to equalize differences between transmission speeds and print speeds
- smart and bidirectional printing so that printhead always takes shortest path to next print position
- high speed horizontal and vertical skipping over white space

In addition to its throughput, the LA120 is distinguished by its printing features. The terminal offers eight font sizes, ranging from expanded (5 characters per inch) to compressed font (16.5 characters per inch). Hence a user could, for example, select a font size of 16.5 cpi and print 132 columns onto an 8½-inch-wide sheet. Other print features include six line spacings ranging from 2 to 12 lines per inch, user-selectable form lengths up to 14 inches, left/right and top/bottom margins and horizontal and vertical tabs.

The LA120 is designed for easy use. Terminal characteristics are selected via clearly labeled keys and simple mnemonic commands. Once the selections have been made, the operator can check his settings by depressing the STATUS key. The terminal will then print a listing of the selected settings.

LA36 Hard Copy Terminal

The LA36 is an exceptionally reliable hard copy terminal. It is a lower-priced device than the LA120 with lower throughput (30 cps vs. 165 cps) and fewer print features.

The LA36 uses a typewriter-like keyboard which produces 128 ASCII characters, consisting of 95 upper- and lower-case printing characters and 33 control characters, and is available with optional special character sets, including various foreign language character sets.

Characters are printed using a 7 × 7 matrix with horizontal spacing of ten characters per inch and vertical spacing of six lines per inch. To ensure clear visibility of the printed line, the print head automatically retracts out of the way when not in operation. Adjustable pin-feed tractors allow for a variable-form width from 3 to 14-7/8 inches (up to 132 columns). The print mechanism will accommodate multipart forms (with or without carbons) of up to six parts.

The LA36 operates at speeds of 110, 150, or 300 baud (10, 15, or 30 characters per second). Printable characters are stored in a buffer during the carriage return operation. While more than one character is in the buffer, the printer mechanism operates at an effective speed of 60 characters per second.

VT100 Video Terminal

The VT100 Video Terminal is an upper- and lower-case ASCII terminal which offers a variety of controllable character and screen attributes. The VT100 features a typewriter-like detachable keyboard which includes a standard numeric/function keypad for data entry applications. Also featured are seven LEDs, four of which are program-controlled, used as operator information and diagnostic aids.

The VT100 offers a number of advanced features. The most important of these are:

- ability to select either of two screen sizes: 24 lines by 80 columns or 14 lines by 132 columns
- ability to select either double-width single-height characters or double-width double-height characters on a line by line basis
- smooth scrolling and split screen capability
- ability to set baud rates, tabs, and Answer Back messages from the keyboard and to store these in RAM (Random Access Memory)
- special line drawing graphic characters providing the ability to display simple graphics for business or laboratory applications
- ability to select black-on-white characters or white-on-black characters on a full screen basis

In addition, several options further extend the capabilities of the VT100. These include the advanced video option, which adds selectable blinking, underline, and dual intensity characters to the existing reverse video attribute; the provision of space, power, and interconnects for the future addition of a terminal processor; and additional RAM allowing 24 lines of 132 characters.

DZ11 Terminal Line Interface

The DZ11 is a serial line multiplexer whose character formats and operating speeds are programmable on a per-line basis. A DZ11 connects the UNIBUS with up to a maximum of 8 or 16 asynchronous serial lines, depending on the configuration. Each line can run at any one of 15 speeds.

Local operation with EIA terminals is possible at speeds up to 9600 baud. Remote dial-up terminals can operate full-duplex at speeds up to 300 baud using Bell 103 or 113 modems, or up to 1200 baud using the Bell 212 modem.

The DZ11 optionally generates parity on output and checks parity on input. Incoming characters are buffered using a 64-character silo buffer. Outgoing characters are processed on a programmed interrupt request basis.

REAL-TIME I/O DEVICES

To enhance real-time performance, VAX supports the DR-11B, the LPA11-K direct memory access (DMA) interfaces, and the DR780 32-bit high performance parallel interface (VAX/11-780 only). These devices allow data to be transferred from a peripheral device to memory and vice-versa without the intervention of computer programs except at the initialization and completion of transfers. The result is that CPU involvement in I/O operations is greatly reduced. Further, since these devices are capable of "driving" large blocks of information at high speeds, their usage can greatly increase I/O bandwidth, i.e., the capacity of the system to sustain a total data transfer load. I/O bandwidth is an important performance measure in real-time applications, since such applications require data transfers between external devices and computer memory.

LPA11-K

The LPA11-K is an intelligent (dual-microprocessor) direct memory access controller that buffers real-time data and transfers them to VAX memory in efficient blocks (rather than a word at a time). Since the LPA11-K has automatic buffer switching capability, transfers may occur continuously. Via a system call, the programmer can instruct the LPA11-K to take samples from a data source at specified time intervals. Sampling is handled by the microprocessors, without the intervention of the CPU. Under VMS, the LPA11-K can be accessed via VAX-11 FORTRAN, VAX-11 BASIC, VAX-11 BLISS-32, and MACRO.

The LPA11-K operates in two distinct modes: dedicated mode and multirequest mode.

In multirequest mode, up to eight requests can be active concurrently. Each user's sampling rate is a user-specified multiple of the common real-time clock rate; thus independent rates can be maintained for each user. Each request specifies the device so that A/D, D/A or digital I/O can be synchronously sampled; the transition of a bit in a digital word can synchronize the sampling with a user event. In multirequest mode, throughput is determined by the number and types of requests. The aggregate throughput rate for all users is typically 15,000 samples per second.

In dedicated mode, one user can sample from analog-to-digital converters, or output to a digital-to-analog converter. Two analog-to-digital converters can be controlled simultaneously. Sampling is initiated by an overflow of the real-time clock, or by an external signal. Two sampling algorithms are implemented. One, at each overflow, samples both analog-to-digital converters in parallel, allowing two channels to be sampled simultaneously. The other algorithm samples the two converters on an interleaved basis, beginning with the first whose sampling begins on alternate clock overflows.

The LPA11-K supports the following I/O devices on VAX:

- AA11K (four-channel 12-bit D/A converter)
- AD11K (eight-channel 12-bit A/D converter)
- AM11K (multiplexer board)
- DR11K (16-bit parallel, general device interface)
- KW11K (real-time clock)

DR11-B

The DR11-B is a general purpose, direct memory access (DMA) digital interface to the UNIBUS. The DR11-B, rather than using programmed controlled data transfers, operates directly to or from memory, moving data between the UNIBUS and the user device. The DR11-B, like the LPA11-K, is a block transfer device. However, it is less expensive than the LPA11-K, does not include a microprocessor, and can only handle a single task for a single programmer.

The DR11-B interface consists of four registers: command and status, word count, bus address, and data. Operation is initialized under program control by loading word count with the 2's complement of the number of transfers, specifying the initial memory or bus address where the block transfer is to begin and by loading the command/status register with function bits. The user device recognizes these function bits and responds by setting up the control inputs. If the user device requests data from memory of a UNIBUS device, the DR11-B performs a UNIBUS Data In transfer (DATI) and loads its data register with the information held at the referenced bus address. The outputs of this register are available to the user device; this output data is buffered. If the user device requests data to be written into memory, the DR11-B performs a UNIBUS Data Out transfer (DATO), moving data from the user device to the referenced bus address; this input data is not buffered. Transfers normally continue at a user-defined rate until the specified number of words is transferred. The DR11-B has the capability of transferring data at a rate of 500,000 bytes/second, but actual transfer rates depend upon the particular configuration.

DR780

The DR780 is a high performance general purpose interface adaptor that enables users to directly interface custom devices to a VAX-11/780 system or to connect two VAX-11/780 systems. This high performance, general purpose interface provides a 32-bit parallel data path capable of transferring data up to 6.67 megabytes/second.

The architecture of the DR780 uses separate interconnect structures for transfer of control information and data. The control interconnect is an asychronous 8-bit bidirectional path for transferring control information to and from the user device. The 8-bit width of the control interconnect makes it possible to have up to 256 individual registers in the user device. The data interconnect is a synchronous 32-bit bidirectional path synchronized to a single clock (either the internal DR780 clock or a clock provided in the user device). By using the DR780 internal clock, the transfer rate is selectable under program control from .156 to 6.67M bytes/sec.

The DR780 provides the high performance interface to utilize the system bandwidth of the VAX-11/780. However, to achieve DR780 bandwidths over 2.0 Mb/second, it is required that the system include two interleaved memory controllers.

Typical applications of the DR780 are high-speed data collection, CPU to CPU communications, signal processing, and interfacing to graphics and array processors.

INTERPROCESSOR COMMUNICATIONS LINK

VAX permits interprocessor communications via the DMC11 communications link or via the MA780, multiport (shared) memory. MA780 is supported by the VAX-11/780 processor only.

DMC11

The DMC11 communications link is designed for high-performance point-to-point serial interprocessor connection based on the DIGITAL Data Communications Message Protocol (DDCMP). The DMC11 provides local or remote interconnection of two computers over a serial synchronous link. Both computers can include the DMC11 and DECnet software, or both computers can include the DMC11 and implement their own communications software. For remote operations, a DMC11 can also communicate with a different type of synchronous interface provided that the remote system has implemented the DDCMP protocol.

By implementing the DDCMP protocol in its high-speed microprocessor, the DMC11 ensures reliable data transmission and relieves the host processor of the details of protocol operation (including character and message synchronization, header and message formatting, error checking, and retransmission control). The DDCMP protocol detects errors on the channel interconnecting the system using a 16-bit Cyclic Redundancy Check (CRC-16). Errors are corrected by automatic retransmission. Sequence numbers in message headers ensure that messages are delivered in proper order with no omissions or duplications.

The DMC11 supports full- or half-duplex operation. Full-duplex operation offers the highest throughput and is used when the communications facilities permit two-way operation. The DDCMP protocol permits continuous simultaneous transmission of data messages in both directions when buffers are available and there are no errors on the channels.

Where both computers are located in the same facility, the DMC11 permits transmission at speeds of up to 1,000,000 bits per second over coaxial cable up to 6,000 feet long, or speeds of up to 56,000 bits per second over coaxial cable up to 18,000 feet long. The necessary modems for local interconnection are built in. Where the computers are located remotely and connected using common carrier facilities, the DMC11 permits transmission of up to 19,200 bits per second using an EIA interface. A DMC11 can interface to synchronous modems such as the Bell models 208 and 209, or other synchronous modems conforming to the RS232-C standard.

MA780

MA780 multiport memory is a bank of MOS semiconductor memory with error correcting code (ECC) that can be shared by up to four VAX-11/780 systems. Each system can randomly access all of the shared memory in exactly the same way it accesses its local memory.

Each MA780 can be expanded from a minimum of 256K bytes to a maximum of 2M bytes. This storage is in addition to each system's local memory, which can be as large as 8M bytes. Since there can be up to two MA780s connected to a CPU, a VAX-11/780 system can now directly address up to 12M bytes of physical memory.

Extensions to VAX/VMS make access to the shared memory transparent to the programmer. That is, processes can be moved from one CPU to another with transparency to the programs involved.

The MA780 can be thought of as a very fast communication device between VAX-11/780 systems. Specifically, VAX/VMS provides support for interprocessor communications through the sharing of data regions, VMS mailboxes, and common event flags. VAX/VMS also allows code to be shared among CPUs.

The MA780 can be used to configure multiple computer systems for very high throughput. Depending on the application, the CPUs can be arranged in either a parallel or pipeline manner, as described in Figure 5-1 below:

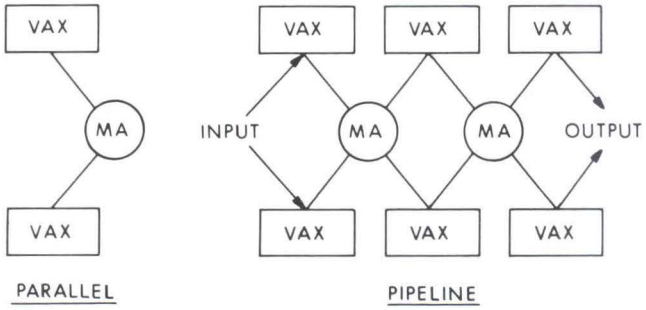


Figure 5-1
Multiported Memory Configurations

In parallel systems, two or more appropriately programmed CPUs can divide a task. This allows the CPUs to effectively pool their power to finish the job quickly. Pipeline systems can increase throughput by allowing instantaneous data exchange between CPUs that are handling sequential parts of an application.

CONSOLE STORAGE DEVICES

The VAX-11/780 console utilizes the RX01 floppy disk

while the VAX-11/750 console utilizes the TU58 magnetic tape cartridge.

RX01 Floppy Disk Cartridge

The RX01 floppy disk is an integral part of the VAX-11/780 console subsystem, storing microdiagnostics and system software. This feature facilitates fast diagnosis (initiated both locally and remotely), simplifies bootstrapping and initialization and improves software update distribution.

The RX01 is a random access mass memory subsystem that stores data in fixed length blocks on a flexible diskette with preformatted, industry standard headers. The RX01 is a single drive floppy capable of storing 256K bytes of data. The RX02 floppy disk system can also read/write data formatted for the RX01 floppy disk.

TU58 tape cartridge

The TU58 Tape Cartridge Drive is an important part of the VAX-11/750 console subsystem. Because the TU58 is connected directly to the CPU, it maintains the capability to administer diagnostics even with some system components inoperative. This feature significantly increases system reliability. The TU58 may also be used to boot the system, to load files into physical memory, and to store files which describe and execute site-specific bootstrap procedures.

The tape cartridge is preformatted to store 2048 records, each containing 128 bytes. The controller provides random access to any record. The TU58 searches at 60 inches per second (ips) to find the file requested, then reads at 30 ips. Data read from the tape are verified through checksums at the end of each record or header. All data transfers between the TU58 and the host are in 512 byte blocks, with the TU58 concatenating four 128 byte records to accomplish this. Data are transferred to the CPU at approximately 2 KB per second.

6 The Operating System

VAX/VMS is the general purpose operating system for VAX systems. It provides a reliable, high-performance environment for the concurrent execution of multiuser timesharing, batch, and real-time applications. VAX/VMS provides:

- virtual memory management for the execution of large programs
- event-driven priority scheduling
- shared memory, file, and interprocess communication data protection based on ownership and application groups
- programmed system services for process and subprocess control and interprocess communication

VAX/VMS uses memory management features to provide swapping, paging, and protection and sharing of both code and data. Memory is allocated dynamically. Applications can control the amount of physical memory allocated to executing processes, the protection pages, and swapping. These controls can be added after the application is implemented.

VAX/VMS schedules CPU time and memory residency on a pre-emptive priority basis. Thus, real-time processes do not have to compete with lower priority processes for scheduling services. Scheduling rotates among processes of the same priority.

VAX/VMS allows real-time applications to control their virtual memory paging and execution priority. Real-time applications can eliminate services not needed to reduce system overhead. Processes granted the privilege to execute at real-time scheduling levels, however, do not necessarily have the privilege to access protected memory and/or data structures.

VAX/VMS includes system services to control processes and process execution, control real-time response, control scheduling, and obtain information. Process control services allow the creation of subprocesses as well as independent detached processes. Processes can communicate and synchronize using mailboxes, shared areas of memory, shared files or multiple common event flag clusters. A group of processes can also communicate using multiported memory.

Applications designers can use the VAX/VMS protection and privilege mechanisms to implement system security and privacy. VAX/VMS provides memory access protection both between and within processes. Each process has its own independent virtual address space which can be mapped to private pages or shared pages. A process cannot access any other process's private pages. VAX/VMS uses the four processor access modes to read and/or write protect individual pages within a process. Protection of shared pages of memory, files, and interprocess communication facilities,such as mailboxes and event flags, is based on User Identification Codes individually assigned to accessors and data.

INTRODUCTION

VAX/VMS is built for executing high-performance applications where:

- Event-driven interprocess communication and procedure and data sharing are important. Order entry and teller transaction systems often consist of many cooperating processes that synchronize record creation and modification.
- Priorities of resource allocation can be set for currently executing jobs. Both real-time processes and resource-sharing processes can execute in the same environment, as in a communications network. High-speed links can be serviced on demand, while interactive terminal users and batch jobs share processor time and peripherals.
- Large programs can be developed to execute in a physical memory smaller than the program's total memory requirements. Engineering computation programs such as simulators often build data arrays which require a large address space to describe the arrays.

The VAX/VMS operating system provides the run time services for executing high-performance application systems. Operations managers and systems programmers have considerable flexibility in designing and controlling data and program flow.

Applications can be divided into several independent subsystems where data and code are protected from one another, and yet which have general communication and data sharing facilities. Jobs can communicate using general, group, or local communication facilities.

Applications which require an immediate response to some external event can be scheduled as real-time processes. When a real-time process is ready to execute, it executes until it becomes blocked or another higher priority real-time process needs the resources of the processor. Normal jobs can be scheduled using a modified pre-emptive algorithm that ensures that they receive processor and peripheral resources at regular intervals commensurate with their processing needs.

If insufficient memory is available for keeping concurrently executing jobs resident, the operating system will swap jobs in and out of memory to allocate each its share of processor time. Real-time processes can be locked in memory to ensure that they can be started up rapidly when they need to execute.

The operating system provides a dynamic virtual memory programming environment. Large programs can be executed in a portion of physical memory that is considerably smaller than the program's memory requirements, without requiring the programmer to define overlays. The operating system optimizes its virtual memory system for program locality and provides tools that support optimization. It makes program performance predictable and controllable by allowing the programmer to restrict physical memory usage, and by bringing in large amounts of a program at one time. Processes executing under VAX/VMS page against themselves and not against the entire system; thus heavily paging processes executing large programs do not affect the paging of other processes.

The operating system provides sophisticated peripheral device management for sharing, protection, and throughput. Devices can be shared among all jobs or reserved for exclusive use by particular jobs. Input and output for low-speed devices are spooled to high-speed devices to increase throughput. Files on mass storage devices can be protected from unauthorized access on an individual, group, or volume basis.

Furthermore, the I/O request processing system is optimized for throughput and interrupt response. The operating system provides the programmer with several data accessing methods, from logical record accessing for easy, device-independent programming to direct I/O accessing for extremely rapid data processing. Files can be stored in any of several ways to optimize subsequent processing.

VAX/VMS provides the programming tools, scheduling services, and protection mechanisms for multiuser program development. Programmers can write, execute, and debug programs on the system interactively, and also create batch command files that perform repetitive program development operations without requiring their attention.

Although it provides a multiuser program development environment, VAX/VMS is unlike traditional program development timesharing systems. VAX/VMS is an application-oriented operating system that optimizes total system throughput and response to high-priority activities. As in a timesharing system, interactive jobs can be given equal opportunities for resource acquisition. In addition, the system can be executing real-time applications while program development jobs run, since higher priority activities always have the ability to pre-empt lower priority activities.

COMPONENTS AND SERVICES

The operating system is the collection of software that organizes the processor and peripherals into a high-performance system. The operating system's basic components include:

- processes that control initial resource allocation, communicate with the system operator, and log errors
- the command interpreters
- user-callable process control services
- memory management routines
- shared run time library routines
- scheduling routines and swapper
- file and record management services
- interrupt and I/O processing routines
- compatibility mode executive routines
- hardware and software exception dispatching

The operating system's jobs run as independent activities on the system. They include the Job Controller, which initiates and terminates user processes and manages spooling; the Operator Communications Manager, which handles messages queued to the system operators; the Swapper, which controls the swapping of a processes working set in and out of main memory; and the Error Logger, which collects all hardware and software errors detected by the processor and operating system.

A command interpreter executes as a service for interactive and batch jobs. It enables the general user to request

the basic functions that the operating system provides, such as program development, file management, and system information services.

Both hardware-detected and software-detected exception conditions are tracked through the exception dispatcher. The exception dispatcher passes control to user-programmed condition handlers or, in the case of system-wide exception conditions, to operating system condition handlers.

The operating system's memory management routines include the image activator, which controls the mapping of virtual memory to system and user jobs; the pager, which moves portions of a process in and out of memory as required; and various system services, callable by users that want to manage their virtual address space directly. They respond to a program's dynamic memory requirements, and enable programs to control their allocated memory, share data and code, and protect themselves from one another.

The scheduler controls the allocation of processor time to system and user jobs. The scheduler always ensures that the highest priority, ready-to-execute real-time process receives control of the processor until it relinquishes it. When no real-time processes are ready to execute, the scheduler dynamically allocates processor time to all other jobs according to their priorities and resource requirements. The swapper works in conjunction with the scheduler to move entire jobs in and out of memory when memory requirements exceed memory resources. The swapper ensures that the jobs most likely to execute are kept in memory.

The operating system's I/O processing software includes interrupt service routines, device-dependent I/O drivers, device-independent control routines, and user-programmed record processing services. The I/O system ensures rapid interrupt response and processing throughput, and provides programming interfaces for both special purpose and general purpose I/O processing.

The next few sections discuss some of the concepts basic to understanding the operating system's functions and services. They are followed by descriptions of the services available to individual and cooperating processes, and descriptions of memory management and scheduling for the systems programmer.

PROCESSING CONCEPTS

To support high-performance multiprogramming application environments, the operating system provides the applications programmer with the tools to implement:

- shared programs
- shared files and data
- interprocess communication and control

To enable the programmer to write shared programs easily, the operating system treats a program independently of the context in which a program executes. The context defines the privileges assigned by the system manager to a particular user. Users with different privileges can share programs, and the operating system will enforce protection independently of the program.

The operating system controls privilege and accounts for resource allocation by job. A job can be performing processing operations under the direction of one user at a terminal, or it can be performing processing operations for several users at multiple terminals. A job can consist of one or several independently executing processes that share the resource allocations for that job. Jobs can be grouped into application subsystems that share files and communication channels that are protected from other application subsystems.

Programs and Processes

The four concepts important for understanding how the operating system supports multiprogrammed application systems are:

- **image**, or executable program
- **process**, or image context and address space
- **job**, or detached process and its subprocesses
- **group**, or set of jobs that can share resources

These concepts are for the most part transparent to the general user whose only contact with the system is the operating system's command language interpreter or an application's command interface. They are, however, significant concepts for the applications programmer. Figure 6-1 illustrates the concepts of groups, jobs, processes, and images.

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

The environment in which an image executes is its context. The complete context of an image not only includes the state of its execution at any one time (known as its hardware context), it also includes the definition of its resource allocation quotas, such as device ownership, file access, and maximum physical memory allocation. These resource allocation quotas are determined by the quotas given to the user who runs the image.

Two or more users can execute the same image concurrently; that is, image code can be shared, in which case the image is executing in two or more different contexts. An image context, including the address space used by an image, is called a **process**. The operating system schedules processes, and a process provides a context in which an image executes.

The distinction between an image and a process is a significant one. We can speak of two processes, each executing the FORTRAN compiler. There may be only one copy in physical memory of the FORTRAN compiler's image, but two different contexts in which the image executes. In one context, the compilation may have just begun; in the other context, it may be almost complete. In one context, the compiler may be reading and writing files listed in one directory; in the other context, the compiler may be reading and writing files listed in another directory.

A process executes only one image at a time, but it provides the context for serially executing any number of different images. For example, when a user logs on the sys-

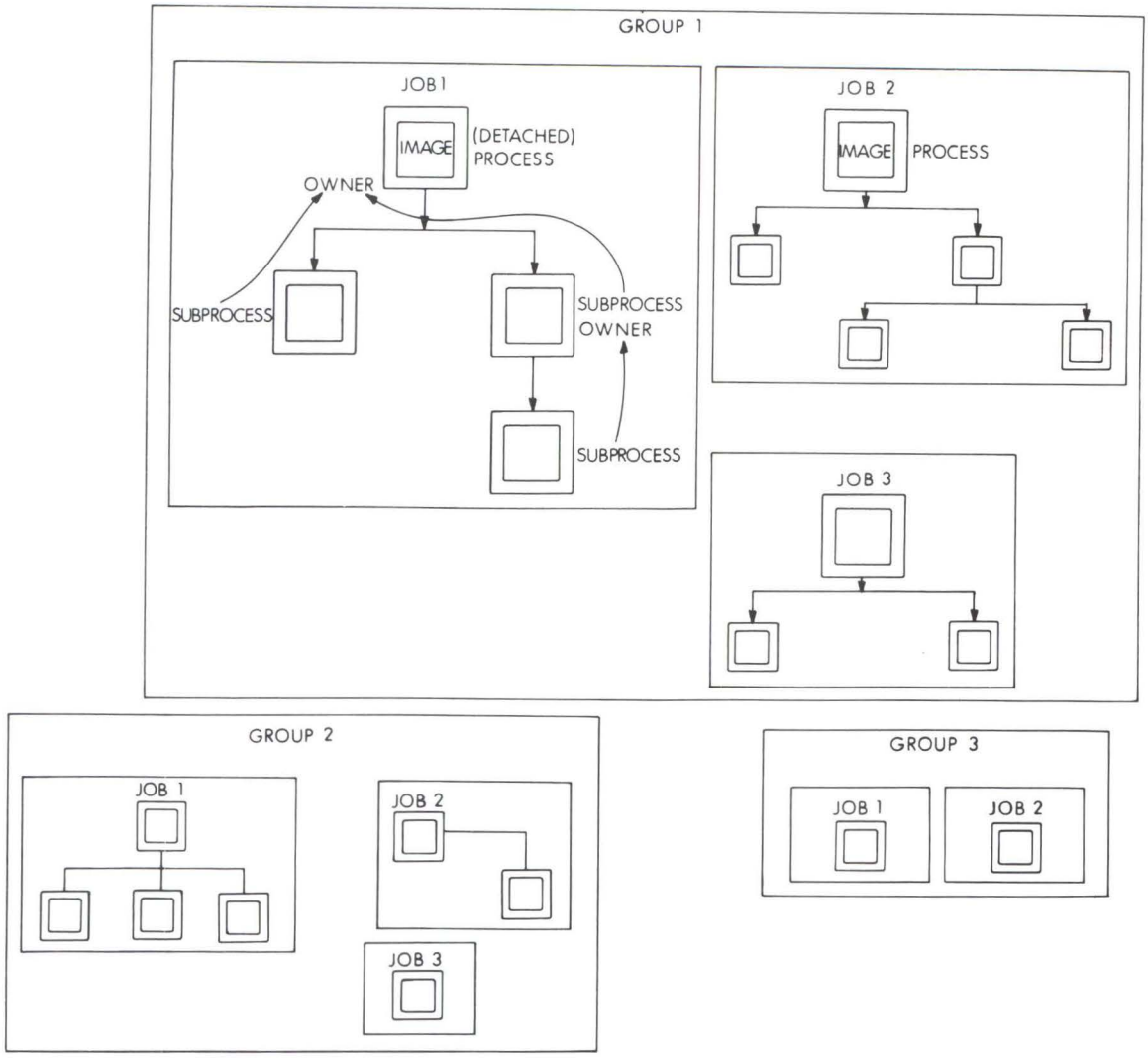


Figure 6-1

Programs and Processes

tem at an interactive terminal, the operating system creates a process for that user. If the user edits a file, the editor image executes in the context of that user's process. If the user then compiles a program, the compiler image executes in the context of that user's process. A process thus acts as a continuous "envelope" for a user's activities.

An image executing in the context of a process can create **subprocesses**. A subprocess can be thought of as an auxiliary process in which a given image executes. When an image creates a subprocess, it identifies the image to be executed in the context of the subprocess or the source of commands to be interpreted in that process. An image executing in a subprocess context can in turn create other subprocesses.

The process executing the image that creates a subprocess is an **owner process**. An owner process has complete control over the execution of the subprocesses it creates. It determines which of its privileges it will allow a subprocess to have. Each detached process and all subprocesses created below it heirarchically share a common pool of quotas. The owner process can control the scheduling of its subprocess, and it can delete the subprocess. When an owner process terminates, all of the subprocesses it owns are terminated.

A **detached process** is the process created by the operating system on behalf of a user who logs onto the system and requests services of the system using a command interpreter. A detached process has no owner. Normally, only the operating system can create detached processes, but a suitably privileged application program could also create a detached process and start up an application command interface to execute images serially for the detached process or any subprocess it creates.

A **job** consists of a detached process and all the subprocesses it creates, and all the subprocesses they create, etc. Jobs are the accounting entities that the system uses to control resource allocation. All processes constituting a job are scheduled independently (they can compete for

processor time individually to overlap processing), but they share the total resources allocated to the detached process. Each job has a set of resources that it can use, i.e., authorized quotas. Subprocesses share these quotas with the detached process.

Jobs can be associated in **groups**. Groups are the basis for the definition and development of application subsystems. Groups are mutually exclusive, that is, if a job belongs to one group, it does not belong to any other group. A process with appropriate privilege can control the execution of other processes in the same group. Processes in the same group can synchronize their activities using protected group communication facilities.

Resource Allocation

The resources of the system are the processor, memory, and peripherals. The system handles many jobs simultaneously, and each job can have different resource requirements. The operating system enables jobs to share the resources according to their individual needs, and the operating system protects each job and its data from other jobs on the system.

The operating system controls resource allocation dynamically through its scheduling, memory management, device allocation, and I/O processing software, and statically through the authorization of users.

The system manager is responsible for creating an authorization file entry for each user of the system. The authorization file provides the operating system with the resource quotas and limits for each job. For example, there are quotas and limits that control:

- total processor time usage
- number of subprocesses a job can create
- number of simultaneously open files
- process virtual and physical memory usage
- number of simultaneous I/O transfers

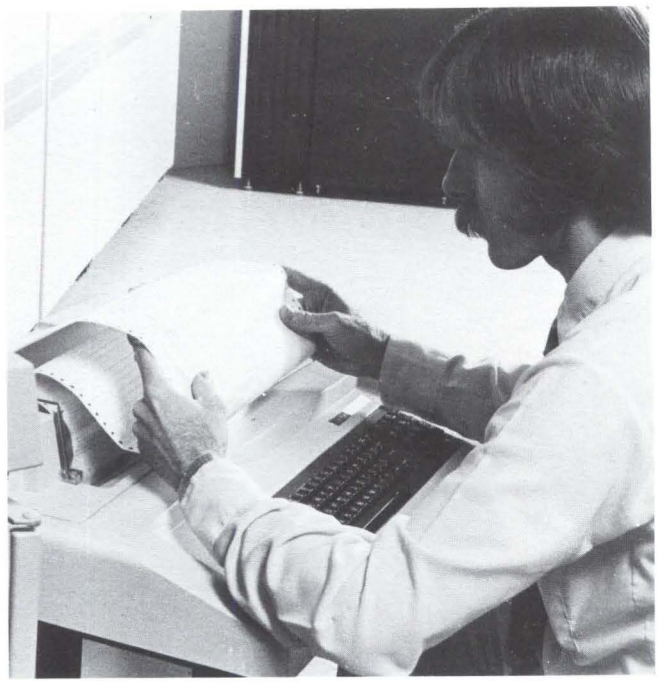
Separate authorization files located on each disk volume control disk usage quotas.

Privileges

In addition to providing job quotas, the user authorization file provides the base definition for each user's privileges. There are potentially 64 distinct privileges that can be individually granted or withheld. Among them are privileges that give the job the right to:

- alter the priority of a process
- execute a user-written program at a more privileged access mode
- execute operator functions
- create detached processes
- set up the communication facilities used by cooperating processes
- control other processes in the same group

Whenever the user executes an image, the image can at most acquire only those privileges and quotas granted directly to that user's job by the authorization file, unless the image is a **known image**. Known images are installed by the system manager, and while they execute they provide a second, dynamic set of privileges granted a user.



When the user executes a known image, the process has the privileges and quotas granted to the user in the authorization file, *plus* those run time execution privileges granted specifically to that image. While that image executes, the user may have the privilege to perform operations not granted when executing any other image. For example, one known image is the operating system's LOGIN image, which enables a user to log on the system. The LOGIN image has the privilege required to access the user authorization file to obtain the user's privileges and quotas.

Protection

The basis for data protection in the VAX system is the user identification code (UIC). A UIC consists of two numbers: a group number and a member number. The system manager assigns each user a user identification code (UIC) in the user authorization file. Images that the user executes are given or denied data access privileges based upon the user's UIC.

When a file or an interprocess communication facility is created, it is assigned a UIC and a protection code. The UIC determines which group of users or programs, and which members within the group, have controlled access to that data. The protection code provides the access control.

The protection code applies to four types of access: **read, write, execute,** and **delete**. Each type of access can be given or denied to:

- the owner: the user whose UIC is the same as the UIC assigned to the data.
- the group: every user whose UIC group number is the same as that assigned the data.
- the world: every user whose UIC group number is different or the same from that assigned the data. (everyone on the system)

- the system: every user or program with the privilege SYSPRV, and those whose UIC group number is a system privileged group number (1-X, where X is a number specified by the sysgen parameter, MAXSYSGROUP).

For example, in a common application of the protection scheme, a user can create a program image file and assign it the same UIC as the user's own UIC (the default case). The user can give it a protection code to:

- enable the user (and all other users with the same UIC) to read, write, execute, and delete the file
- enable other users in the group to execute the program image, but prevent them from reading, writing or deleting the file
- prevent all users outside the group (other than privileged system users) from reading, writing, executing, or deleting the file
- enable the the privileged system users to read the file (so that it can be backed up, for example)

Read and write access applies to both files and interprocess communication facilities. Delete access applies only to files, and execute access applies only to program image files. (The privileges and quotas granted in the user authorization file control creation and deletion for interprocess communication facilities.)

USER PROCESS ENVIRONMENT

The user program environment is the process, which is the entity the operating system schedules for execution. Each process has its own independent address space in which an image executes. Each image executing in a process can call system service procedures to acquire resources and request special processing services from the operating system. The following paragraphs introduce program virtual address allocation and the fundamental system service procedures available to user programs directly, as well as indirectly through the more complex programmed requests provided by the operating system.

Virtual Address Space Allocation

Process virtual address space is the set of 32-bit addresses that an image executing in the context of a process uses to identify byte locations in virtual memory. For the purpose of allocating virtual memory to processes, the operating system divides process virtual address space into four sets of virtual addresses. The first three sets of addresses are called the program region, the control region, and the system region. The fourth set of addresses is unused.

Figure 6-2 illustrates the general allocation of virtual address space for each process. Addresses in the first two regions are used for process code and data, where the first region is generally used for image-specific code and data, and the second for stacks, process permanent data, buffers, and operating system code. Addresses in the system region are also used for code and data maintained by the operating system, but in this case the addresses refer to the same locations in every process context. The system region addresses provide a set of locations whose addresses are independent of process context, and therefore do not have to be context switched.

When a user program is translated and linked, the image is allocated addresses starting with address 512 and continuing up. The first page is not normally allocated (although it can be) because it helps catch programming errors caused by improperly initialized pointers, by branching or jumping to 0, or by passing 0 or other small addresses as arguments. The linker allocates the remainder of address space to image sections according to whether they are shared or private, position-independent or position-dependent, and read-only or read/write, such that memory protection can be used to full advantage in preventing and isolating programming errors.

The addresses in the control region are used to identify the locations containing temporary image control information and data such as the stacks, permanent process control information such as I/O channel allocations, and code provided by the operating system. These addresses are allocated from address $2^{31}-1$ *down*. One reason this method of allocating in reverse is convenient is because the control region contains the process stacks, and stacks grow to lower addresses as data are added, and to higher addresses as data are removed.

There are four stack areas reserved in the control region, one for each access mode protection level that the processor provides for software executing in the context of a process. (Refer to the VAX Processors section for a description of access modes.) The stack seen by the image executing in the program region is the user stack. All other stack areas are protected from that image. These stacks are used by operating system software executing in the context of the process on behalf of the image the process is executing. For example, command interpreters use the supervisor stack, the record management services use the executive stack, and the exception dispatcher and some exception handlers use the kernel stack.

The system region addresses, which start at address 2^{31}, are used to identify the locations containing the entry vectors for system service procedures, followed by locations containing privileged operating system code and data. The system service entry vectors are permanently reserved virtual addresses so that no relinking is required if system services are modified. Other addresses in the system region are not generally used by the image allocated to the program region, and access to areas mapped by these addresses is restricted.

System Services

An image requests services of the operating system directly through calls to the **system services**. The system services are to the operating system what the instruction set is to the processor. They provide all the primary resource request activities, such as I/O processing and interprocess communication. Other programmed requests available to the user are often derived from system services. For example, the record management services use the I/O processing system services as the basis for logical record processing functions.

Images that use the system services can be written in assembly language or any native programming language that has a Call statement. (Refer to the Languages section and the section on the RSX-11M Programming Environment for system services available for compatibility mode

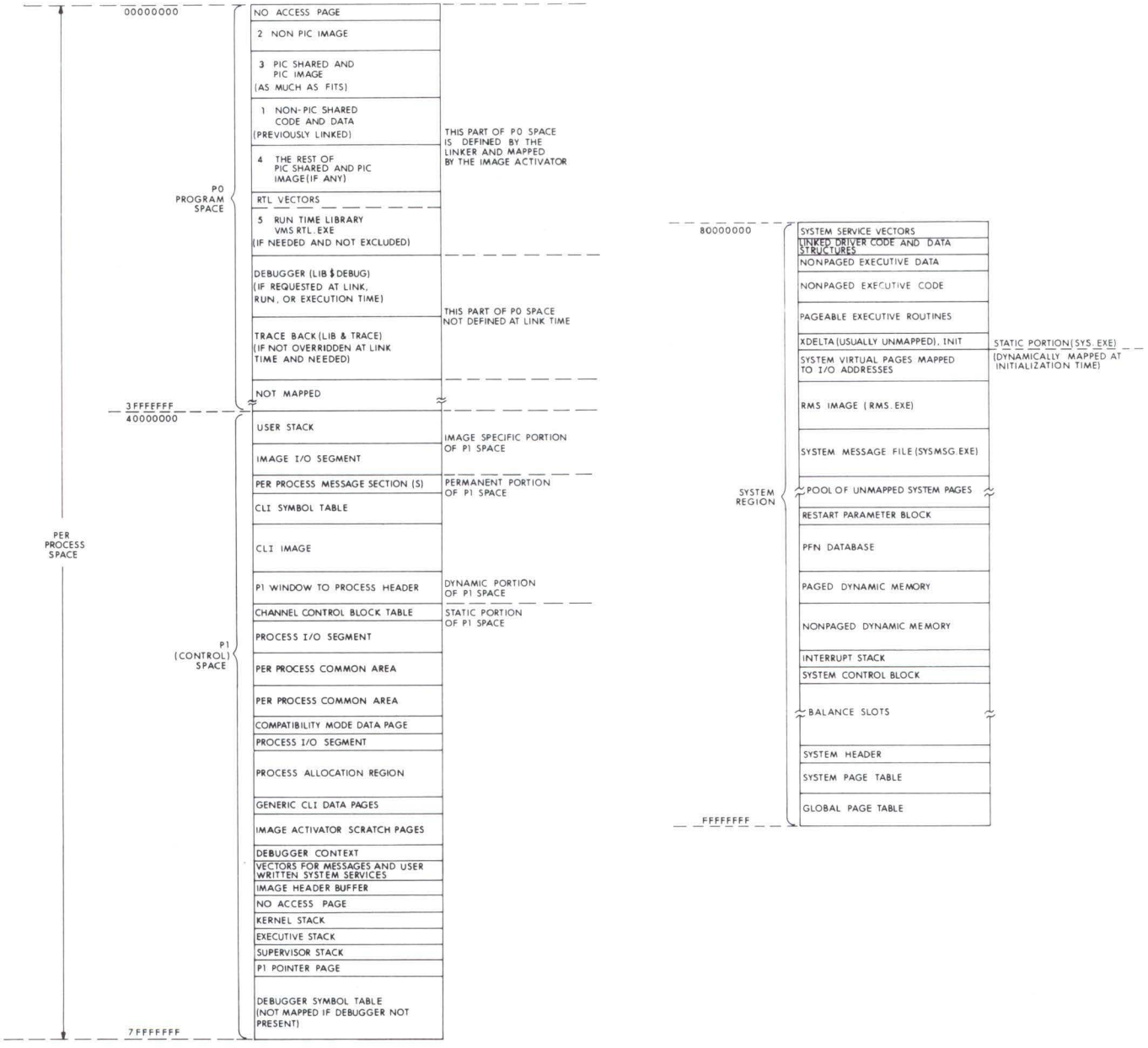


Figure 6-2

Virtual Address Space Allocation

programming languages.) The Call interface is the same independent of the programming language selected with the exception of CORAL 66.

Table 6-1 summarizes the system services available to the applications programmer, some of which are controlled by privilege. The table also lists those system services used primarily by the operating system but which can also be used by suitably privileged application system code.

Table 6-1

System Services

I/O SERVICES FOR DEVICE-DEPENDENT I/O	
Assign I/O Channel ($ASSIGN)	Establish a path for an I/O request. Establish a path for network operations.
Deassign I/O Channel ($DASSGN)	Release linkage for an I/O path. Release a path from the network.
Get I/O Channel Information ($GETCHN)	Provide information about a device to which an I/O channel has been assigned.
Get I/O Device Information ($GETDEV)	Provide information about a physical device.
Allocate Device ($ALLOC)	Reserve a device for exclusive use by a process and its subprocesses.
Deallocate Device ($DALLOC)	Relinquish exclusive use of a device.
Queue I/O Request ($QIO)	Initiate an input or output operation and continue processing while I/O is in progress.
Queue I/O Request and Wait ($QIOW)	Initiate an input or output operation and cause the process to wait until the I/O is complete before continuing execution.
Cancel I/O on Channel ($CANCEL)	Cancel pending I/O requests on a channel.
Formatted ASCII Output ($FAO)	Perform ASCII string substitution, and convert numeric data to ASCII representation and substitute in output.
Formatted ASCII Output with List Parameter ($FAOL)	
I/O SERVICES FOR MAILBOXES AND MESSAGES	
Create Mailbox and Assign Channel ($CREMBX)	Create a temporary mailbox. Create a permanent mailbox.
Delete Mailbox ($DELMBX)	Mark a permanent mailbox for deletion.
Broadcast ($BRDCST)	Send a high-priority message to a specified terminal or terminals.
Send Message to Accounting Manager ($SNDACC)	Control accounting log file activity. Write an arbitrary message to the accounting log file.
Send Message to Symbiont Manager ($SNDSMB)	Request symbiont manager to initialize, modify, or delete a printer, device, or batch job queue. Request symbiont manager to queue a batch or print file, or delete or change characteristics of a queued file.
Send Message to Operator ($SNDOPR)	Write a message to designated operator(s) terminal(s). Enable or disable an operater's terminal, send a reply to a user request or initialize the operator's log file.
Send Message to Error Logger ($SNDERR)	Write arbitrary data to the system error log file.
Get Message ($GETMSG)	Return text of system or application error message from message file.
Put Message ($PUTMSG)	Write a system or application error message to the current output and error devices.
LOGICAL NAME SERVICES	
Create Logical Name ($CRELOG)	Place logical name/equivalence name pair in process logical name table. Place logical name/equivalence name pair in group logical name table. Place logical name/equivalence name pair in system logical name table.
Delete Logical Name ($DELLOG)	Remove logical name/equivalence name pair from process logical name table. Remove logical name/equivalence name pair from group logical name table. Remove logical name/equivalence name pair from system logical name table.
Translate Logical Name ($TRNLOG)	Search logical name table for a specified logical name and return its equivalence name when the first match is found.

Table 6-1 (con't)
System Services

EVENT FLAG PROCESSING

Associate Common Event Flag Cluster ($ASCEFC)	Create a temporary common event flag cluster.
	Create a permanent common event flag cluster.
	Create a common event flag cluster in memory shared by multiple processors.
	Establish association with an existing common event flag cluster.
Disassociate Common Event Flag Cluster ($DACEFC)	Cancel association with a common event flag cluster.
Delete Common Event Flag Cluster ($DLCEFC)	Mark a permanent common event flag cluster for deletion.
Set Event Flag ($SETEF)	Turn on an event flag in a process-local event flag cluster.
	Turn on an event flag in a common event flag cluster.
Clear Event Flag ($CLREF)	Turn off an event flag in a process-local event flag cluster.
	Turn off an event flag in a common event flag cluster.
Read Event Flags ($READEF)	Return the status of all event flags in a process-local event flag cluster.
	Return the status of all event flags in a common event flag cluster.
Wait for Single Event Flag ($WAITFR)	Place the current process in a wait state pending the setting of an event flag in a process-local event flag cluster.
	Place the current process in a wait state pending the setting of an event flag in a common event flag cluster.
Wait for Logical OR of Event Flags ($WFLOR)	Place the current process in wait state pending the setting of any one of a specified set of flags in a process-local event flag cluster.
	Place the current process in a wait state pending the setting of any one of a specified set of flags in a common event flag cluster.
Wait for Logical AND of Event Flags ($WFLAND)	Place the current process in a wait state pending the setting of all specified flags in a process-local event flag cluster.
	Place the current process in a wait state pending the setting of all specified flags in a common event flag cluster.

AST (ASYNCHRONOUS SYSTEM TRAP) SERVICES

Set Power Recovery AST ($SETPRA)	Establish AST routine to receive control following power recovery condition.
Set AST Enable ($SETAST)	Enable or disable the delivery of ASTs.
Declare AST ($DCLAST)	Queue an AST for delivery.

CONDITION HANDLING SERVICES

Set Exception Vector ($SETEXV)	Define condition handler to receive control in case of hardware- or software-detected exception conditions.
Set System Service Failure Exception Mode ($SETSFM)	Request or disable generation of a software exception condition when a system service call returns an error or severe error.
Unwind from Condition Handler Frame ($UNWIND)	Delete a specified number of call frames from the call stack following a nonrecoverable exception condition.
Declare Change Mode or Compatibility Mode Handler ($DCLCMH)	Designate a routine to receive control when change mode to user instructions are encountered.
	Designate a routine to receive control when change mode to supervisor instructions are encountered.
	Designate a routine to receive control when compatibility mode exceptions occur.

PROCESS CONTROL SERVICES

Create Process ($CREPRC)	Create a subprocess.
	Create a detached process.
Set Process Name ($SETPRN)	Establish a text string to be used to identify the current process.
Get Job/Process Information ($GETJPI)	Return information about the current process.
	Return information about the current process context of other processes in the same group.
	Return information about any other process in the system.
Delete Process ($DELPRC)	Delete the current process, or a subprocess.
	Delete another process in the same group.
	Delete any process in the system.
Hibernate ($HIBER)	Make the current process dormant but able to receive ASTs until a subsequent wakeup request.
Schedule Wakeup ($SCHDWK)	Wake a process after a specified time interval or at a specific time.

Table 6-1 (con't)
System Services

Cancel Wakeup ($CANWAK)	Cancel a scheduled wakeup request.
Wake ($WAKE)	Restore executability of the current process or a hibernating subprocess.
	Restore executability of a hibernating process in the same group.
	Restore executability of any hibernating process in the system.
Suspend Process ($SUSPND)	Make the current process or a subprocess nonexecutable and unable to receive ASTs until a subsequent resume or delete request.
	Make another process in the same group nonexecutable and unable to receive ASTs until a subsequent resume or delete request.
	Make any process in the system nonexecutable and noninterruptible until a subsequent resume or delete request.
Resume Process ($RESUME)	Restore executability of a suspended subprocess.
	Restore executability of a suspended process in the same group.
	Restore executability of any suspended process in the system.
Exit ($EXIT)	Terminate execution of an image and returns to command interpreter.
Force Exit ($FORCEX)	Cause image exit for the current process or a subprocess.
	Cause image exit for a process in the same group.
	Cause image exit for any process in the system.
Declare Exit Handler ($DCLEXH)	Designate a routine to receive control when an image exits.
Cancel Exit Handler ($CANEXH)	Cancel a previously established exit handling routine.
Set Priority ($SETPRI)	Change the execution priority for the current process or a subprocess.
	Change the execution priority for a process in the same group.
	Change the execution priority for any process in the system.
Set Resource Wait Mode ($SETRWM)	Request wait, or that control be returned immediately, when a system service call cannot be executed because a system resource is not available.
Set Privileges ($SETPRV)	Allow a process to enable or disable specified user privileges.

TIMER AND TIME CONVERSION SERVICES

Get Time ($GETTIM)	Return the date and time in system format.
Convert Binary Time to Numeric Time ($NUMTIM)	Convert a date and time from system format to numeric integer values.
Convert Binary Time to ASCII String ($ASCTIM)	Convert a date and time from system format to an ASCII string.
Convert ASCII String to Binary Time ($BINTIM)	Convert a date and time in an ASCII string to the system date and time format.
Set Timer ($SETIMR)	Request setting of an event flag or queuing of an AST, based on an absolute or delta time value.
Cancel Timer Request ($CANTIM)	Cancel previously issued timer requests.
Schedule Wakeup ($SCHDWK)	Schedule a wakeup for the current process or a hibernating subprocess.
	Schedule a wakeup for a hibernating process in the same group.
	Schedule a wakeup for any hibernating process in the system.
Cancel Wakeup ($CANWAK)	Cancel a scheduled wakeup request for the current process or a hibernating subprocess.
	Cancel a scheduled wakeup request for a hibernating process in the same group.
	Cancel a scheduled wakeup request for any hibernating process in the system.
Set System Time ($SETIME)	Set or recalibrate the current system time.

MEMORY MANAGEMENT SERVICES

Adjust Working Set Limit ($ADJWSL)	Change maximum number of pages that the current process can have in its working set.
Expand Program/Control Region ($EXPREG)	Add pages at the end of the program or control region.
Contract Program/Control Region ($CNTREG)	Delete pages from the end of the program or control region.
Create Virtual Address Space ($CRETVA)	Add pages to the virtual address space available to an image.
Delete Virtual Address Space ($DELTVA)	Make a range of virtual addresses unavailable to an image.

Table 6-1 (con't)
System Services

Create and Map Section ($CRMPSC)	Identify a disk file as a private section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify a disk file containing shareable code or data as a temporary global section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify a disk file containing shareable code or data as a permanent global section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify a disk file containing shareable code or data as a system global section and establish correspondence between virtual blocks in the file and the process' virtual address space.
	Identify one or more page frames in physical memory as a private or global section and establish correspondence between the page frames and the process' virtual address space.
Update Section File on Disk ($UPDSEC)	Write modified pages of a private or global section into the section file.
Map Global Section ($MGBLSC)	Establish correspondence between a global section and a process' virtual address space.
Delete Global Section ($DGBLSC)	Mark a permanent global section for deletion.
	Mark a system global section for deletion.
Set Protection on Pages ($SETPRT)	Control access to a range of virtual addresses.
Lock Pages in Working Set ($LKWSET)	Specify that particular page cannot be paged out of the process' working set.
Unlock Pages from Working Set ($ULWSET)	Allow previously locked pages to be paged out of the working set.
Purge Working Set ($PURGWS)	Remove all pages within a specified range from the current working set.
Lock Page in Memory ($LCKPAG)	Specify that particular pages may not be swapped out of memory.
Unlock Page in Memory ($ULKPAG)	Allow previously locked pages to be swapped out of memory.
Set Process Swap Mode ($SETSWM)	Control whether or not the current process can be swapped out of the balance set.
CHANGE MODE SERVICES	
Change Mode to Executive ($CMEXEC)	Execute a specified routine in executive mode.
Change Mode to Kernel ($CMKRNL)	Execute a specified routine in kernel mode.
Adjust Outer Mode Stack Pointer ($ADJSTK)	Modify the current stack pointer for a less privileged access mode.

I/O System Services

The operating system provides the programmer with two request interfaces for performing input/output operations: the I/O system services and the record management services. Record management services, discussed in the Data Management Facilities section, provides a general purpose file and record programming interface that satisfies most I/O processing needs, and allows the programmer to implement I/O processing quickly. The I/O system services provide the programmer with direct control over the I/O processing resources of the operating system. In particular, the I/O system services enable the programmer to:

- perform both device-independent and device-dependent I/O processing
- read and write blocks on mass storage media using physical (device-oriented), logical (volume-relative), or virtual (file-relative) addressing

The I/O system recognizes several types of devices, and within the extents of their capabilities, all devices are programmed in the same manner. All devices can be sequentially accessed, including mass storage devices such as disks and magnetic tapes, and record-oriented devices, such as terminals, card readers and line printers. In addition, disk volumes can be accessed randomly.

Mass storage volumes can be either file-structured or non-file-structured, according to the choice of the user. The I/O system services enable programmers to use either the physical (device assigned) address or a logical (driver assigned) address for directly addressing blocks on **foreign** mass storage volumes. A foreign volume can be either non-file-structured, or structured with the user's own file structure. If the volume is structured using the operating system's Files-11 disk file or ANSI magnetic tape structure, the I/O system services enable the programmer to address blocks directly using virtual (file system assigned) addresses.

A special type of record-oriented device is the **mailbox**, which is a virtual device that a process creates for the receipt of messages from other processes. Mailboxes are

treated like any other record-oriented device: they can be read from and written to using either the I/O system services or record management services. Mailboxes are discussed further in the section on Interprocess Communication.

Before a process requests I/O to a device, it obtains a channel assignment from the operating system. A process can use a device name or a **logical name** in a channel assignment request to identify the device for which the channel is desired.

A device name is a unique name assigned by the operating system to a particular physical device. The name identifies the type of device and its controller and line or unit number, as applicable. For example, DMA3: is the operating system's device name for the RK06 disk drive unit 3 on controller A, and TTA12: is the operating system's name for the terminal on line 12 on multiplexer A.

A logical device name is any string of characters a user or program assigns to a device name assigned by the operating system. The Create Logical Name system service not only enables a process to define logical names for device names, but it enables a process to assign logical names to any portion of a file specification, or to other logical names. Furthermore, logical names can be assigned on a per-process, per-group, or system-wide basis. (For more information on logical names, refer to the Data Management Facilities Section.)

Once a channel is obtained, a process can issue I/O requests on that channel. The Queue I/O Request system service is a general I/O request interface. All I/O using system services is asynchronous: both I/O and computation can be taking place simultaneously. An I/O request is simply queued to the device driver and control is normally returned to the requesting process before the I/O operation is complete.

The process is responsible for synchronizing with I/O completion. The process can simulate synchronous I/O processing by using the Queue I/O Request and Wait system service, or it can continue to execute during the I/O operation and request I/O completion notification using the general purpose event flag or asynchronous system trap notification mechanisms.

Real-time interface extensions (connect-to-interrupt and map-to-I/O page) provide the real-time programmer (MACRO and BLISS-32) a technique of more simply interfacing to user-specialized devices. The connect-to-interrupt facility can be used to cause an interrupt to be delivered directly to the user's program. As a consequence of this approach, response to the interrupt occurs in the shortest time possible without writing an I/O driver.

The map-to-I/O page is a complement to the connect-to-interrupt facility by allowing the user program to access the device registers. Before these extensions were available, the user had to write both a device driver and an application program to achieve the same results.

Local Event Flags

An event flag is a status bit used for posting an event, such as I/O completion or elapsed time interval. Event flags are an extremely efficient means of starting up or synchronizing procedures.

Each process has available for its own use two local event flag clusters, each of which contains 32 event flags. Eight flags in the first cluster are reserved by the operating system. A process can set, clear, and read individual event flags, as well as wait for one or more event flags to be set. The advantage to having two clusters of event flags is that the flags in each cluster can be treated as a related group. A process can wait until any of a specified set of flags in a particular cluster is set, or wait until all of a specified set of flags in a particular cluster are set.

Aside from their use with I/O processing and timer scheduling, a process can assign its own meanings to local event flags. Event flags can be used to coordinate several asynchronous events, such as multiple I/O request completions, or to simplify asynchronous processing. For example, a program may wish to know if a terminal user has typed a CTRL/C (indicating the desire to interrupt execution) only at well-defined points during processing. An asynchronous system trap routine can set an event flag to indicate that a CTRL/C has been received.

Asynchronous System Traps

An asynchronous system trap (AST) is a software-simulated interrupt used for event notification within a process. An asynchronous system trap routine is a procedure that handles an AST. AST routines provide an efficient means for processing events that can occur at any time during processing (such as terminal input) because they eliminate the need for polling.

For example, a program can specify AST routines for I/O request processing, timer scheduling, and power recovery. When the I/O operation completes, time interval expires, or power is restored, the operating system declares an AST. When the AST is delivered, the operating system interrupts the process and executes the AST routine. A process can be hibernating and still receive ASTs declared for it.

Code executing at one processor access mode can declare an AST for code executing at the same or a less privileged access mode. The operating system automatically disables AST delivery while an AST routine is executing, and code executing at a given access mode can explicitly disable AST delivery. While ASTs are disabled, the operating system queues any ASTs waiting to be delivered to that access mode in the order in which they were declared. When AST delivery is again enabled for that acess mode, the ASTs are delivered in the order in which they were queued.

Exception Conditions and Condition Handlers

A program may request the processor or a system service to do operations they cannot perform correctly. For example, a program might inadvertently issue a divide instruction using a divisor of zero. Normally there is no way to recover and the program cannot continue. In this system, however, it is possible for a program to continue if it declares a **condition handler** that can correct the situation. If a user program declares a condition handler, control transfers to the condition handler when an **exception condition** occurs.

This system treats all errors or special events that occur synchronously with respect to a program's execution as

exception conditions, and provides a general purpose mechanism for dispatching condition handlers. Exception conditions include:

- errors from which the processor cannot normally recover, such as the divide by zero arithmetic trap
- special conditions for which a program does not wish to test continually, for example, the floating point overflow arithmetic trap or unsuccessful system service completion

Some of the exceptions detected by the processor are handled automatically by the operating system. For example, the pager is a condition handler for translation-buffer-not-valid faults.

In addition to processor and system service detected exception conditions, any software procedure can define cases for which it will fail or produce an exception by calling a system library procedure that signals an exception condition. The search sequence for a condition handler is independent of the nature of the exception condition: the search sequence is the same whether an exception condition is detected by hardware or software.

A process can declare two kinds of condition handlers: those that are process-wide and those that are applicable to individual procedures. Process-wide condition handlers are declared using the Set Exception Vector system service, which enables a process to declare a primary and a secondary condition handler. Condition handlers applicable to individual procedures are declared by the procedure when it is called using one instruction.

When an exception condition occurs, the exception dispatcher does not differentiate between exception conditions, it simply transfers control to the first condition handler it can find that wants to handle the exception condition. This method for handling exception conditions is an efficient means of transferring control to the appropriate condition handler rapidly, since condition handling is defined by the module or modules in which an exception condition may occur.

For programs written in high-level languages, each language may have different definitions of what is and what is not an exception condition. As the user program calls language functions, the exception conditions for those functions can be handled locally with the procedure. And where exception conditions should be handled on a process-wide basis, the primary and secondary exception vectors provide a top level exception condition trap. For example, when a user program is linked with the debugger, the debugger uses the primary exception vector to declare a process-wide condition handler.

INTERPROCESS COMMUNICATION AND CONTROL

This system supports both simple and complex job definitions. A simple job is a detached process created by the operating system on behalf of the user who logs in at a terminal, or for the purpose of executing a batch job. A simple job serially executes images, but it does not create subprocesses.

A complex job is one in which a detached process creates subprocesses in which designated images execute. These subprocesses can also create their own subprocesses, and so on. The advantage of a complex job over a simple job is that a complex job performs parallel processing operations because it has control over several images executing concurrently.

The following sections describe the services that enable a process to control and communicate with other processes.

Process Control Services

The system services provide process control by enabling a process to:

- create and delete subprocesses
- hibernate, then reactivate, a process via the Hibernate/Wake and Suspend/Resume system services

The ability to create subprocesses is granted to a user by the system manager, where the number of subprocesses a job can create is a resource limit. When a process creates a subprocess, it can give the subprocess all or some of its privileges, and its resource quotas and limits are shared with the subprocess. Other resource quotas are shared between the creator and the subprocess.

The Hibernate/Wake and Suspend/Resume mechanisms are methods of process control which are especially efficient in real-time applications. They allow the user to prepare an image for execution and then place it into a wait state until some event occurs which requires its activation.

The Hibernate system service provides the greatest flexibility in sequencing processes for execution. When a Hibernate system service is invoked, normal execution can be resumed only by issuance of a $WAKE system service (or a variant, $SCHDWK, which allows wake-up at an absolute time or at a fixed time interval). However, a hibernating process can be interrupted temporarily by the delivery of an AST (Asynchronous System Trap). When the AST service routine completes execution, the process continues hibernation. If, however, the process calls the $WAKE system service during execution of the AST service routine, the process wakes itself after the service routine completes. Figure 6-3 shows an example of a program which uses the hibernate and wake system services.

Using the $SUSPEND system service, a process can place itself or another process into a wait state similar to hibernation. However, a suspended process cannot be as easily activated as a hibernating one. It cannot, for example, be interrupted by delivery of an AST. Nor can it wake itself, but can only resume normal execution following issuance of a $RESUME system service by another process. Table 6-1 summarizes the differences between hibernation and suspension.

The interprocess system services can be used by a process to control another process executing in the same group. While only an owner process can create and delete subprocesses, a process can be given the privilege to suspend, resume, and wake other processes in its group.

Jobs with sufficient privilege can also create detached processes, and delete, suspend, resume, or wake any process in the system. These privileges are normally reserved for the operating system or the system manager.

Interprocess Communication Facilities

In addition to providing process control services, the oper-

```
Process: GEMINI

ORION: .ASCID 'ORION'                          ;SUBPROCESS NAME
FASTCOMP: .ASCID 'COMPUTE.EXE'                 ;IMAGE

                .
                .
1       $CREPRC_SPRCNAM=ORION,—
                        IMAGE=FASTCOMP         ;CREATE ORION - HE'LL
                                               ;SLEEP
        BLBC    R0,ERROR                       ;BRANCH IF SERVICE ERROR
                                               ;CONTINUE
                .
                .
                .
3       $WAKE_S PRCNAM=ORION                   ;WAKE ORION
        BLBC    R0,ERROR                       ;BRANCH IF SERVICE ERROR
                .
                .
                .
        $WAKE_S PRCNAM=ORION                   ;WAKE ORION AGAIN
        BLBC    R0,ERROR                       ;BRANCH IF SERVICE ERROR

Process: ORION

FASTCOMP:
2       .WORD   0                              ;ENTRY MASK
10$     $HIBER_S                               ;SLEEP 'TIL GEMINI WAKES
                                               ;ME
        BLBC    R0,ERROR                       ;BRANCH IF SERVICE ERROR
                .                              ;PERFORM...
                .
                .
        BRB     10$                            ;BACK TO SLEEP
```

Notes:

1. Process GEMINI creates the process ORION, specifying the image name FASTCOMP.
2. The image FASTCOMP is initialized, and ORION issues the $HIBER system service.
3. At an appropriate time, GEMINI issues a $WAKE request for ORION. ORION continues execution following the $HIBER service call. When it finishes its job, it loops back to repeat the $HIBER call and to wait for another wake.

Figure 6-3
Program Using Hibernate/Wake System Services

ating system provides process communication facilities for synchronizing execution, for sending messages, and for sharing common data. The three techniques that cooperating processes can use to communicate are:

- common event flags
- mailboxes
- shared areas of memory

Common event flags are available by group association to processes within jobs. Mailboxes and shared areas of memory are more general purpose facilities which can be limited or unlimited in scope. They can be limited to a specific member family within a group or to a specific group of jobs, or they can be extended to all jobs in the system.

Common Event Flags

In addition to the local event flags available to each process, cooperating processes can communicate using common event flags. Every group in the system can define any number of common event flag clusters. Each cluster contains 32 flags. The flags can be assigned any meaning for the processes in the group.

Each process in a group can associate with up to two of its group's common event flag clusters at one time. A process can read, set, clear, or wait for common event flags to be set. The ability to read, set, or clear event flags is controlled by the protection code and User Identification Code assigned to the common event flag cluster.

Common event flag clusters can also be used by cooperating processes on different processors in a multiport memory configuration.

Mailboxes

A mailbox is a record-oriented virtual I/O device created by a process. Mailboxes can be used to pass status information, return codes, messages, or any other data from one process to another. A process can protect its mailboxes from read and/or write access by any process outside

its member family or outside its group. Mailboxes can also be used by processes to communicate with other processes on different processors in a multiport memory configuration.

All of the I/O system services and record management services can be applied to mailboxes. Other processes write messages to a process' mailbox by queuing write requests for the device. A process reads messages in its mailbox by queuing read requests for the device. A process can request AST notification when anything is written to its mailboxes, and it can assign mailboxes logical names.

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 all or a portion of an image or data file that can be mapped in a process virtual address space at run time. Global sections can be used for shared data structures, as communication regions for cooperating processes, or they can be used simply to eliminate multiple copies of image code or data.

Global sections can be created dynamically by a process or they can be permanently present 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 created by a sufficiently privileged process, and remain until they are explicitly deleted. They are loaded into and removed from memory dynamically as references are made to them.

Each process that maps a global section into its virtual address space can have a different access privilege to a global section. When a global section is created, it is assigned a User Identification Code (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 be shared by all processes in the system, or shared only by processes within a particular group, or shared only by processes within a particular job. One or more controlling processes can have write privileges while other processes in the system, group, or job have only read privileges.

A process can map to a global section explicitly by issuing a Map Global Section system service, or it can be mapped implicitly by referring to a shareable image. If an image references a shareable image, the linker does not normally include the shareable image in the image. The shareable image is installed as a global section or set of global sections. When the image is executed, the image activator calls the Map Global Section system service on behalf of the image. For example, the Common Run Time Procedure Library is a shareable image consisting of library procedures that is mapped as a system-wide permanent global section. The use of permanent global sections significantly reduces the size of programs using common library procedures and the overall system memory requirements.

Interprocessor Communication Facility

VAX/VMS support for the multiport memory subsystem means that both user data and subroutines may reside in shared memory for access from multiple processors connected to the multiport memory. All three of the interprocess communication facilities, i.e., common event flags, mailboxes, and global sections, may reside in multiport memory, thus providing interprocessor communication facilities. Through the use of logical names, common event flags, mailboxes, and global sections may be placed either in local or multiport memory, transparently to the program. Common event flags, mailboxes, and global sections are the communication facilities permitted in multiport memory configurations.

MEMORY MANAGEMENT

In a multiprogramming system, many processes coexist simultaneously in main memory. The system switches between these processes, giving each some time to execute. In most multiprogramming environments, however, the number, size, and kind of concurrently executing processes change rapidly, while the amount of memory available for processes remains constant. Users log on and off the system, production activities vary periodically, and special production jobs occur. Since it is generally inefficient to have available the maximum amount of memory that might ever be needed at one time, it becomes the task of the operating system to provide a dynamic memory that responds to the changing multiprogramming environment.

VAX/VMS uses two interdependent complementary techniques to allocate limited memory to competing processes: paging and swapping. These techniques relieve the general programmer of concern for memory allocation while still allowing system programmers to optimize program performance in limited configurations. This section and the following section on scheduling discuss how this system's paging and swapping techniques extend limited memory resources with minimum effect on the system or programs when the system has sufficient memory to hold all concurrently executing processes.

Mapping Processes into Memory

The operating system's memory management software is responsible for creating and maintaining the information used to map the virtual addresses used in a program to physical memory addresses. The unit mapped is the **page**, which is a block of 512 contiguous byte locations in physical memory.

Virtual addresses are also grouped into 512-byte pages, and each page of virtual addresses can be mapped to a page of real memory locations. Any number of virtual pages can be mapped to one physical page. Unlike systems that partition or statically allocate portions of physical memory, this system dynamically allocates physical memory, with the result that pages of a process may be scattered anywhere throughout memory. It is never the concern of the programmer to determine how physical memory is allocated. To illustrate this, Figure 6-4 shows how two processes might be mapped into physical memory.

When a process is created, the operating system sets up its mapping information, called **page tables**. Each process has its own page tables mapped by system region virtual

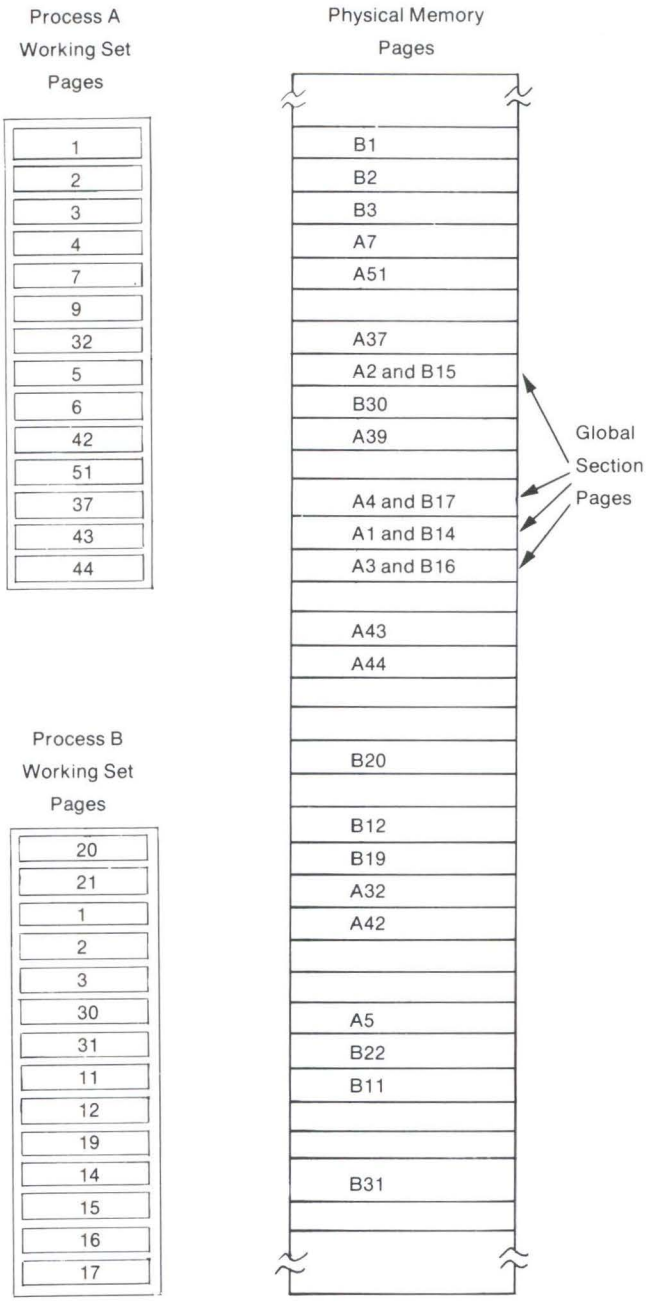


Figure 6-4

Mapping Processes into Memory

addresses. (Refer to the Processor section on memory management for a complete description.) Initially, a process page table simply maps those pages of the control region that define the permanent process context.

When a program is linked, the addresses the linker assigns in the image are always virtual addresses. The linker has no knowledge of how physical memory is allocated. Its primary function is to build descriptions of the size and protection requirements of the program's code and data areas.

When an image is executed, the memory management software uses the linker's descriptions of code and data areas to map image virtual pages to physical pages. The operating system's image activator builds the image's mapping information in the process' page tables.

Process Virtual Memory and Working Set

The total virtual memory requirement of a process is called **process virtual memory**. Process virtual memory consists of all the pages of the process program region and control region which are mapped by the process page tables.

At any one time, some of the pages of process virtual memory may be mapped to disk and some to physical memory. The physical memory requirement of a process is the **process working set**. When a process is executing, a process working set consists of all the pages of a process' virtual memory residing in physical memory that the process can directly access without incurring a page fault, *plus* any actively used portions of the process page tables and process header information.

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 has to be in a process 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 a process working set affects its paging and swapping performance, as well as affecting the number of process working sets that can be resident when the process working set is resident.

A process may increase or decrease its working set, within the authorized limits, through the use of command language commands or system service calls.

Under version 2.0 of VAX/VMS, working set size adjustments are made automatically by the operating system. This facility, when enabled by the system manager, monitors the page fault rate of a process and automatically increases or decreases the working set (again within authorized limits) to optimize performance and memory usage. This automatic adjustment provides a more immediate response in system reaction/performance.

Paging

Through its paging technique, the operating system can execute programs that are too large to fit in the amount of physical memory allocated to a process, without requiring the programmer to define overlays. Inactive portions of a program are automatically stored on disk while the active portions are resident in memory. When the program references a disk-resident portion of the program, the operating system reads in, or **pages** in, the referenced portion, moving out other portions of the program to disk if necessary. This system's paging technique has several features that distinguish it from other techniques:

- clustering, or the ability to read in several pages at one time
- paging processes against themselves, not against the entire system
- maintaining an available page pool from which processes can recover recently discarded pages without incurring disk I/O

- writing back to disk only the modified pages that are released from a process working set and only writing them when several have accumulated
- activating a process waiting for page fault I/O to execute AST routines when they are delivered

When the operating system activates an image for the first time, a number of pages are read into memory from the image file on disk. The number of pages read in the first time can be controlled by a cluster factor the programmer can assign optionally per image. The ability to read in several pages at once allows the image to execute for some time without incurring page faults, and provides significantly improved responsiveness in starting programs.

A process is subsequently paged only when it executes an image that needs more pages than the process is allowed to have in its working set. If the number of pages in the image plus the number of pages for the remainder of the process is less than the working set size limit, all the pages are read in and the process is never paged.

If all the pages are not read in initially, at some point the image will reference the pages that have not been read in. At that time, the process incurs a page fault, that is, a reference to a page not mapped in the process working set.

The operating system's pager is a condition handler that executes when a process incurs a page fault. If the working set size limit has not yet been reached, the pager reads in the faulted page from disk, plus any additional pages, again according to a cluster factor for that section of the image.

If a page fault occurs when the working set size limit is reached, the pager obtains a page from a pool of available pages to read in the faulted page, and releases the least recently faulted page from the process working set into the pool and writes it to disk if it has been modified. Figure 6-5 illustrates two different size working sets for a process running the same program in each case. The illustration shows the order in which the pages were faulted. (Refer to Figure 6-2 to see how the pages appear in virtual address space.)

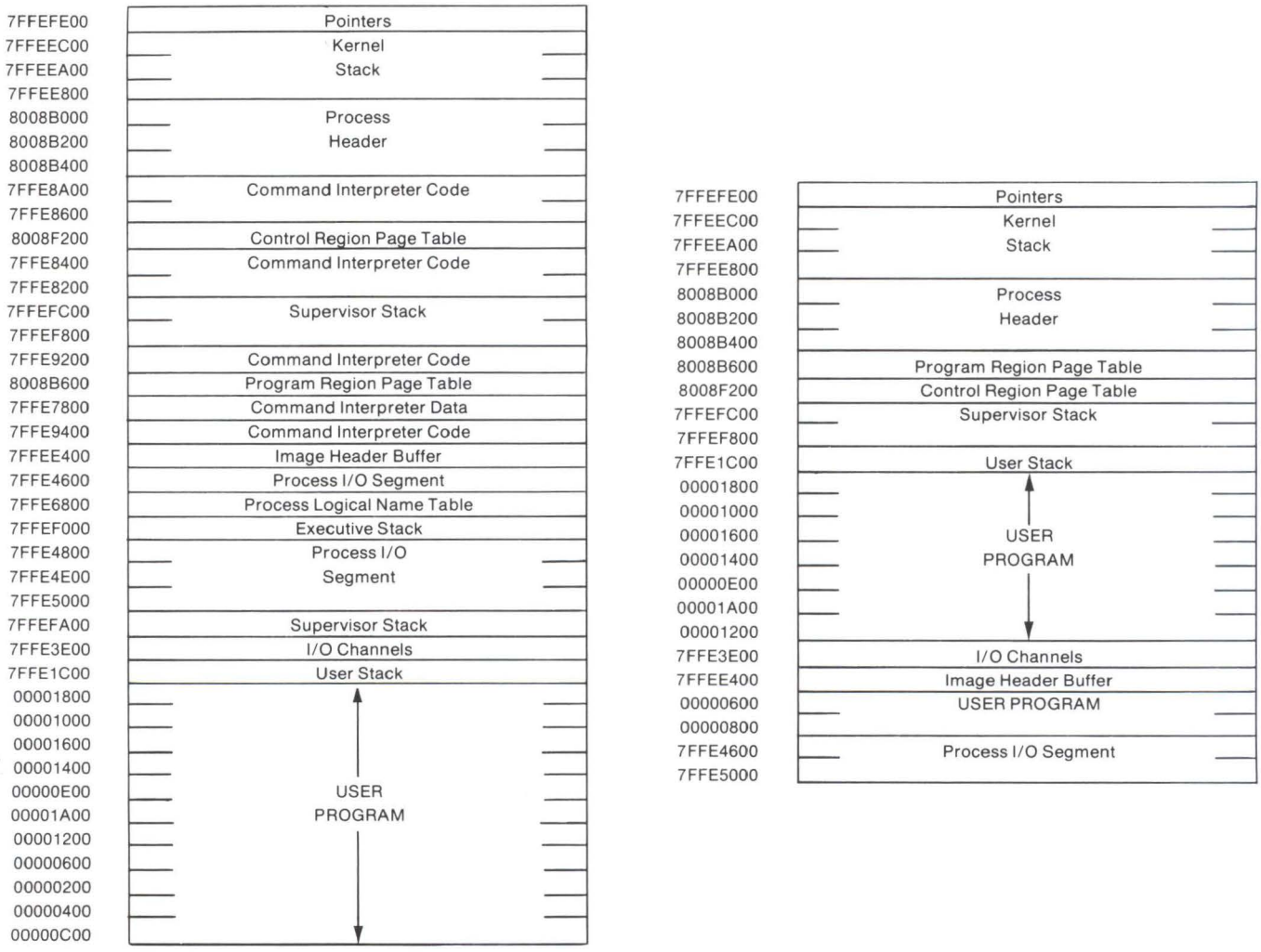


Figure 6-5

Process Working Sets

The pager pages a process only against itself. It does not release pages of one process to satisfy another's needs. This ensures that only those processes that need paging are affected by paging. Other processes in the system need not be affected by another process's memory requirements.

The list of available pages works as a cache of pages that effectively extends a working set size above its limit when few processes are competing for memory resources and there are many pages in the list. If a process faults a page that was released and is still in the list, the page does not have to be read in from disk, it is simply taken from the list and remapped into the working set.

When a page is released, it is placed on one of two lists: the free page list or the modified page list. Modified pages are pages the process has written into and, if they need to be added to the free page list to be used by another process, must be saved on disk. Modified pages are only written to disk when the modified page list exceeds a threshold size or when an image's execution is terminated and the files containing the modified pages must be closed. When modified pages are written, they are writen in clusters to increase system performance.

Virtual Memory Programming

The processor provides the programmer with a large virtual address space and rapid address translation, and the operating system provides the programmer with extremely efficient mapping and paging algorithms. Furthermore, these memory management mechanisms are totally transparent to the application programmer. It is not necessary for a programmer to be concerned with address allocation or page mapping: the high-level language compilers and the linker take advantage of the memory management mechanisms to set up the memory allocation optimal for most programming requirements.

For those systems with limited memory or special processing requirements, however, this system enables users to control and optimize memory management. The system manager can control the memory allocation requirements of the system as a whole by initialization parameters such as desired and minimum acceptable number of available pages, and of individual jobs by user authorization parameters such as paging file usage limit and maximum working set size. It is possible to have a process avoid paging entirely by making its working set size equal to its virtual memory requirements, or to reduce paging by choosing a working set size that satisfies the average demand for pages over time.

The programmer also has the ability to control memory allocation for images in two ways: through properly coded programs and through the memory management system services. This system's memory management software optimizes for program locality. Programs that incur paging infrequently are those in which the code and data used during each stage of processing are contained in the fewest possible number of virtually contiguous pages.

For the most part, the linker allocates virtual addresses so that images require the minimum mapping information possible. The programmer can also ensure that images that process large data structures require the least possible mapping information and potential paging by organizing data structures as if they were disk-resident files. In general, the programmer need not be concerned with data structures such as tables and arrays whose elements are virtually contiguous and sequentially processed.

Large data structures that are randomly accessed can, however, be optimized. For example, processing down a linked chain in which the chain elements are spaced far apart with no useful data in between requires that an image reference a large number of pages in a short period of time. If all of the pages cannot fit in the process working set at the same time, the references to successive chain elements will incur disk I/O.

On the other hand, a large data structure can be efficiently accessed using directory trees, where a page or set of consecutive virtual pages contains all one kind of information. One page can contain all of the information that points to randomly arranged, but virtually contiguous, pages containing the data processed at that locality.

VAX/VMS Memory Management Services

For those who have special processing requirements, there are system services that control memory management within the quotas and limits assigned by the system manager. These memory management system services enable a process to:

- modify the working set size limit
- add or delete pages from process virtual memory
- expand or contract the program region or control region
- lock pages in the working set
- lock pages in physical memory

A program can impose a limit on process working set size anywhere between the minimum required by the system and the maximum specified by the system manager. The limit can be adjusted in accordance with program behavior and real-time requirements. By maintaining the smallest working set size consistent with an acceptable paging rate, a program that temporarily requires a large working set can reduce its impact on the system. For example, a process control program or simulator might use a small working set while processing interactive initialization commands. Once real-time processing is underway, the program can expand its process working set size to reduce paging. When real-time processing is finished, the program can contract the working set.

A process can add selected pages to and delete selected pages from its virtual memory dynamically. Deleting a page is in effect saying that the image is no longer going to use those virtual addresses, and the operating system does not need to map them to pages in virtual memory. Deleting read/write pages (such as those used for interprocess communication) as soon as they are no longer used eliminates the need for the system to write them out as modified pages to a paging file. When an image has reached its paging file quota, it can delete pages in order to map other pages in its virtual address space.

A process can request an extension to the amount of virtual memory allocated to its program region. The operating system will map zero-filled pages into the process virtual address space following the highest addressed page allocated for the program region. This service is useful for dynamically creating data arrays whose size is not known be-

forehand, and it eliminates the need for allocating a data area in a program image. A process can also extend the initial allocation of pages for the user stack by requesting the operating system to map zero-filled pages into process virtual address space preceding the lowest addressed page allocated for the control region. In Version 2.0 of VAX/VMS, the system will automatically extend the stack if the process references unmapped addresses in the control region.

In unusual situations, a process can lock pages in its working set. Locking a page in the working set is useful when a process does not reference a particular page regularly, but the page needs to be in the working set to increase the performance of the code in that page. For example, it might be desirable to keep the page containing asynchronous system trap routines in a working set to ensure that the routines are started up rapidly when an AST is delivered. Note, however, that locking a page in the working set causes other pages to be paged more frequently, since the page will not be paged out, no matter how long it has been in the working set. A page can be unlocked when it is no longer necessary to keep it in the working set.

It is also possible to lock pages in memory. A page locked in memory is not only locked in the working set, it is not swapped out with the process. This service is useful for real-time processes that need to keep buffers in memory for I/O transfers.

PROCESS SCHEDULING

VAX/VMS features event-driven scheduling based on process priority. Unlike traditional timeshared scheduling systems, this system's ability to respond to events enables it to dispatch real-time processes efficiently as well as to share processing time among normal processes competing for resources. Furthermore, priority assignment enables the user to bias processor time allocation based on process activity, to bias the allocation absolutely for certain processes, or to mix both allocation methods.

The operating system's scheduler and swapper are responsible for ensuring that the processes executing in the system receive processor time commensurate with their priority, which is controlled by assignment, and with their ability to execute, which is controlled by system events.

System Events and Process States

In VAX/VMS, dispatching a process for execution involves little decision making. The selected process is always the highest priority executable process. The real scheduling decisions are made as the result of system events that make processes executable.

A system event is an event that affects the ability of a process in the system to execute. System events include events external to the process currently executing, such as I/O completion or timer interrupt. System events also include events internal to the process currently executing. The process may issue a wait request or a hibernate request, or it may request or release a system resource, for example, a page of memory.

Every active process in the system is listed in one of several state queues that identifies whether or not a process is executable, and if not, the event or resource for which the process is waiting. Whenever a system event occurs, the scheduler adjusts the process state queues accordingly. For example, the scheduler adds a process to the executable state queue when a resource for which it is waiting becomes available, or removes it when it requests an event or resource for which it must wait.

The executable state queue supplies the scheduler with a list of processes that are eligible to execute. Priority determines which process among those eligible executes. Rescheduling occurs when a system event makes executable a process with higher priority than the one currently executing.

Unlike timeshared scheduling, therefore, event-driven scheduling is based on the activities of the processes themselves, not on a time limit imposed by the scheduler. Because scheduling intervals are determined by system events, the interval between rescheduling is random. Quantum keeping and requested timer events provide a minimum level of event activity but, in practice, the average interval between events is determined by the duration of the typical I/O operation.

Priority: Real-Time and Normal Processes

The scheduler recognizes 32 scheduling priorities, where priority 31 is high and 0 is low. Priorities 31-16 are for real-time processes, and priorities 15-0 are for normal processes. When a process is created, the system assigns it a scheduling priority. A program image that the process executes can modify the process priority using a system service. The system manager grants jobs the privilege to execute at real-time priorities.

The scheduler maintains a queue for each scheduling priority. Processes having the same priority are listed in the same queue. The priority assigned to a process when it is created is its base priority. The scheduler does not alter the priority of a real-time process during execution. The scheduler may temporarily increase the priority of a normal process during its execution, but its priority never drops below its base priority.

Scheduling by strict priority for real-time processes and by potentially modifying priority for normal processes allows the scheduler to achieve maximum overlap of compute and I/O activities while still remaining responsive to high-priority real-time applications.

Scheduling Real-Time Processes

When a system event occurs that makes a real-time process eligible to execute, it receives control of the processor unless another higher priority process is currently executing. A real-time process retains control of the processor until it finishes execution, enters a wait state, or is pre-empted by a higher priority process. (Note that under VAX/VMS, real-time processes actually have a higher priority than system processes, thus ensuring that real-time processing will never be encumbered by system overhead.)

A higher priority real-time process can pre-empt any lower priority process whenever a system event occurs that makes it eligible to execute. For example, a device interrupt may occur that signals the completion of an I/O transfer requested by the higher priority real-time process.

When a real-time process is pre-empted to dispatch a

process of higher priority, the pre-empted process is placed at the end of its priority queue. This rotates processes within a priority, with the result that available processor time is distributed among processes of the same priority.

Scheduling Normal Processes

When no real-time processes are executing, the scheduler distributes processor time among the processes on the normal priority levels. As with real-time processes, the scheduler selects the highest priority ready-to-execute normal process. That process executes until it finishes execution, enters a wait state, or is pre-empted by a higher priority process. Unlike real-time process scheduling, however, the scheduler modifies normal process priority whenever a system event occurs for a normal process and whenever a normal process is scheduled.

When a system event occurs that affects a normal process, the scheduler increases the priority of the normal process (but not to more than the maximum priority of 15) and places the process at the tail of the queue for its new priority. The amount of priority increment depends on the nature of the event. For example, the scheduler increases the priority of a normal process on the following events:

- terminal input completed
- terminal output completed
- resource available
- wake, resume, delete request received
- nonterminal I/O completion, page fault completion, or other event

In this case, the terminal I/O events receive the highest priority increments to enable the system to be most responsive to the interactive terminal user. When the scheduler increases a normal process's priority, that process gets control of the processor if its new priority is higher than that of the process currently executing.

Each time a normal process is scheduled, the scheduler decreases its priority by one (unless it is already in its base priority queue) and places it at the end of that priority queue. The effect of dynamically increasing and decreasing normal process priority ensures maximum overlap of computation and I/O.

Swapping and the Balance Set

It is the job of the swapper to keep the scheduler supplied with the highest priority executable processes in configurations that do not have a sufficient amount of physical memory to keep all process working sets memory-resident. The balance set is the set of all process working sets that are currently in memory. The swapper ensures that the balance set always contains the highest priority executable processes by moving low priority or nonexecutable memory resident process working sets to a swap area on disk, and moving high priority or executable process working sets into memory.

Swapping is a very efficient way of extending limited memory resources when many processes are executing concurrently. Process working sets for small processes (less than 64K bytes or 128 pages) can be swapped in and out of memory in one disk I/O operation. Where paging extends limited memory resources on a per-process basis and is limited to moving few pages in and out of memory, swapping balances the memory requirements of the system as a whole.

The swapper is activated whenever a system event occurs that can make a nonresident process resident, a nonresident process executable, or a resident process nonexecutable. For example, a resident process might release sufficient memory to enable the swapper to move in a nonresident process. An I/O completion event might make a nonresident process executable. A resident process might enter a wait state and become nonexecutable. In any case, the swapper uses three conditions to determine which processes should be swapped in and which should be swapped out:

- which processes are executable and which are not (and the reason for the wait state)
- what the process priorities are
- whether a process balance set quantum has expired

The balance set quantum effectively enforces a swapping rotation for compute-bound normal processes. Every normal process is assigned a time quantum that provides a guaranteed minimum amount of time in which the process can perform useful work before it is eligible to be swapped out of the balance set. A process can be pre-empted many times before it has received its full quantum. It remains in the balance set until it completes its first quantum unless a real-time process that is swapped out becomes executable and no other processes can be swapped out to make room for the real-time process.

VAX/VMS Process Control Services

In addition to the VAX/VMS system services that enable processes to create, delete, suspend, resume, and wake other processes, or to hibernate and wake themselves, a process can control the manner in which it is scheduled by:

- setting process swap mode
- setting resource wait mode

A suitably privileged process can request that it not be swapped out of the balance set, even when it becomes inactive. This is useful for high priority real-time processes that need to be activated rapidly when they become executable.

Normally, when a process requires dynamic resources of the system and they are not available, the process enters a wait state until the resources become available. Dynamic resources primarily include the buffer space needed for mailboxes, I/O requests, etc. A process can request to be notified when resources are not available and take alternative action instead of entering a wait state.

I/O PROCESSING

The I/O processing system consists of several modular, interdependent components that enable programmers to choose the programming interface and processing method appropriate for their needs, without incurring run time space or performance overhead for features not used. In addition, 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. Figure 6-6 presents an overview of the major I/O processing system components and their relationships.

Programming Interfaces

The I/O programming tools are the record management system, VAX-11 RMS, for general purpose file and record processing, and the Queue I/O system services, for direct I/O processing. Table 6-2 summarizes the programming interfaces.

RMS (refer to the Data Management Facilities Section) provides device-independent access to file-structured I/O devices. The most general purpose type of access enables programs to process logical records; 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 optimize special record processing. Users performing their own record blocking address blocks using a virtual block number (which is the number of the block relative to the file being processed) for volume-independent 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. Virtual block addressing is used on Files-11 disk or ANSI magnetic tape volumes. In addition, users with sufficient privilege can perform I/O operations using either logical or physical block addressing for defining their own file structures and accessing methods on disk and magnetic tape volumes.

The I/O system services also provide device-dependent programming of devices not supported by RMS, such as real-time interfaces.

Ancillary Control Processes

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. Typical ACP functions would include creating a directory entry or file, accessing or deaccessing a file, modifying file attributes, and deleting a directory entry or file header. There are three kinds of ACPs provided in the system: Files-11 disk, ANSI magnetic tape, and network communications link.

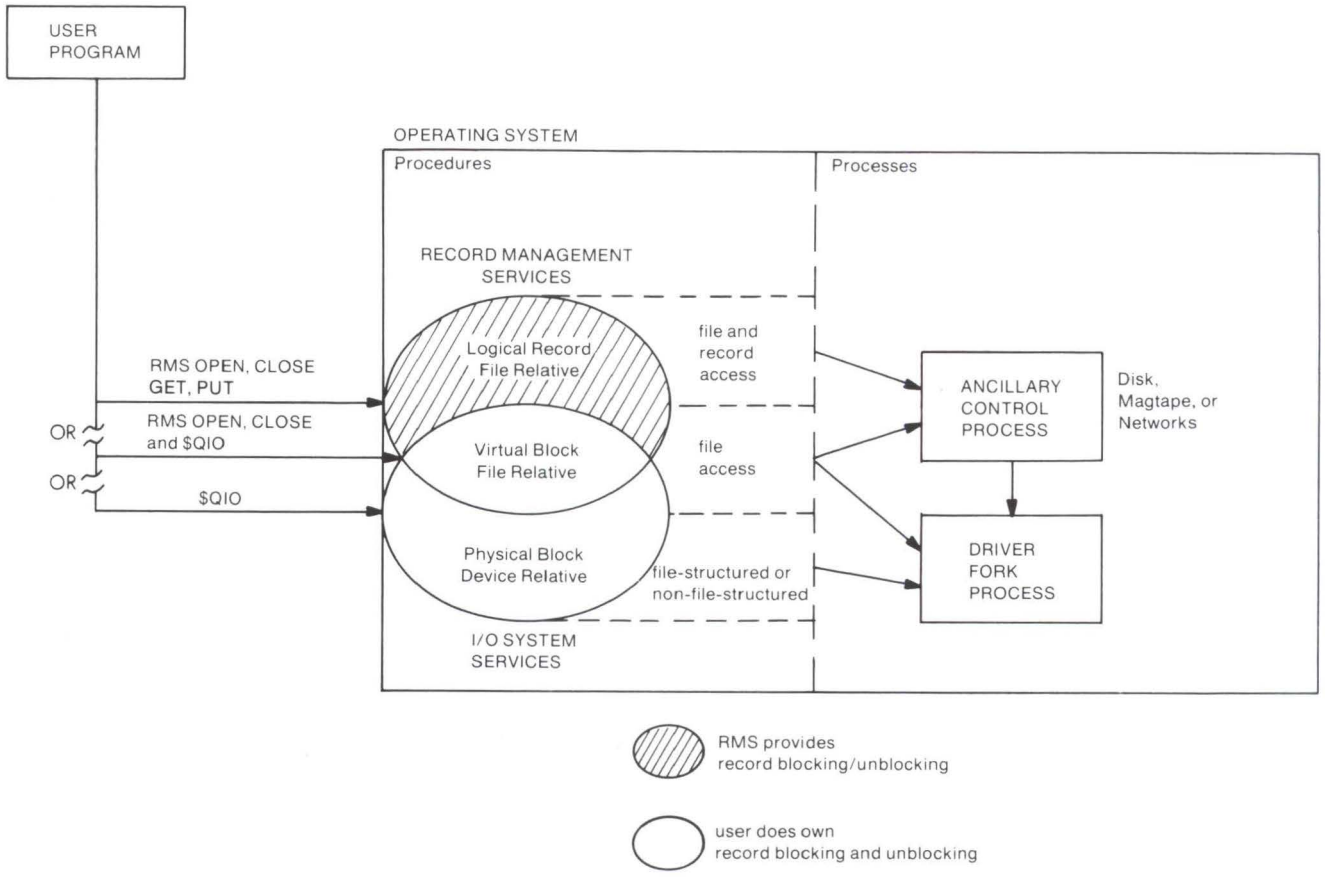


Figure 6-6

User Interfaces to I/O Services

Table 6-2

I/O Processing Interfaces

METHOD	PROGRAM INTERFACE	I/O COMPONENTS	PURPOSE
Record I/O	RMS requests	RMS, ACP and Driver	Use Files-11 disk or ANS magtape file structure, use RMS record access methods
File I/O	RMS OPEN and $QIO requests	RMS for OPEN, ACP and Driver	Use Files-11 disk or ANS magtape file structure, implement own record access methods
Device I/O	$QIO requests	Driver	Fast dumps to disk or magnetic tape, foreign file structure

The RMS and I/O system services programming interfaces are the same regardless of the ACP involved, but since ACPs are particular to 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.

Device Drivers

Once the ACP sets up the information for file-structured I/O requests, a request can be passed on to a device driver. All non-file-structured I/O requests are passed directly to a device driver.

A VAX/VMS 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

Device drivers work in conjunction with the VAX/VMS operating system. The operating system performs all I/O processing that is unaffected by the particular specifications of the target device (i.e., device-independent) processing. 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 (i.e., device-dependent processing). Since different peripheral devices expect different commands and setups, each type of device on VAX/VMS requires its own supporting driver.

The VAX/VMS operating system contains device drivers for a number of standard DIGITAL-supported devices. These include both MASSBUS and UNIBUS devices. In addition, the user can write additional drivers for non-standard UNIBUS devices.

I/O Request Processing

All I/O requests are generated by 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
- keep each disk controller as busy as possible by overlapping seeks with I/O transfers

The processor's many interrupt priority levels improve interrupt response because they enable the software to have the minimum amount of code executing at high priority levels by using low priority levels for code handling request verification and completion notification. In addition, device drivers take advantage of the processor's ability to overlap execution with I/O by enabling processes to execute between the initiation of a request and its completion. User processes can queue requests to a driver at any time, and the driver immediately initiates the next request in its queue upon receiving an I/O completion interrupt.

All access validation and checking takes place before an I/O request is actually queued. For file-structured I/O requests, the Queue I/O Request system service obtains all the virtual block mapping and volume access checking information from the ACP or directly from tables created by the ACP. For example, on virtual block I/O requests for multivolume files, the system service obtains from the ACP's tables the mapping information that 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. Because disk seeks do not require the controller once they are initiated, if a disk driver receives a seek request and the controller is currently busy with an I/O transfer request on some other disk unit, the driver queues the request so that the controller will initiate the seek request before any pending I/O transfers when it has finished the current transfer.

When 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 interrequest idle time. The I/O post routine notifies the process of I/O completion and releases or unlocks buffers.

COMPATIBILITY MODE OPERATING ENVIRONMENT

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:

- the PDP-11 MACRO assembler
- the PDP-11 FORTRAN IV/VAX to RSX compiler
- the PDP-11 BASIC-PLUS-2/VAX compiler
- the RSX-11M program development and file management utilities, including the task builder, text editor, etc.

In addition, the operating system supports the RMS-11 and RMS-11K record management services procedures for compatibility mode programs. Program and data files can therefore be transported between VAX and RSX systems.

The operating system also supports the RSX-11M Monitor Console Routine (MCR) commands, either typed directly on a terminal, or submitted as indirect command files.

User Programming Considerations

Any PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX, or PDP-11 MACRO program can be executed in compatibility mode, provided that it is first linked by the RSX-11M Version 3.2 task builder and that the resulting task image meets the following requirements:

- it must not execute PDP-11 privileged instructions
- it must have been built for a mapped system
- it must not depend on 32-word memory granularity
- it must not use the privileges that enable it to map into the executive or I/O page
- it must not use the PLAS (program logical address space) executive directives
- it must not rely on environmental features of RSX-11M that VAX/VMS does not support, e.g., partitioning or significant events
- it must not use DECnet

The task can be privileged to issue directives other than memory management directives—direct volume access using the QIO request executive directive, for example. IAS or RSX-11D tasks that meet these requirements can also be executed. They must first be built with the RSX-11M Version 3.2 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.

For most RSX-11M executive directives, the native mode operating system executes a functionally equivalent system service. In most cases, the system service duplicates the function. 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 a channel assignment for the appropriate device.

In some cases the operating system cannot duplicate the function, but it does what it can to let a program continue. For example:

- A task image is allowed to declare a significant event, but the directive is ignored.
- A set priority directive is ignored, since the scheduling priority ranges are different. To run at a given priority, the image must be run in the context of a process given that priority.

For the most part, however, many RSX-11M and VAX/VMS program environment characteristics correspond. For example, 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.

File System and Data Management

Both RSX-11M and VAX/VMS recognize User Identification Codes as a protection mechanism. UICs provide the default user file directory in RSX-11M systems, while, in VAX/VMS, a UIC is not necessarily associated with an account name or default directory name. UIC-based file protection, however, is much the same in both systems. That is, it is used in determining read, write, and delete privileges for system, owner, group, and world.

Tasks may use any of the RSX data management services including File Control Services (FCS), RMS-11, and RMS-11K. Special versions of FCS and RMS-11/RMS-11K are supplied with VAX/VMS. A compatibility mode task built on VAX/VMS is thus provided with the full file naming capabilities of VAX/VMS, including logical names and multilevel directories. However, update of a file by multiple tasks is not supported.

Both magnetic tape and Files-11 disk volumes can be transported between systems. VAX/VMS can read and write both Files-11 Level 1 disk structures (ODS-1) and the Level 2 disk structures (ODS-2). The Extend access protection field in ODS-1 is used for Execute access protection in ODS-2. While reading files stored on ODS-1 volumes, therefore, this protection field is ignored.

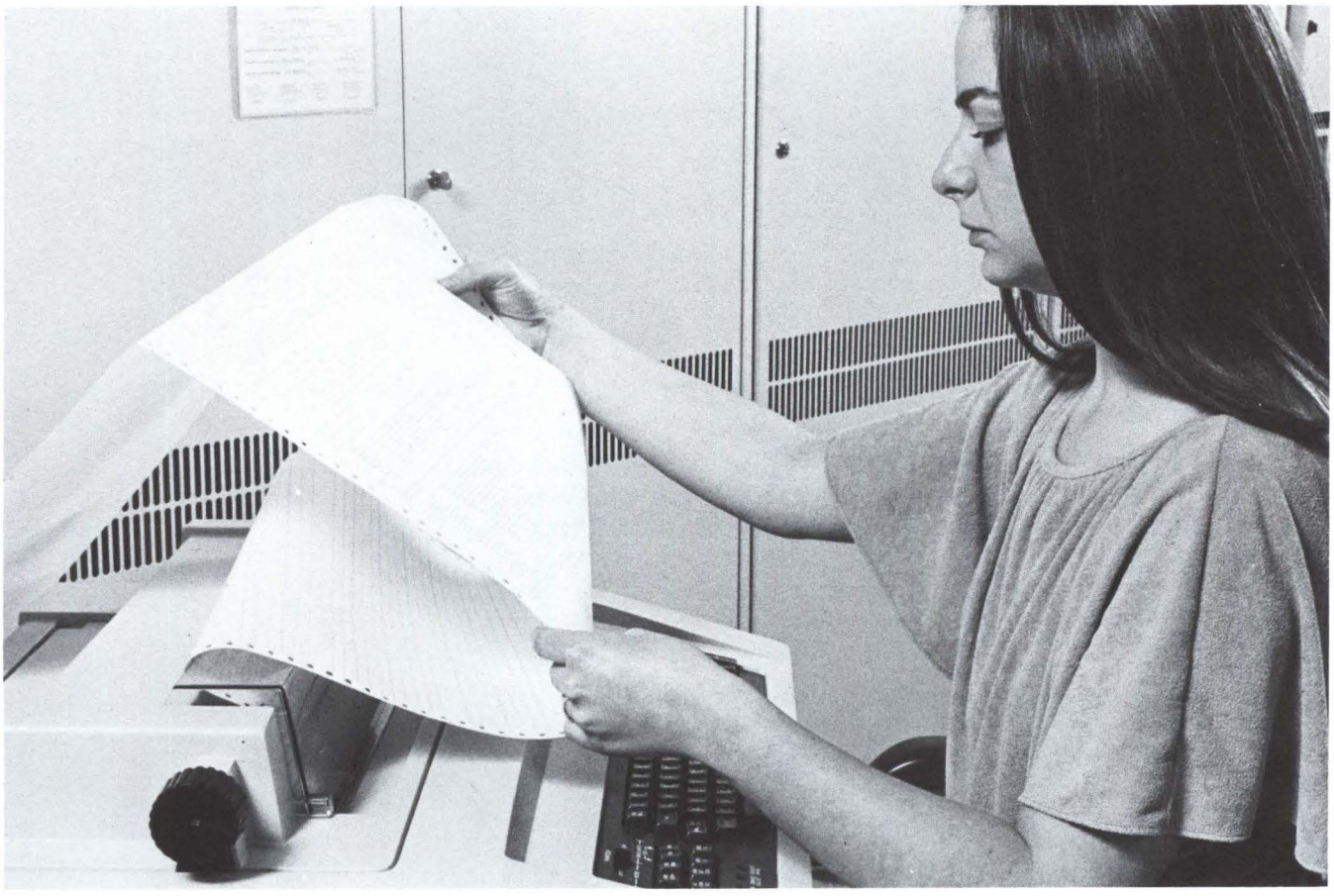
Command Languages

VAX/VMS users can select the MCR command interpreter, which allows them to execute a language (VAX/VMS MCR) that is similar to the RSX-11M MCR command language. Selecting the MCR command interpreter allows the VAX/VMS user to perform the following:

- Run RSX-11M images and VAX-11 images.
- Use RSX-11M components for RSX-11M program development, for example, MACRO-11 or the task builder.
- Use VAX/VMS components for native program development, for example, VAX-11 MACRO or the linker.
- Execute RSX-11M indirect command files. (The VAX/VMS user can use this facility to execute those files required for RSX-11M or RSX-11S system generation.)

VAX/VMS will associate the MCR command interpreter with a process, if "MCR" is the default command interpreter named in the user's authorization file entry or if the user specifies /CLI=MCR following "username" in the LOGIN statement (overriding the default).

7
The Languages



VAX/VMS includes a complete program development environment for a wide range of languages. In addition to the native assembly language, VAX/VMS offers many optional high-level programming languages commonly used in developing both scientific and commercial applications: FORTRAN, COBOL, BASIC, PL/I, PASCAL, CORAL 66, and BLISS-32. It provides the tools necessary to write, assemble or compile, and link programs, as well as build libraries of source, object, and image modules.

Programmers can use the system for development while production is in progress. They can interact with the system on-line, execute command procedures, or submit command procedures as batch jobs. Novice programmers can learn the system quickly because the command language accepts standard defaults for invoking the editors, compilers, and linker. Experienced programmers will appreciate the flexibility and control each tool offers.

INTRODUCTION

VAX/VMS provides a complete program development environment. In addition to the assembly language, MACRO, it offers the optional higher level languages commonly needed in engineering and scientific, commercial, instructional, and implementation applications—FORTRAN, COBOL, BASIC, PL/I, PASCAL, CORAL 66, and BLISS-32. 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.

These native mode language processors produce native object code, and take advantage of the native instruction set and 32-bit architecture of the VAX hardware.

In addition, there is the host development mode programming environment which provides support for PDP-11 BASIC-PLUS-2/VAX, PDP-11 FORTRAN IV/VAX to RSX, and MACRO-11. These produce compatibility mode object code.

VAX COMMON LANGUAGE ENVIRONMENT

An important feature provided by VAX is a "common language" environment, i.e., the VAX languages adhere to a specific set of standards, including:

- symbolic debugger interface
- use of the symbolic traceback facility
- use of the Common Run Time library
- conformance to the VAX calling standard which allows calls among any set of VAX languages, to VAX/VMS system services and to SORT and FMS subroutines
- common handling of exceptions
- use of VAX-11 RMS for record handling

Symbolic Debugger Interface

VAX/VMS 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.

Symbolic Traceback Facility

VAX/VMS supports the Symbolic Traceback Facility. This is a run time facility that aids programmers in finding errors by describing the call sequences that occurred prior to the error. The traceback facility is automatic and does not require that any special qualifiers be included with the FORTRAN or LINK commands (but it can be suppressed by specifying NOTRACE with the LINK command).

When an error condition is detected, the error message is displayed by the run time library indicating the nature of the error and the address at which the error occurred (user PC). This is followed by the traceback information, which is presented in inverse order to the calls. For each call frame, traceback lists module name, routine name, source program line, and absolute and relative PC. Using this information, the programmer can usually locate the source of the error in a relatively short period of time.

Common Run Time Library

The VAX-11 Common Run Time Procedure Library contains sets of general purpose and language-specific procedures. User programs call these procedures to perform specific tasks required for program execution. Both VAX-11 MACRO and native mode high-level language programmers can use any of the Run Time Library procedures in any combination. Because all procedures follow the same programming standards and make no conflicting execution assumptions, a language-independent common run time environment is provided for user programs. Such an environment encourages a user program to be composed of procedures written in different languages, and thus increases programming flexibility.

VAX Calling Standard

The VAX-11 procedure calling standard defines and supports the mechanisms for passing arguments between modules of major VAX-11 software subsystems such as languages, VAX-11 RMS, and the VAX/VMS operating system. The standard facilitates the calling of a procedure written in one language from a program written in another language.

Exception Handling

The mechanisms defined by the VAX-11 calling standard are also used by the condition handling facility to signal the occurence of exceptions detected by hardware or software.

VAX-11 RMS

VAX-11 Record Management Services (RMS) is the technique programmers use to handle record I/O within programs. VAX-11 RMS routines are system routines that provide an efficient and flexible means of handling files and their data. Typically, VAX-11 RMS routines allow the programmer to create a file and:

- accept new input
- read or modify data
- produce output in a meaningful form

High-level language programmers normally use the I/O facilities of their particular language to perform record and file operations. These operations are implemented using the VAX-11 RMS facilities. VAX-11 MACRO programmers can use the VAX-11 RMS routines directly within their programs.

VAX-11 RMS routines are an integral part of the operating system. The programmer need not perform any special linking or declaring of global entry points for the routines.

Furthermore, calls to VAX-11 RMS routines are consistent with the VAX calling standard.

The elements of the common language environment are discussed more fully as they apply to each individual VAX language. Introduced below are each of the VAX-supported languages, their attributes, characteristics, and sample coding.

VAX-11 FORTRAN

Introduction

VAX-11 FORTRAN is an optional language processing system whose language specifications are based on the American National Standard FORTRAN X3.9-1978 (commonly called FORTRAN-77). The VAX-11 FORTRAN compiler supports this standard at the full-language level. At the same time, it provides optional support for certain FORTRAN features based on the previous ANSI standard, X3.9-1966. The VAX-11 FORTRAN compiler performs the following functions:

- produces highly optimized VAX native object code
- makes use of the VAX floating point and character string instructions
- produces shareable code

The VAX-11 FORTRAN language is upwardly compatible with the PDP-11 FORTRAN language. Table 7-1 lists extensions to the ANSI FORTRAN-77 language. Table 7-2 lists the features of FORTRAN-77.

Some characteristics of VAX-11 FORTRAN are described below.

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. The OPEN statement can contain an ERR keyword which specifies 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.

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 provides a concatenation operator, substring notation, CHARACTER relational expressions, and CHARACTER-valued functions. CHARACTER constants, consisting of a string of printable ASCII characters enclosed in string quotes, can be assigned symbolic names using the PARAMETER statement. Operations which use CHARACTER 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).

Figure 7-1 illustrates two VAX-11 FORTRAN subroutines. This figure illustrates the use of the FORTRAN CHARACTER data type and some of the VAX-11 FORTRAN extensions to FORTRAN-77. The first subroutine, which reverses a character string, illustrates CHARACTER declarations (both fixed and passed length), the intrinsic function LEN, substring manipulation, and the ENDDO statement. The second subroutine locates a substring of a character string and marks the starting position of the substring. This subroutine illustrates CHARACTER declarations (both fixed and passed length), assignments of character values to variables, the intrinsic function INDEX, substring manipulation, and the FORTRAN-77 block IF statement.

```
      SUBROUTINE REVERSE(S)
      CHARACTER T, S*(*)

      J = LEN(S)
      DO I=1, J/2
          T = S(I:I)
          S(I:I) = S(J:J)
          S(J:J) = T
          J = J-1
      ENDDO
      END

      SUBROUTINE FIND_SUBSTRINGS(SUB, S)
      CHARACTER*(*) SUB, S
      CHARACTER*132 MARKS

      I = 1
      MARKS = ' '
10    J = INDEX(S(I:), SUB)
      IF (J .NE. 0) THEN
          I = I + (J-1)
          MARKS(I:I) = '#'
          I = I+1
          IF (I .LE. LEN(S)) GO TO 10
          ENDIF
      WRITE(6,91) S, MARKS
91    FORMAT( 2(/1X, A))
      END
```

Figure 7-1
FORTRAN CHARACTER Data Type Program

Table 7-1
Language Extensions to FORTRAN-77, X3.9-1978

VAX-11 FORTRAN	
31-character symbolic names	Symbolic names used to identify programs, subprograms, external functions and subroutines, COMMON blocks, variables, arrays, symbolic constants, and statement functions can be longer than the standard six characters. Symbolic names can include letters, digits, dollar sign, and underscore; however, the first character in name must be a letter.
CALL extensions	Permit interfacing to VAX/VMS system service procedures using the VAX-11 calling standards.
Hexadecimal and octal constants and field descriptors	Both octal and hexadecimal constants can be expressed in DATA statements. No conversion of the defined value (such as sign-extension) is performed. The Z field descriptor in FORMAT statements enables a program to read and write hexadecimal digits which are stored in an internal format in an I/O list element.
DO WHILE/END DO	Structured looping control constructs.
Data initialization in type-declaration statements	Variables can be assigned initial values in type declaration statements.
INTEGER data type defaults	A compiler command specification allows all INTEGER and LOGICAL declarations without explicit length specifications to be considered as INTEGER*2 and LOGICAL*2 or INTEGER*4 or LOGICAL*4, respectively.
VAX-11 FORTRAN and PDP-11 FORTRAN IV-PLUS	
Additional data types and type declaration statements (DOUBLE COMPLEX, COMPLEX*16, and **CHARACTER*n** are VAX-11 FORTRAN only) NOTE Names appearing on the same line above are synonyms. Those in boldface are the ANSI standard ones.	BYTE, LOGICAL*1, LOGICAL*2, **LOGICAL**, LOGICAL*4, INTEGER*2, **INTEGER**, INTEGER*4, **REAL**, REAL*4, **DOUBLE PRECISION**, REAL*8, **COMPLEX**, COMPLEX*8, DOUBLE COMPLEX, COMPLEX*16, **CHARACTER*n**
Indexed File I/O	Extensions are provided to allow FORTRAN language access to RMS ISAM files.
Keyed READ	Key types: INTEGER*2, INTEGER*4, CHARACTER with generic, and approximate key match.
Indexed File WRITE	New records can be written to ISAM files with the write statements.
REWRITE statement	Existing records in ISAM files can be modified with the REWRITE statement.
DELETE statement	Existing records can be deleted from ISAM or relative files with the DELETE statement.
UNLOCK statement	Single-record locking (in the VAX environment) and bucket-level locking (in the PDP-11 environment) for shared file applications involving relative and indexed organization files.
Logical operations on integers	The logical operators .AND., .OR., .NOT., .XOR., and .EQV. may be applied to integer data to perform bit masking and manipulation.
INCLUDE statement	The INCLUDE statement incorporates FORTRAN source text from a separate file into a FORTRAN program.
VAX-11 FORTRAN, PDP-11 FORTRAN IV-PLUS, and PDP-11 FORTRAN IV	
Array subscripts using general expressions of any numeric data type	Any arithmetic expression can be used as an array subscript. If the value of the expression is not an integer, it is converted to integer format.
End-of-Line comments	Any FORTRAN statement can be followed, in the same line, by a comment that begins with an exclamation point.
Conditional compilation of debugging statements	Statements that are included in a program for debugging purposes can be so designated by the letter D in column 1. Those statements are compiled only when the associated compiler command option is set. They are treated as comments otherwise.
Default FORMAT width	The programmer can specify input or output formatting by type and default width and precision values will be supplied.

Table 7-2
FORTRAN-77 Features

VAX-11 FORTRAN	
Additional data types	The data type INTEGER*4 provides a sign plus 31 bits of precision. INTEGER*4 allows a greater range of values to be represented than INTEGER*2. Both data types can be used in the same program.
Additional I/O statements	READ (u'r,fmt) and WRITE (u'r,fmt) provide input and output to direct access files.
DO control variable data types	The control variable of a DO statement can be a REAL or DOUBLE PRECISION variable, as well as an INTEGER*2 or INTEGER*4 variable. The initial, terminal, and increment parameters can be of any data type and are converted before use to the type of the control variable if necessary.
Additional data type	The data type CHARACTER permits manipulation of strings of ASCII characters expressed as constants, variables, arrays, substrings, symbolic names, or functions.
IF THEN ELSE statements	The FORTRAN-77 block-IF statements are provided: IF, ELSE IF, ELSE, and ENDIF. These structured programming statements provide more readable and reliable methods for expressing conditional statement execution.
Standard CALL facility	Provides standard argument definitions for called procedures.
VAX-11 FORTRAN and PDP-11 FORTRAN IV-PLUS	
ENTRY statement	ENTRY statements can be used in SUBROUTINE and FUNCTION subprograms to define multiple entry points in a single program unit.
PARAMETER statement	PARAMETER statements can be used to give symbolic names to constants.
Generic function selection	Function selection by argument data type is provided for many FORTRAN library functions.
Array dimension bounds	Lower bounds as well as upper bounds of the array dimension can be specified in array declarators. The value of the lower bound dimension declarator can be negative, zero or positive.
List-Directed I/O statements	The READ (u,*), WRITE (u,*), TYPE*, ACCEPT*, and PRINT* statements provide list-directed, or "free format," I/O without requiring a FORMAT specification.
Additional I/O statements	OPEN and CLOSE statements provide file control and attribute definition. ACCEPT, TYPE, and PRINT statements provide device-oriented I/O. ENCODE and DECODE statements provide memory-to-memory formatting. DEFINE FILE, READ (u'r), WRITE (u'r), and FIND (u'r) provide unformatted direct access I/O, which allows the FORTRAN programmer to read and write files written in any format.
End-of-file or Error Condition transfer	The specifications END=n and ERR=n (where n is a statement label) can be included in any READ or WRITE statement to transfer control to the specified statement upon detection of an end-of-file or error condition. The ERR=n option is also permitted in the ENCODE and DECODE statements, allowing program control of data format errors.
Additional data type	The byte data type (keyword LOGICAL*1 or BYTE) is useful for storing small integer values as well as for storing and manipulating character information.
IMPLICIT declaration	The IMPLICIT statement has been added to redefine the implied data type of symbolic names.

DO loop iteration count	The terminal and increment parameters can be modified within a DO loop without affecting the iteration count. The number of times a DO loop is executed is determined at the initialization of the DO statement and is not re-evaluated during successive executions of the loop. Consequently, the number of times the loop is executed will not be affected by changing the variables used in the DO statement.

VAX-11 FORTRAN, PDP-11 FORTRAN IV-PLUS, and PDP-11 FORTRAN IV

Array dimensions	Arrays can have up to seven dimensions.
Character literals	Character strings bounded by apostrophes can be used in place of Hollerith constants.
Mixed-mode expressions	Mixed-mode expressions can contain any data type, including complex and byte.
General expression DO and GO TO parameters	General expressions are permitted for the initial value, increment, and limit parameters in the DO statement, and as the control parameter in the computed GO TO statement.
DO increment parameter	The value of the DO statement increment parameter can be negative.
Optional statement label list	The statement label list is an assigned GO TO is optional.
General expressions in I/O lists	General expressions are permitted in I/O lists of WRITE, TYPE, and PRINT statements.

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 subroutines written in any other VAX language, and also system services, using the VAX-11 procedure calling standard. Special operators exist for passing arguments by immediate value, by reference, or by descriptor. A special operator also exists for obtaining the location of argument values used by the system services procedures.

Shareable Programs

The FORTRAN language can be used to create shareable 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 nonstandard syntax; or when it corrects a syntactically incorrect statement. The message indicates the existence of possible trouble in executing the program.

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. Figure 7-2 is an illustration of VAX-11 FORTRAN code and its equivalent VAX-11 MACRO 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 nonoptimized 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

```
0001            SUBROUTINE RELAX2(EPS)
0002            PARAMETER M=40, N=60
0003            DIMENSION X(0:M,0:N)
0004            COMMON X
0005            LOGICAL DONE
0006     1      DONE = .TRUE.
0007            DO 10 J = 1,N-1
0008            DO 10 I = 1,M-1
0009                    XNEW = ( X(I-1,J)+X(I+1,J)+X(I,J-1)+X(I,J+1) )/4
0010                    IF ( ABS(XNEW-X(I,J)) .GT. EPS ) DONE = .FALSE.
0011     10             X(I,J) = XNEW
0012            IF (.NOT. DONE) GO TO 1
0013            RETURN
0014            END
```



```
                        .TITLE  RELAX2
                        .IDENT  01
0000                    .PSECT  $BLANK
0000    X:
0000                    .PSECT  $CODE
0000    RELAX2::
0000                    .WORD   ^M<IV,R5,R6,R7,R8,R9,R10,R11>
0002                    MOVAL   $LOCAL, R11
                                                        ; 0006
0009    .1:
0009                    MNEGL   #1, DONE (R11)
                                                        ; 0007
000C                    MOVL    #1, R6
000F                    MOVAL   $BLANK, R5
0016    L$1:
                                                        ; 0008
0016                    MOVL    #1, R9
0019                    MULL3   #41, R6, R7
001D    L$2:
                                                        ; 0009
001D                    ADDL3   R9, R7, R10
0021                    ADDF3   X+4(R5)(R10), X-4(R5)[R10], R0
0029                    ADDF2   X-164(R5)[R10], R0
002F                    ADDF2   X+164(R5)[R10], R0
0035                    MULF3   #^X3F80, R0, R8
                                                        ; 0010
003D                    SUBF3   X(R5)[R10], R8, R0
0042                    BICW2   #^X8000, R0
0047                    CMPF    R0, ^EPS(AP)
004B                    BLEQ    L$3
004D                    CLRL    DONE(R11)
004F    L$3:
                                                        ; 0011
004F                    MOVL    R8, X(R5)[R10]
0053                    AOBLEQ  #39, R9, L$2
0057                    AOBLEQ  #59, R6, L$1
005B                    MOVL    R6, J(R11)
005F                    MOVL    R8, XNEW(R11)
0063                    MOVL    R9, I(R11)
                                                        ; 0012
0067                    BLBC    DONE(R11), .1
                                                        ; 0013
006A                    RET
                        .END
```

Page 1 above illustrates, as a VAX-11 FORTRAN subroutine, a relaxation function often found in engineering applications. This particular example is a planar (2-dimensional) function that can be used to obtain the values of a variable at coordinates on a surface, for instance, temperatures distributed across a metal plate. The algorithm illustrated here locates the array element values relative to a given point in the plane.

Page 2 contains the equivalent VAX-11 MACRO assembly code for this VAX-11 FORTRAN subroutine. The line numbers in the comment just to the left of this paragraph refer to the lines in the VAX-11 FORTRAN subroutine listing above. Several VAX-11 FORTRAN compiler optimizations are illustrated, including global and local register assignment, removal of invariant computations from the DO loop, recognition of common subexpressions, branch instruction optimizations, in-line ABS function, and peephole optimization.

*The code for lines 7 and 8 contains the global register assignments for the function. The multiply statement just preceding the code for line 9 is an invariant computation (J*41) removed from the DO loop. DO loop control is provided by the Add One and Branch Less Than or Equal (AOBLEQ) instructions in the code for line 11.*

The code for line 9 evaluates the common subexpression for the computation. The code contains a local register assignment (R10), and uses 2- and 3-operand instructions and context switching ([R10]) to calculate an array element value. The last instruction for line 9 is a peephole optimization that increases execution speed by using a "multiply by .25" in place of the FORTRAN statement's "divide by 4."

Figure 7-2

VAX-11 FORTRAN Program

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.

Debugging Facilities

VAX-11 FORTRAN debugging facilities include diagnostic messages, conditional compilation flags, and access to the VAX/VMS DEBUG program. The DEBUG program lets the programmer set breakpoints and trace points, and examine and modify the contents of locations dynamically when executing the program.

DEBUG understands FORTRAN data type representations and syntax. It can examine and deposit locations using floating point representation, and it can reference FORTRAN symbols, statement labels, and line numbers symbolically. It can also reference arrays symbolically, for example:

```
EXAMINE A(I,J+3)
```

When debugging VAX-11 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.

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. FORTRAN provides a facility for conditionally compiling such source lines so that they can be compiled during the development stage but treated as comments once the program has been debugged.

Symbolic Traceback

Figure 7-3 illustrates a source VAX-11 FORTRAN program and the symbolic traceback facility supported by VAX/VMS. (Note that some of the entries in the list show relative and absolute PC but no corresponding values for module name and routine name; this indicates that the values refer to procedure calls internal to the run time library.)

```
0001          I=1
0002          CONTINUE
0003          J=2
0004          CONTINUE
0005          K=3
0006          CALL SUB1
0007          CONTINUE
0008          END

0001          SUBROUTINE SUB1
0002          I=1
0003          J=2
0004          CALL SUB2
0005          END

0001          SUBROUTINE SUB2
0002          COMPLEX W
0003          COMPLEX Z
0004          DATAW/(0.,0.)/
0005          Z = LOG(W)
0006          END
%MTH-F-INVARGMAT, invalid argument to math library
     user PC 00000449
%TRACE-F-TRACEBACK, symbolic stack dump follows
```

module name	routine name	line	relative PC	absolute PC
			0000074C	0000074C
			0000081C	0000081C
	SUB2	5	00000011	00000449
	SUB1	4	00000017	00000437
	T1$MAIN	6	0000001B	0000041B

Figure 7-3
FORTRAN Symbolic Traceback

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 its predecessor, VAX-11 COBOL-74, 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 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.

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 bidirectional 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 Interprogram 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 standard data types are:

- Numeric DISPLAY Data
  - Trailing overpunch sign
  - Leading overpunch sign
  - Trailing separate sign
  - Leading separate sign
  - Unsigned
  - Numeric-edited
- Numeric COMPUTATIONAL Data
  - Word fixed binary
  - Longword fixed binary
  - Quadword fixed binary
- Packed-Decimal Data (COMPUTATIONAL-3)
  - Unsigned packed decimal
  - Signed packed decimal
- Floating Point Data
  - F_floating (COMPUTATIONAL-1)
  - D_floating (COMPUTATIONAL-2)
- Alphanumeric DISPLAY Data
  - Alphanumeric
  - Alphabetic
  - Alphanumeric-edited

As indicated previously, VAX-11 COBOL supports the COMP-3 (packed decimal) data type (two decimal digits per byte). This data type offers the following advantages:

- disk storage savings
- faster arithmetic operations than standard numeric display data type
- compatibility with and migration from other COBOL vendors

Figure 7-4 illustrates a record definition of a typical payroll master file application in which the COMP-3 data type is frequently used. In this record definition, all numeric fields on which arithmetic operations are performed are defined to be the COMP-3 data type.

Figure 7-5 illustrates a sample calculation of one such COMP-3 data item in the record. Here, the year-to-date net pay is calculated as a function of the gross pay, and all voluntary and involuntary deductions to date.

Infrequently, commercial applications arise in which the utilization of floating point data (COMP-1 and COMP-2) is useful. For example, a large corporation may want to survey its customers regarding its product quality. The corporation wishes to select a statistically valid sample of its customer base without going to the expense of contacting each and every customer. Hence, it is necessary to randomly sample its customer base; a random number generator is used to select those customers to be sampled.

Figure 7-6 illustrates a COBOL program fragment in which a CALL to the VAX-11 run-time procedure library routine MTH$RANDOM is made to generate random numbers. This routine returns a random number in COMP-1 (F_floating) data type representation in the range from 0.0 to 1.0. Such numbers are then integerized and subsequently used to select those customers to be sampled in the product quality survey.

The COMP-2 data type may be used in similar commercial applications.

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,

```
FD      PAYROLL-MASTER
          LABEL RECORDS ARE STANDARD.
01      PAYROLL-REC.
          02  EMPLOYEE-NAME                PIC X(30).
          02  EMPLOYEE-ID                  PIC 9(9)          USAGE IS DISPLAY.
          02  YTD-GROSS-PAY                PIC 9(5)V99       USAGE IS COMP-3.
          02  YTD-FED-WITHHOLD-TAX         PIC 9(5) V99      USAGE IS COMP-3.
          02  YTD-FICA                     PIC 9(4)V99       USAGE IS COMP-3.
          02  YTD-STATE-WITHHOLD-TAX       PIC 9(5)V99       USAGE IS COMP-3.
          02  YTD-LOCAL-WITHHOLD-TAX       PIC 9(5)V99       USAGE IS COMP-3.
          02  YTD-VOLUNTARY-DEDUCTIONS     PIC 9(4)V99       USAGE IS COMP-3.
          02  YTD-NET-PAY                  PIC 9(5)V99       USAGE IS COMP-3.
```

Figure 7-4
COMP-3 Record Definition

```
•
•
•
SUBTRACT YTD-FED-WITHHOLD-TAX,
         YTD-FICA
         YTD-STATE-WITHHOLD-TAX,
         YTD-LOCAL-WITHHOLD-TAX,
         YTD-VOLUNTARY-DEDUCTIONS
         FROM YTD-GROSS-PAY
         GIVING YTD-NET-PAY.
•
•
•
```

Figure 7-5
Arithmetic on COMP-3 Data Type

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.

SORT/MERGE Facility

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.

Figure 7-7 illustrates how to sort a file with the USING and GIVING phrases of the SORT statement. The fields to be sorted are S-KEY-1 and S-KEY-2; they contain account numbers and amounts. The sort sequence is amount within account number. Notice that OUTPUT-FILE is a relative file.

In Appendix B, the sample program is merging three identically sequenced regional sales files into one total sales file. The program adds sales amounts and writes one record for each product-code.

Symbolic Characters Facility

It is often useful for the programmer to be able to construct on a video terminal, the image of a form similar to a printed form. This process involves imbedded or non-printing characters (i.e., line feed, carriage return, escape key, etc.). VAX-11 COBOL provides the user with the ability to include, within the COBOL code, non-printing control characters. Essentially, these characters control the position of the cursor during an interactive session utilizing a video terminal (i.e., VT52, VT100, etc.).

Figure 7-8 illustrates a sample data entry form used as a prompt for data input.

The VAX-11 COBOL code used to generate this particular form is listed in Appendix C. The sample code is used in conjunction with the VT100 terminal. Code for the VT52 is similar.

```
01      RAND-NUM                                USAGE IS COMP-1.
01      RAND-CUST-NUM           PIC 9(7).
01      CUST-NUM REDEFINES RAND-CUST-NUM.
          02 DISTRICT            PIC 9(2).
          02 WITHIN-DIST         PIC 9(5).
01      SEED                    PIC 9(8)        USAGE IS COMP.
•
•
•
          CALL "MTH$RANDOM"             USING SEED
                                        GIVING RAND-NUM.
          COMPUTE RAND-CUST-NUM ROUNDED = RAND-NUM * 10000000.
•
•
•
```

Figure 7-6
Example of COMP-1 Data Type

```
IDENTIFICATION DIVISION.
PROGRAM-ID.        SORT EXAMPLE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.                    VAX-11.
OBJECT-COMPUTER.                    VAX-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT INPUT-FILE ASSIGN TO "INPFIL".
        SELECT OUTPUT-FILE ASSIGN TO "OUTFIL"
                        ORGANIZATION IS RELATIVE.
        SELECT SORT-FILE ASSIGN TO "SRIFIL".
DATA DIVISION.
FILE SECTION.
SD      SORT-FILE.
01      SORT-REC.
        03   S-KEY-1.
                        05   S-ACCOUNT-NUM    PIC X(8).

        03   FILLER    PIC X(32).
        03   S-KEY-2.
                        05   S-AMOUNT         PIC S9(5)V99.
        03   FILLER    PIC X(53).
FD      INPUT-FILE
        LABEL RECORDS ARE STANDARD.
01      IN-REC         PIC X(100).
FD      OUTPUT-FILE
        LABEL RECORDS ARE STANDARD. PIC X(100).
01      OUT-REC        PIC X(100).
PROCEDURE DIVISION.
000-00-THE-SORT.
                SORT SORT-FILE ON ASCENDING KEY
                                S-KEY-1
                                S-KEY-2
                WITH DUPLICATES IN ORDER
                USING INPUT-FILE GIVING OUTPUT-FILE.
        DISPLAY "END OF PROGRAM SORT EXAMPLE".
        STOP RUN.
```

Figure 7-7
Sample SORT Code

```
CUSTOMER NUMBER:12345678
CUSTOMER NAME:ROLAND J. JONES-----
CUSTOMER ADDRESS:747 FIRST AVE.-----
CITY:ANYTOWN------STATE:NH ZIP:03061
```

Figure 7-8
Video Form

CALL Facility

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. Figure 7-9 illustrates a sample program utilizing all three types of argument passing mechanisms.

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).

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

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLTST2.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.    VAX-11.
OBJECT-COMPUTER.    VAX-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 RETURN-VALUE                              PIC 9(9)USAGE IS COMP
                                             VALUE IS 999999999.

01 D-RETURN-VALUE                            PIC 9(9) VALUE IS 999999999.
PROCEDURE DIVISION.
PO.

DISPLAY "CALL SYS$ASCTIM".
CALL "SYS$ASCTIM"
        USING
                BY REFERENCE TIMLEN
                BY DESCRIPTOR TIMBUF
                BY VALUE ZERO
                BY VALUE ZERO
        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 7-9
System Services Call

module name, routine name, and source line number in which either an invocation to another user routine occurs or the fatal error itself occurs.

As an example of the traceback facility, Figure 7-10 illustrates the printing of error messages and the subsequent traceback for a COBOL module in which an I/O error occurs at run time. Specifically, a COBOL OPEN statement failed because the file "DB2:[COBOL]MASTERFIL.DAT" was not found on the OPEN operation. The "module name" and "routine name" fields (of the traceback) identify the entry point, IOERRTEST, into the COBOL module. The OPEN failure occurs on line number 22 of the source module. The "relative PC" field specifies that the OPEN failure corrrspondingly occurs at "67" hexadecimal bytes into the object code relative to the entry point IOERRTEST. The "absolute PC" field also specifies that the OPEN failure occurs at absolute location "667" in the executable image containing IOERRTEST.

Additionally, the user can request a complete explanation of the OPEN error by interrogating the system interactively with the command "HELP ERRORS COB FILNOTFOU". This VAX/VMS command displays the information shown in Figure 7-11.

Thus, the issuance of specific, English-like error messages coupled with the traceback facility and interactive interrogation of the system to explain completely the run-time error offers the user a powerful debugging tool in identifying programming errors.

Also, the VAX-11 COBOL debugging facilities provide access to the VAX/VMS SYMBOLIC DEBUGGER. The SYMBOLIC DEBUGGER lets the programmer set breakpoints, and examine and modify the contents of locations dynamically while 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 differences 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

```
%COB-F-FILNOTFOU, file _D32:[COBOL]MASTERFIL.DAT; not found on OPEN
-RMS-E-FNF, file not found
%TRACE-F-TRACEBACK, symbol stack dump follows
```

module name	routine name	line	relative PC	absolute PC
IOERRTEST	IOERRTEST	22	00000067	00000667

Figure 7-10
Example of Traceback Facility

- different methods of specifying file optimization attributes

Fortunately, most differences are transparent to the programmer, and moving programs from PDP-11 COBOL or VAX-11 COBOL-74 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.

```
ERRORS
    COB
        FILNOTFOU
            file not found on OPEN
            Explanation: The named file was not
            found during the execution of the open
            statement. The file status variable, if pre-
            sent, has been set to 97. No applicable
            USE procedure has been found.
            User Action: The user should examine
            the referenced directory to check for the
            existence of the named file. Another
            common source of this error is a mistake
            in spelling the file specification for the
            file.
```

Figure 7-11
Interactive Explanation of Error

Additional Features

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

- Subscripts can be arithmetic expressions.
- Subscripting and indexing are interchangeable.
- The CONTINUE statement is included. It transfers control to the next executable statement and can replace conditional or imperative statements.
- The AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs are included.
- INITIAL clause on the Program-ID is included.
- User-defined alphabets are included.
- PADDING CHARACTER is supported in the FILE-CONTROL paragraph.
- VALUE OF clause is included.
- Delimited scope statements are included (e.g., END-ADD, END-IF).
- All arithmetic statements with overlapping operands function as if the operands did not overlap except for operands specified in LINKAGE SECTION or as EXTERNAL.
- ALTER statement is included.
- CALL data-name is included. Both ON OVERFLOW and EXCEPTION are supported.
- CANCEL statement is fully implemented.
- INITIALIZE statement is fully implemented.
- INSPECT statement is fully implemented including combined TALLYING and REPLACING format.
- SET statement supporting mnemonic-names and condition-names is included.
- Independent segments (segments 50 and above) of the SEGMENTATION module are included.
- WRITE advancing mnemonic-name and associated SPECIAL NAMES C01 is included.
- Use of source file libraries by the COPY statement is 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.

Sample VAX-11 COBOL Code

This sample VAX-11 COBOL code demonstrates some of

the powerful language elements of VAX-11 COBOL. 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 7-12, 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 "CUSTOM.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 "STATEM.REP"
           FILE STATUS IS
                STATEMENT-REPORT-STATUS.
```

Figure 7-12
File Description

In Figure 7-13, 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.

```
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-STORAGE.
IF OPTIONS-AREA = SPACES
           DISPLAY "Discrepancy Report Only"
           GO TO CONFIRM-OPTIONS.
MOVE 0 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".
```

Figure 7-13
Procedure Division Using Interactive COBOL Verbs

```
IF OPTION-STORAGE = ALL ZERO
           DISPLAY "No options recognized"
           STOP RUN.
DISPLAY "Selected options:".
IF WANT-STATEMENTS
           DISPLAY "Statements:".
IF WANT-INVOICES
DISPLAY "Invoices"
```

Figure 7-13 (Con't)
Procedure Division Using Interactive COBOL Verbs

Figure 7-14 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.

```
OPEN INPUT CUSTOMER-FILE.
MOVE "000000" TO CUST-CUST-NUMBER.
START CUSTOMER-FILE
                    KEY IS > CUST-CUST-NUMBER.
OPEN OUTPUT STATEMENT-REPORT.

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

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

Figure 7-14
Random to Sequential Access

VAX-11 BASIC

Introduction

The new BASIC product gives the VAX user all the benefits of a highly interactive programming environment and high-performance development language. It combines the best features of PDP-11 BASIC-PLUS-2 and RSTS/E BASIC-PLUS with the significant performance and addressing benefits provided by a native mode VAX language that is fully integrated with the VMS environment.

VAX-11 BASIC is a highly extended implementation language. It provides powerful mathematic and string handling facilities, support for symbolic characters, and full RMS indexed, sequential, and relative I/O operations. There does not yet exist an ANSI standard comparable to this level of BASIC.

VAX-11 BASIC can be used as though it were either an interpreter or a compiler. A fast RUN command and support

for direct execution of unnumbered statements (immediate mode) gives the VAX-11 BASIC user the "feel" of an interpreter. However, this product can also be used in a compilation mode, where it generates native-mode object modules like the other VAX compilers. In either case, VAX-11 BASIC generates optimized VAX native-mode instructions which have extremely fast execution times. Typical compilation speeds are up to 3,000 lines per minute and computations will generally execute up to five times faster than the same programs on a PDP-11.

Following is a brief overview of the general characteristics of VAX-11 BASIC.

General Characteristics

VAX-11 BASIC generates in-line native VAX-11 instructions in both its RUN and its compilation modes. The code produced takes advantage of VAX/VMS native mode capabilities, including:

- direct calls on operating system service routines, even in immediate mode
- transparent access to DECnet
- direct calls to the Common Run Time Library and standard system utilities, including VAX-11 SORT/MERGE
- direct calls to separately compiled native mode procedures written in any language that utilizes the VAX procedure calling standard
- program sizes up to 2 billion bytes are allowed
- all modules are position-independent (PIC) and can be run as fully re-entrant code
- the VAX-11 DEBUG facility has full support for VAX-11 BASIC

The code generated by VAX-11 BASIC uses the standard VAX/VMS traceback facility for determining the source of run-time errors. If a fatal program error should occur, an English message is printed identifying the module and line number where the error occurred. The English text, the traceback, and the integrated BASIC HELP utility provide a powerful program debugging environment.

Object modules produced by VAX-11 BASIC can be linked with native mode modules produced by other language processors including BLISS, COBOL, FORTRAN, PASCAL, and MACRO.

Structured Programming

Structured programming adds some of the features of a block structured language (such as PASCAL) to BASIC to allow complex programs to be written without recourse to subroutines or obscure programming techniques. This makes programs easier to write and maintain.

Figure 7-15 below illustrates a record defined by a MAP statement, successive retrievals by the use of a GET statement, and iteration controlled by a WHILE...NEXT statement block.

The SUBPROGRAM and FUNCTION constructs in VAX-11 BASIC have structured END and EXIT statements. In addition, BASIC allows the use of statement modifiers which allow conditional or repetetive execution of the statement without requiring the user to construct artificial loops or block constructs. Any non-declarative statement in VAX-11 BASIC can have one or more statement modifiers. The BASIC statement modifiers include FOR, IF, UNLESS, UNTIL, and WHILE constructs. Each of the statements in Figure 7-16 illustrates the use of a statement modifier.

Data Types

VAX-11 BASIC increases the number of data types available to the BASIC programmer by allowing the use of 32-bit integer and 64-bit floating point data values. Tabel 7-3 below describes the data type supported by VAX-11 BASIC.

Table 7-3 Data Types

Data Type	Meaning
REAL	Specifies that the variable or constant contains floating-point data. The precision depends on the COMPILE command qualifier you use. COMPILE/SINGLE specifies 32-bit floating point numbers; COMPILE/DOUBLE specifies 64-bit floating point numbers.
WORD	Specifies that the variable or constant contains word-length integer data, regardless of the COMPILE command qualifier you use.
LONG	Specifies that the variable or constant contains longword integer data, regardless of the COMPILE command qualifier you use.
INTEGER	Specifies that the variable or constant contains integer data. This data type defaults to the qualifier used at compile-time. If you compile the program with the /WORD qualifier, integers are 16 bits long; with the /LONG qualifier, 32 bits long.
STRING	Specifies that the variable or array contains string data.

Declarations

VAX-11 BASIC allows inplicit declaration of variables. Unless specifically named in a declaration statement, a variable will be declared by its first reference. The PDP-11 BASIC-PLUS-2 convention is to implicitly type a variable or value by the trailing character in its representation, e.g. ABC$ is a STRING variable; XYZ% and 123% are INTEGER; T12, 314159, and 3.14 are implicitly REAL.

Variables can be declared in COMMON, MAP, or DECLARE statements. Both COMMON and MAP statements are used to declare static storage areas—typically I/O records or shared data blocks. If a program has several named common statements with the same name, the common program sections (PSECTs) are stored one after the other. If several MAP statements have the same name, they overlay the same PSECT.

The DECLARE statement is used to explicitly type variables, functions, and constants. Note that the appearance of a variable name in a DECLARE statement implies that

```
99      !   ---------------------------------------------                                  &
        !                          EMPLOYEE RECORD DEFINITION(S)                          &
        !                                                                                 &
        !                          LINE 100: THE "GENERAL DEFINITION"                     &
        !                          LINE 200: THE "EXPANDED DEFINITION"                    &
        !
100         MAP (REC1)             STRING EMPLOYEE.RECORD = 36,                           &
                                   REAL RATE,                                             &
                                   INTEGER ENDFLAG
110     !
200         MAP (REC1)             STRING LAST.NAME = 20,                                 &
                                   STRING FIRST.NAME = 12,                                &
                                   STRING MID.INITS = 4,                                  &
                                   REAL FILL,                                             &
                                   INTEGER FILL                                           &
210     !   ---------------------------------------------!
298     !
299     !
300         FILE.NAME.1$ = "EMPLOYEE.DAT"
310     !
320         OPEN FILE.NAME.1$ AS FILE #1,SEQUENTIAL, ACCESS READ, MAP REC1
330     !
400         TOTAL.RATES = 000000.00
410     !
411     !
490     !   --------------  COMPUTE SUM OF RATES IN FILE --------------
498     !
499     !
500         WHILE NOT ENDFLAG
        !
                                   GET #1
                                   TOTAL.RATE = TOTAL.RATE + RATE
        !
            NEXT
510     !
511     !
590     !   --------------  REPORT CUMULATIVE SUM BELOW  --------------
591     !
599     !
600         PRINT "TOTAL.RATE: $";TOTAL.RATE
610     !
611     !
690     !   --------------  REPORT COMPLETED: CLOSE FILE(S)  --------------
699     !
700         CLOSE #1
900     !
999         END
```

Figure 7-15

Sample Structured Basic Program

```
100         A(I) = A(I) + 1                 FOR       I = 1 TO 100
110     !
200         PRINT SUMMARY.DATA              IF        OPTION.1 AND REPORT="MONTHLY"
210     !
300         PRINT FNHOUSE.PAYMENT           UNTIL     RATE < 123.45
310     !
400         GET #1                          WHILE     EMPLOYEE.NUMBER < 76000
410     !
500         GOSUB 12300                     UNLESS    ERROR.FLAG
510     !
600         PRINT "NORMAL EXIT"             IF        TOTAL > 1000   UNLESS ERRORS > 0
```

Figure 7-16

Statement Modifiers

that variable will not be in static storage (see MAP, COMMON above).

Finally, the EXTERNAL statement is provided to let the BASIC programmer explicitly declare data types for symbols external to the current program unit, e.g. the name of a VMS system service module, an external BASIC function, or an external constant which is to be global in an application.

Figure 7-17 illustrates the use of COMMON, MAP, DECLARE, and EXTERNAL statements.

Files and Records

VAX-11 BASIC supports RMS sequential, indexed, and relative file organization. In addition, BASIC applications can access virtual arrays (stored on files), terminal-format files, and block I/O files via RMS.

The OPEN statement in VAX-11 BASIC allows specification of file organization, access modes, file sharing, record formats, record size, and file allocation. At the record level, a BASIC program can FIND, GET, PUT, UPDATE, DELETE, or RESTORE any record in a file either sequentially or randomly.

VAX-11 BASIC can access files created by other native mode languages, assuming appropriate data representations are maintained with the records.

Symbolic Characters

BASIC now supports references to symbolic characters—those characters in the 96-character ASCII set which do not print, e.g. NUL, SOH, FF, CR, etc. Figure 7-18 illustrates the use of symbolic characters in a BASIC program.

```
100  !  ---------------------  COMMON STATEMENTS  --------------------
101  !
102     COMMON (DATASET1) REAL A,B,C,D,E,F,G,H,O,P,Q,R,S,T,U,V,W,X,Y,Z,                    &
                                  INTEGER      I,J,K,L,M,N                                  &
                                  STRING       S1,S2,S3,S4
103  !
104     COMMON (DATASET1) LAST.NAME$=10, FIRST.NAME$=5
105  !
200  !  ----------------------  MAP STATEMENTS  --------------------
201  !
202     MAP (DATASET2)              REAL         PART.NUMBER, COST,                           &
                                    INTEGER      VENDOR.CODE, QA.INDEX,                       &
                                    STRING       VENDOR.ID=40
203  !
204     MAP (DATASET2)              REAL         FILL,     FILL,                              &
                                    INTEGER      FILL,     FILL,                              &
                                    STRING       VENDOR.NAME = 10, FILL,                      &
                                                 VENDOR.TWX = 30
205  !
300  !  -----------------------  DECLARE STATEMENTS  -----------------------
301  !
302     DECLARE                     INTEGER      COUNTER.1, COUNTER.2,                        &
                                    REAL         STANDARD.DEVIATION,                          &
                                    LONG         A.32.BIT.VARIABLE,                           &
                                    WORD         A.16.BIT.VARIABLE,                           &
                                    STRING       LAB.NAME = 20                                &
303  !
304     DECLARE                     INTEGER      CONSTANT      DEBUG.MODE     = 0,            &
                                                              MY.P = 3,                        &
                                    REAL         CONSTANT      MY.PI          = 3.1416,       &
                                    STRING       FUNCTION      CONCAT
305  !
306     DEF CONCAT( STRING Y, STRING Z)
307             CONCAT = Y + Z
308     FNEND
309  !
310     PRINT CONCAT("THIS IS"," THE RESULT")
311  !
312  !---------------------------  EXTERNAL STATEMENTS  -------------------------
401  !
402  !                                                                  CAN BE USED FOR VMS SERVICES
404     EXTERNAL                    INTEGER FUNCTION SYS$ASSIGN
405  !
406     EXTERNAL                    INTEGER FUNCTION SYS$TRNLOG      ! LOGICAL TRANSLATIONS
407  !
408     EXTERNAL                    INTEGER FUNCTION SYS$QIOW        ! SYNCHRONOUS QIO CALL
410  !
500  !-----------------------------------------------------------
```

Figure 7-17

Declaration Statements

```
10          PRINT "PROGRAM STARTS...";LF;LF;"AT "+TIME$(0)
11  !
15          TITLE$ = "SUMMARY REPORT"
19  !
20          PRINT TITLE$;CR; FOR I = 1 TO 5        !     Bold copy
                                                        by overprinting
21  !
30          PRINT
31  !
40          PRINT A(I) FOR I = 1 TO 10             !     Output report data
41  !
50          PRINT
51  !
99          END
Ready
RUN
TEST5                       28-MAY-1980         17:20
PROGRAM STARTS...
                        AT 05:20 PM
SUMMARY REPORT
     0
     0
     0
     0
     0
     0
     0
     0
     0
     0
Ready
```

Figure 7-18

Symbolic Characters

CALL Facility

The CALL statement allows the BASIC programmer to invoke procedures and functions that are external to the current source module. By using the VMS native mode LINK utility, procedures written in any of the VAX native mode languages can be invoked, i.e., BASIC routines can call or be called by procedures written in COBOL, CORAL, FORTRAN, PASCAL, etc.

The CALL statement in VAX-11 BASIC has been extended to allow a procedure to pass parameters BY REFerence, BY VALUE, or BY DESCriptor. These mechanisms conform to the VAX-11 procedure calling standard and allow BASIC programs to call VMS service routines and accept returning status values.

Shareable Programs

Applications written in VAX-11 BASIC can be made shareable images by the VMS LINKER. BASIC now generates fully re-entrant PIC code.

Developing BASIC Programs

VAX-11 BASIC delivers a high-productivity development environment. The key features of this environment include:

- Automatic line number generation via SEQUENCE command.
- Integral line editing with EDIT.
- A RUN command which allows a program to be placed directly into execution without requiring a separate LINK operation.
- Direct execution of unnumbered BASIC statements, allowing quick verification of algorithms, inspection/change of data values, and invocation of subroutines or functions in a halted BASIC program.
- An integral HELP facility helps program debug/development by providing online reference text from the BASIC manual set.
- The VAX-11 BASIC system can produce source language listings with embedded diagnostics indicating the line and position of any errors. Fully descriptive diagnostic messages are provided at the point of an error. Many error conditions are caught at compile time. At the user's option, VAX-11 BASIC can also output a machine language listing and/or a cross-reference listing.
- The VAX/VMS SYMBOLIC DEBUGGER lets the programmer set breakpoints, and inspect or change the value of variables during execution of a program.

Figure 7-19 illustrates the use of several of these features. The text appearing in blue type in Figure 7-19 corresponds to user input, the remaining text is supplied by the BASIC system.

The LOAD Command

A major goal of VAX-11 BASIC is to support a program development *environment*. The LOAD command allows a user to stay in BASIC, even when a program under development involves several separately compiled BASIC subroutines. When a RUN command is issued, any BASIC modules moved into memory by the previous LOAD command are automatically bound together with the module under development and the resulting in-memory image begins execution, i.e., the user is not required to leave BASIC, invoke the LINKER, and use the DCL $RUN command. This speeds program development considerably.

Once an application has been checked out, a final call on the LINKER can be used to create a shareable native mode executable image for production use.

Error Handling

VAX-11 BASIC allows user-directed error and event handling. Occurence of an error can activate one or more routines which handle the error (or event), and then return control to the point where the error occurred (RESUME), or to the calling program (ON ERROR GOBACK), or to the BASIC system itself for standard cleanup and return of control at the BASIC command level.

In determining the cause of an error, the BASIC program can use the value of: ERR—the error message number assigned by BASIC, ERL—the line number where the error occured, ERN$—the name of the BASIC module where the error occurred, and ERT$(ERR)—the error message text which the BASIC system would print if the error were not trapped by the program.

Migration to VAX/VMS

During the VAX-11 BASIC Field Test, numerous sites moved programs from BASIC-PLUS-2 and BASIC-PLUS (on PDP-11 systems) to the VAX native BASIC. A typical site converted literally hundreds of programs and generally had few difficulties. Minor changes were made to BASIC-PLUS-2 programs: the error checking in VAX-11 BA-

```
100     !--------------------INPUT A FILE NAME, COUNT NUMBER OF LINES IN IT----------------
110     LINPUT "What file to be opened ",FILE.NAME$
140     F.NAME$ = EDIT$(FILE.NAME$,32%)
160     OPEN F.NAME$ FOR INPUT AS FILE #1
180     ON ERROR GOTO 900
200     LINPUT #1%,TEXT$        FOR I = 1 to 1000000
210     STOP
900     LINE. = ERL
        NUMBER. = ERR
        MESSAGE$ = ERT$(NUMBER.)
        RESUME LINE 910
910     PRINT "*END, FROM LINE";LINE.;"WITH TEXT: ";MESSAGE$;
        PRINT " - AFTER ";I;"RECORDS"
991     STOP
995     PRINT "*** THE END ***"
999     END
Ready
RUNNH
%BASIC-E-SYNERR, syntax error
        at line 900 statement 4
              RESUME LINE 910
                          ↑
Ready
HELP RESUME
RESUME

The RESUME statement marks the end of an error handling routine,
and returns program control to a specified line number.
Format
        RESUME [<lin-num>]
Examples
        990 RESUME 300
        or
        990 RESUME


Ready
LIST 900
TEST6 28-MAY-1980 17:15
900     LINE. = ERL
        NUMBER. = ERR
        MESSGE$ = ERT$(NUMBER.)
        RESUME LINE 910
Ready
EDIT 900 / LINE / /
900     LINE. = ERL
        NUMBER. = ERR
        MESSAGE$ = ERT$(NUMBER)
        RESUME 910
Ready
RUN
TEST6 28-MAY-1980 17:16
What file to be opened          ? TEST6.BAS
*END, FROM LINE 200 WITH TEXT: ?End of file on device - AFTER 17 RECORDS
%BAS-I-STO, Stop
-BAS-I-FROLINMOD, from line 991 in module TEST 6
Ready

PRINT MESSAGE$;" FROM FILE";F.NAME$
?End of file on device FROM FILETEST6.BAS
Ready
PRINT F.NAME$;CR; FOR I = 1 TO 5
TEST6.BAS
Ready
```

Figure 7-19

BASIC Program Development Features

SIC caught actual bugs in many "working" programs. BASIC-PLUS programs were converted to EXTEND mode (or run through the BASIC-PLUS to VAX-11 BASIC translator) and then modified as though they were in BASIC-PLUS-2. Typically, these changes were made:

- the MODE expression on an OPEN statement was changed to the corresponding set of keywords, e.g.,

  OPEN F$ AS FILE #1 MODE2%

  becomes

  OPEN F$ AS FILE #1, ACCESS APPEND
- MAP and DIM statements were moved to occur before OPEN statements
- RSTS/E SYS-CALLS were examined and removed if not supported by VMS

Files were then copied over on tape or by using DECnet, and the programs were RUN under VAX-11 BASIC. In the event errors were detected by BASIC, the online HELP facility was used to determine any additional changes needed for correct compilation.

Certain features were carried forward from PDP-11 BASIC-PLUS and PDP-11 BASIC-PLUS-2 to VAX-11 BASIC in order to make the move to VAX easier. These include:

- BASIC-PLUS to VAX-11 BASIC Translator utility
- Program RESEQUENCE utility from BASIC-PLUS-2 V1.6
- FIELD statement
- CVT, SWAP, and MAGTAPE functions
- Foreign buffer support
- String arithmetic
- Numerous non-privileged RSTS/E SYS calls
- Virtual arrays

Performance

The programs in Figure 7-20 illustrate the level of compute-bound performance possible under VAX-11 BASIC. Program A was taken from page 84 of the March, 1980 issue of BYTE magazine. Program B is very similar and is from page 130 of the June, 1980 issue of Interface Age.

Finally, initial performance tests on VAX-11 BASIC were samples of the "Towers of Hanoi" program and the Whetstone benchmark. These tests show VAX BASIC execution speeds comparable to non-optimized VAX-11 FORTRAN.

Additional Functions

The features listed below complete the promise of a BASIC that leads the competition in virtually every area. This is not an exhaustive list, but does serve to indicate key capabilities of this new product.

- powerful string manipulation functions for creating, converting, searching, editing, and extracting character values
- variable names up to 30 characters long
- maxiumum length of a single string is 65,535 characters
- multiple statements on a line
- multiline IF...THEN...ELSE statements
- optional use of line continuator "&" and statement separator "\", e.g.,

```
100 PRINT          vs.    100   PRINT        &
    PRINT                     \ PRINT        &
    PRINT                     \ PRINT
```

- DCL pass-through in the BASIC command mode by simply prefixing the DCL command line with a dollar-sign, e.g.,

```
Ready
$DIR      *.BAS, *.OBJ
```

- Provision for up to ten individual BASIC object library files for automatic use at RUN time when developing an application using separately-compiled BASIC subroutines.

```
PRIMES          29-MAY-1980 21:08
90      OPEN "T.1" FOR OUTPUT AS FILE #1, RECORDSIZE
132
100     rem     Interface Age's benchmark program to
110     rem     discover the first 1000 prime numbers
120     rem
125     PRINT CHR$(7)
130     t1 = time(1)
140     FOR n = 1 to 1000
150             FOR k = 2 TO 500
160                     m=n/k
170                     I=int(m)
180                     IF I = 0 THEN 230
190                     IF I = I THEN 220
200                     IF m > I THEN 220
210                     IF m = I THEN 240
220             next k
230     PRINT #1, N;
240     NEXT n
250     t2 = time(1)
255     PRINT CHR$(7)
260     PRINT "Elapsed time: ";0.1*(t2-t1);" seconds"
270     END
```

System	CPU	Run-time
TRS-80	Z80	1982 sec
Technico	TI9900	585 sec
DEC-10	PDP-10	65 sec
BASIC-PLUS-2	PDP-11/70	11 sec
VAX-11 BASIC	11/780	2.7 sec

Figure 7-20A
Program A

```
PRIME3      29-MAY-1980 21:09
10    !     PRIME NUMBER PROGRAM #3 OF 3          &
      !                                           &
      !     FROM MARCH 1980 BYTE MAGAZINE         &
      !     PAGE 84                               &
      !     "TRS-80 PERFORMANCE...                &
      !     EVALUATION BY PROGRAM TIMING          &
      !     (INCLUDES 370/148 PL/I AND BAL TIMES  &
      !
15    DECLARE INTEGER M,K
20    OPEN "PRIMES3.TMP" FOR OUTPUT AS FILE #1
30    PRINT "PRIME3 "+TIME$(0)
40    PRINT
45    T1=TIME(1)
50    PRINT #1, 1;2;3;
55    C=0
70    M=3
80    M=M+2
90    FOR K = 3 TO M/2 STEP K-1
100         IF INT(M/K)*K-M = 0 THEN 190
110   NEXT K
121   PRINT #1%, M;
122   C=C+1
190   IF M < 10000 then 80
195   PRINT #1%, "C=";C
196   PRINT "C=";C
199   T2 = TIME(1%)
      P$ = "DONE: "+NUM1$(0.1*(T2-T1))+" CPU SEC"
200   PRINT P$
      PRINT #1, P$
201   END
```

TRS-80 BASIC	Z80	23470 sec
Assembler	Z80	1370 sec
Optimizing PL/I	370/148	79 sec
BAL	370/148	56 sec
VAX-11 BASIC	11/780	58.2 sec

Figure 7-20B
Program B

VAX-11 PL/I

Introduction

VAX-11 PL/I is an extended implementation of the General Purpose Subset (X3.74—"Subset G") of ANSI PL/I, X3.53-1976. VAX-11 PL/I extensions to the subset language are either full language PL/I features included because they were highly desirable, or system-specific extensions intended to provide more complete access to VAX/VMS features. VAX-11 PL/I is a shareable compiler which runs under the VMS operating system and generates highly optimized positon-independent machine code.

All compiler-generated code, with the exception of some built-in functions calls and I/O operations, is inline. Out-of-line operations are performed by the VAX-11 Common Run Time Library. Most high-level language operations are supported directly by VAX hardware instructions.

VAX-11 PL/I supports the VAX Symbolic Debugger.

All VMS system services are available to programs written in PL/I via the CALL statement. Furthermore, VAX-11 PL/I fully supports RMS, the VAX/VMS record management services. A set of ENVIRONMENT options provides access to a large subset of RMS features. All RMS file organizations are supported: sequential, relative, and indexed.

VAX-11 PL/I fully supports the VAX interlanguage calling standard. Routines written in any other native mode language can call PL/I and vice versa. In addition, all VAX/VMS system services and system utilities (Run Time Library, SORT, etc.) are available via the PL/I CALL statement. For system services, a library of predefined ENTRY declarations is provided to minimize the coding required to use these services.

Subset G is a rich language that combines the scientific computing abilities of FORTRAN, the commercial data handling of COBOL, the string manipulation of BASIC, and the block structuring of ALGOL. Selected extensions further enhance these basic capabilities.

The remainder of this section:

- provides an overview of the G Subset
- lists the extension made to the language to provide enhancements for PL/I programs executing in the VAX/VMS operating system environment
- lists features of full PL/I that were excluded from the G Subset but that have been incorporated in the implementation of VAX-11 PL/I
- lists the implementation-defined values that are used in VAX-11 PL/I

THE G (GENERAL-PURPOSE) SUBSET

The G subset of PL/I was designed to be useful in scientific, commercial, and system programming, especially on small and medium-size computer systems. Among the primary goals of the design of the subset were:

- to include features that were easy to learn and to use and to exclude features that were difficult to learn or prone to error
- to provide a subset that would be easily portable from one computer system to another

- to exclude features that were not often used and whose implementation greatly increased the complexity of the run time support required by the compiler

The essential elements of the subset are described below.

Program Structure

The G subset includes a complete character set, with comments, identifiers, decimal arithmetic constants, and simple string constants.

Begin blocks and DO-groups are included in the subset. Each block or group in the program must be terminated with an END statement. For example:

```
ON ENDFILE(INFILE) BEGIN;
   PUT SKIP LIST('End of input file');
   CLOSE FILE(INFILE);
   END;
```

Program Control

The following program control statements are included in the subset: CALL, RETURN, IF, DO, GOTO, null, STOP, ON, REVERT, and SIGNAL.

The DO statement options supported are TO, BY, WHILE, and REPEAT.

An IF statement may contain unlabeled THEN and ELSE clauses. A null statement may be used to specify no action for a given condition. For example:

```
IF A<B THEN; /* no action */
   ELSE PUT LIST('valid');
```

An ON statement may specify a single condition. The condition names supported are ERROR, ENDFILE, ENDPAGE, FIXEDOVERFLOW, KEY, OVERFLOW, UNDEFINEDFILE, UNDERFLOW, and ZERODIVIDE. For example, an attempt to divide by zero can be detected and handled by an ON-unit that specifies ZERODIVIDE:

```
ON ZERODIVIDE BEGIN;
   /* action to be taken
      .
      .
   */
   END;
```

Storage Control

The subset includes the assignment statement and the assignment of array and structure variables whose dimensions and data types match. The subset also permits aggregate promotion, that is, the assignment of a scalar expression to every element or member of an aggregate variable. For example:

```
ARRAY = A1;
```

evaluates the expression A_1 and assigns the result to every element of ARRAY.

The subset also provides the INITIAL attribute, which specifies initial values for variables when they are declared. For example:

```
DECLARE EOF BIT(1) STATIC INITIAL('0'B);
```

declares an "end-of-line" flag named EOF and sets its initial value to '0'B (false). In the subset, only static variables may be initialized.

The ALLOCATE statement with the SET option and the FREE statement are included in the subset. ALLOCATE dynamically allocates storage for a based variable and sets a pointer to the location of the allocated storage. For example:

```
ALLOCATE LIST_ELEMENT SET(LIST_POINTER);
```

allocates storage for the based variable LIST_ELEMENT and sets LIST_POINTER to the location of the allocated storage. The allocated storage can be subsequently released by:

```
FREE LIST_POINTER→LIST_ELEMENT
```

Input/Output

The I/O statements are:

- OPEN and CLOSE.
- READ, WRITE, DELETE and REWRITE for record I/O. Record I/O statements operate on an entire record in a file.
- GET, and PUT, with FILE, STRING, EDIT, LIST, PAGE, SKIP, and LINE options for stream I/O. Stream I/O statements operate on a stream of ASCII input or output data; the stream may be a file of such data or a character-string variable or expression.

The file attributes, specified in DECLARE or OPEN, are DIRECT, ENVIRONMENT, INPUT, KEYED, OUTPUT, PRINT, RECORD, SEQUENTIAL, STREAM, and UPDATE.

The FORMAT statement is included. The format items are E, F, P, A, X, R, PAGE, SKIP, COLUMN, TAB, and LINE. Format items, and the GET EDIT and PUT EDIT statements, provide a formatted I/O capability comparable to that of FORTRAN. For example:

```
PUT EDIT(A) (F(5,2));
```

writes out the value of A as a fixed-point decimal number of up to five digits, two of which are fractional.

Attributes and Pictures

The DECLARE statement is included in the subset. All names must be declared, either by means of a DECLARE statement or by means of a label prefix.

The attributes supported are: ALIGNED, AUTOMATIC, BASED, BINARY, BIT, BUILTIN, CHARACTER, DECIMAL, DEFINED, DIRECT, ENTRY, ENVIRONMENT, EXTERNAL, FILE, FIXED, FLOAT, INITIAL, INPUT, INTERNAL, KEYED, LABEL, OPTIONS, OUTPUT, PICTURE, POINTER, PRINT, RECORD, RETURNS, SEQUENTIAL, STATIC, STREAM, UPDATE, VARIABLE, and VARYING.

The picture characters included are CR, DB, S, V, Z, 9, −, +, $, and *. The picture insertion characters (. , / B) are also included. Pictures allow special characters to be inserted in a fixed-point decimal number. The picture facility (on output) is comparable to the PRINT USING statement of BASIC. For example:

```
PUT EDIT(A) (P '$99999V.99');
```

Writes out the value of A in fixed-point format with five integral digits and two fractional digits, and precedes the number with a dollar sign.

Built-in Functions and Pseudovariables

PL/I provides a set of built-in functions that perform common calculations and data manipulation. A built-in function may be used without declaration, wherever an expression of the same data type is permitted. The built-in

functions in the subset are: ABS, ACOS, ADDR, ASIN, ATAN, ATAND, ATANH, BINARY, BIT, BOOL, CEIL, CHARACTER, COLLATE, COPY, COS, COSD, COSH, DATE, DECIMAL, DIMENSION, DIVIDE, EXP, FIXED, FLOAT, FLOOR, HBOUND, INDEX, LBOUND, LENGTH, LINENO, LOG, LOG2, LOG10, MAX, MIN, MOD, NULL, ONCODE, ONFILE, ONKEY, PAGENO, ROUND, SIGN, SIN, SIND, SINH, SQRT, STRING, SUBSTR, TAN, TAND, TANH, TIME, TRANSLATE, TRUNC, UNSPEC, VALID, and VERIFY.

Subset G also provides pseudovariables that can be used as the targets of assignment statements. The pseudovariables are PAGENO, STRING, SUBSTR, and UNSPEC. For example, whereas the SUBSTR built-in function returns a substring of a specified bit or character string, the SUBSTR pseudovariable allows the assignment of an expression to such a substring.

Expressions

The subset supports all infix and prefix operators, the locator qualifier, parenthesized expressions, subscripts, and function references. Implicit conversion from one data type to another is restricted to those contexts in which the conversion is likely to produce the desired results. For example:

```
DECLARE I DECIMAL;
I = '1.2';
```

converts the character string '1.2' to the decimal equivalent and assigns it to the decimal variable I. However, the following assignment:

```
DECLARE B BIT(8);
B = 'A';
```

is not valid because, to be convertible to a bit string, a character string must consist entirely of 0 and 1 characters.

VAX-11 EXTENSIONS TO THE G SUBSET STANDARD

Procedure-Calling and Condition-Handling Extensions

The following extensions to PL/I were made to allow VAX-11 PL/I procedures to call procedures written in any other programming language that also supports the VAX-11 calling standard.

1. The attributes ANY and VALUE describe how data are to be passed to a called procedure.
2. The VARIABLE option for the ENTRY attribute permits a PL/I procedure to call a non-PL/I procedure with an argument list of variable length. It also permits a procedure to omit arguments in an argument list.
3. The DESCRIPTOR built-in function may be used to pass an argument by descriptor to a non-PL/I procedure.

The following new attributes provide storage classes for PL/I variables. These attributes permit PL/I programs to take advantage of features of the VAX-11 linker and to combine PL/I procedures with other procedures that use these storage classes.

1. The GLOBALDEF and GLOBALREF attributes let you define and access external global variables and optionally to place all external global definitions in the same program section.
2. The READ ONLY attribute can be applied to a static computational variable whose value does not change.
3. The VALUE attribute defines a variable that is, in effect, a constant whose value is supplied by the linker.

The following extensions to ON condition handling provide support for condition handling in the VAX/VMS environment:

1. The ON statement supports the ANYCONDITION keyword. The ON-unit established by this keyword is executed when any condition occurs for which no explicit ON-unit exists.
2. The ON statement supports programmer-named conditions with the VAXCONDITION keyword.
3. The RESIGNAL built-in subroutine permits an ON-unit to keep a signal active.
4. The ONARGSLIST built-in function provides an ON-unit with access to the mechanism and signal arguments of an exception condition.

Support of VAX-11 Record Management Services

The options of the ENVIRONMENT attributes provide support for many of the features and control values of the VAX-11 Record Management Services (RMS). Additional extensions have been made to the PL/I language to augment this support, as described below.

1. The OPTIONS option is supported on the GET, PUT, READ, WRITE, REWRITE, and DELETE statements. For example:

   ```
   GET LIST(A) OPTIONS(PROMPT('Enter A>'));
   ```

   displays the prompt "Enter A" on the user's terminal.
2. These built-in subroutines provide file handling and control functions: DISPLAY, EXTEND, FLUSH, NXTVOL, REWIND, and SPACEBLOCK.

Miscellaneous Extensions and Deviations

The following list summarizes miscellaneous extensions and deviations.

1. The RANK and BYTE built-in functions are supported, which return the ASCII code for a given character and the ASCII character for a given code, respectively.
2. The expression in a WHILE clause or in an IF statement may be a bit string of any length. When evaluated, the expression results in a true value if any bit of the string expression is a one and in a false value if all bits in the string expression are zeroes.
3. The control variable and the expressions in the TO, BY, and REPEAT options of the DO statement are not restricted to integers and pointers.

FULL PL/I FEATURES SUPPORTED

The items listed below are features that are explicitly excluded from the subset standard but that have been implemented in VAX-11 PL/I. These features all exist in full PL/I.

1. The ENTRY statement is supported. The ENTRY statement provides a means of defining alternate entry points to a procedure. For example, such an alternate entry point can be invoked by a function reference even though the containing procedure is invoked as a subroutine.
2. The ENVIRONMENT option is supported on the CLOSE statement.
3. The picture characters Y, T, I, and R are supported,

and pictures may include iteration factors. These characters provide an additional means of representing zeros in a number (Y) and allow a digit and a sign to be represented by a single character (T, I, R).

4. RETURNS CHARACTER(*) is valid. That is, a function can return a character string whose length is determined by the program.
5. The FINISH condition is supported.
6. A REWRITE statement need not specify the FROM option if the most recent I/O operation on the file was a READ statement with the SET option. (A READ SET statement acquires a record from an input file and sets a pointer to the location of the I/O buffer containing the contents of the record.)
7. The AREA and OFFSET attributes are supported. The AREA attribute allows declaration and manipulation of an entire region of memory (up to 500 million bytes), and the OFFSET attribute declares variables that are offsets into such an area. Offset variables may be used in most contexts in which pointer variables are permitted. Allocation within an area must be controlled by a user-written procedure.
8. The OFFSET and POINTER built-in functions are supported. These functions convert pointer values to offset values, and vice versa.
9. The POSITION attribute is supported. POSITION allows the declaration of a DEFINED variable to specify the position within the base string at which the definition begins.
10. Automatic variables may be initialized. The INITIAL attribute may contain scalar expressions and asterisks with automatic variables. Asterisks indicate that the corresponding variable or element in the declaration is not assigned an initial value.
11. The SET option is optional on the ALLOCATE statement if the allocated variable was declared with BASED(pointer-reference). The ALLOCATE statement then allocates storage for the based variable and sets the referenced pointer variable to the storage location.
12. The character pair /* may be embedded in a comment. This feature permits a statement such as:

```
IF ↑VALID(P) THEN
   PUT LIST('Input error');/* check validity */
```

to be "commented out":

```
/* IF ↑VALID(P) THEN
   PUT LIST('Input error');/* check validity */
```

IMPLEMENTATION-DEFINED VALUES AND FEATURES

1. VAX-11 PL/I supports the full 256-character ASCII character set.
2. The default precisions for arithmetic data are:

   FIXED BINARY (31)
   FIXED DECIMAL (7)
   FLOAT BINARY (24)
   FLOAT DECIMAL (7)

   where each precision is the number of binary or decimal digits, as appropriate.
3. The maximum record size for SEQUENTIAL files is 32,767 bytes minus the length of any fixed-length control area.
4. The maximum key size is 255 bytes for character keys.
5. The default value for the LINESIZE option is as follows. The line size is used by stream I/O statements to determine when to go to a new line.
   - If the output is to a physical record-oriented device, such as a line printer or terminal, the default line size is the width of the device.
   - If the output is to a print file, the default line size is 132.
   - If the output is to a nonrecord device (magnetic tape), the default line size is 510.
6. The default value for the PAGESIZE option is as follows. The page size is used by stream output (PUT) statements to determine when to signal the ENDPAGE condition.
   - If the logical name SYS$LP_LINES is defined, the default page size is the numeric value of SYS$LP_LINES – 6.
   - If SYS$LP_LINES is not defined, or if its value is less than 30 or greater than 90, or if its value is not numeric, the default page is 60.
7. The values for TAB positions are columns beginning with column 1 and every eight columns thereafter: 1, 9, 17, 25, .. 8*i+1, where i is (line size)/8. List-directed output (by the PUT LIST statement) is positioned on tab stops if the output is to a file with the PRINT attribute.
8. The maximum length allowed for a file title is 128 characters. File titles are VAX/VMS file specifications that are associated with PL/I file constants and file variables.
9. The maximum number of digits in editing fixed-point data is 34.
10. The maximum numbers of digits for each combination of base and scale are:

    FIXED BINARY —31
    FIXED DECIMAL —31
    FLOAT BINARY —113
    FLOAT DECIMAL —34
11. The default precision for integer values is 31.
12. The maximum number of arguments that can be passed to an entry point is 253.

VAX-11 PL/I Programming Example

Figure 7-21 illustrates a VAX-11 PL/I program. This program calls a VAX/VMS system service (SYS$TRNLOG) to determine the equivalence string for a logical name.

The program TRNLN calls a VAX/VMS system service, SYS$TRNLOG, to determine the equivalence string for a logical name. SYS$TRNLOG is declared as an external entry constant, with three parameters:

1. A character string of any length, representing the logical name.
2. An integer representing the length of the translated name.
3. A character string of any length representing the translated name itself.

```
/* Translates logical names to equivalent strings */
TRNLN: PROCEDURE OPTIONS (MAIN);
DECLARE SYS$TRNLOG ENTRY(                    /* VAX/VMS system service */
                                             / Parameters: */
  CHARACTER(*),                              /* Logical name */
  FIXED BINARY(15),                          /* Length of translated name */
  CHARACTER(*)                               /* Translated name */
          )
  OPTIONS(VARIABLE)                          /* Some arguments optional */
  RETURNS(FIXED BINARY(31));                 /* SYS$TRNLOG returns integer */
DECLARE INPUT CHARACTER(63) VARYING,                    /* Input name */
  OUTPUT CHARACTERS(63),                     /* Translated name */
  OUTPUT_LEN FIXED BINARY(15),               /* Length of translated name */
  RETURN_STAT FIXED BINARY(31),              /* Return status of SYS$TRNLOG */
  SUCCESS BIT(1)                             /* Successful return */
                    ALIGNED BASED (ADD(RETURN_STAT));

DECLARE SS$_NOTRAN                                      /* Flag for undefined name */
  GLOBALREF FIXED BINARY(31) VALUE;
%REPLACE NOTEND BY '1'B;                    /* "Always true"--to control DO-group */
ON ENDFILE(SYSIN) STOP;                                 /* Stop on CTRL/Z from terminal */
DO WHILE (NOTEND);                          /* Terminated by ON-unit */
  PUT SKIP;
  GET LIST(INPUT)
                    OPTIONS(PROMPT('Enter logical name '));

    /*Invoke sytem service as PL/I function reference: */
    RETURN_STAT = SYS$TRNLOG((INPUT),OUTPUT_LEN,OUTPUT,,,);
    IF RETURN_STAT = SS$_NOTRAN THEN
      PUT SKIP LIST(INPUT::'not defined');
    ELSE IF SUCCESS THEN
      PUT SKIP LIST(INPUT::'is '::SUBSTR(OUTPUT,1,OUTPUT_LEN));
    END;
  END TRNLN;
```

Figure 7-21
Sample VAX-11 PL/I Code

The declaration also states that SYS$TRNLOG returns an integer and can have an argument list of variable length.

Note that the explicit declaration of SYS$TRNLOG is shown here for clarity. VAX-11 PL/I is supplied with a library of predefined entries for VAX/VMS system services, so the declaration of SYS$TRNLOG can be replaced by the single statement:

```
%INCLUDE SYS$TRNLOG;
```

The program also uses a global reference to SS$_NOTRAN; the return value of the system service equals SS$_NOTRAN when there is no defined equivalence for the given logical name.

SYS$TRNLOG actually requires that its first and third arguments be the addresses of character string descriptors. VAX-11 PL/I allows you to pass a character string by descriptor by declaring the corresponding parameter as CHARACTER(*). Such a parameter can have an argument that is a fixed-length character string of any length. If the written argument for such a parameter is a varying-length string, a dummy argument is created by the compiler. To avoid the accompanying warning message, the argument INPUT is enclosed in parentheses.

When a dummy argument is created, the invoked procedure cannot modify the associated parameter. Therefore, the translated name (OUTPUT) is not declared as a varying-length string. Instead, OUTPUT and OUTPUT_LEN both are supplied as arguments to SYS$TRNLOG, which then assigns the necessary values to them. The translated name is written out with a reference to the SUBSTR built-in function:

```
SUBSTR(OUTPUT,1,OUTPUT_LEN);
```

which writes out a substring beginning at character 1 of OUTPUT and continuing for OUTPUT_LEN characters.

A sample session with the program might be:

```
$ DEF LNK$LIBRARY DB1:[PROJECT]MYLIB,OLB<RET>
$ R TRNLN<RET>
```

```
Enter logical name>LNK$LIBRARY<RET>
LNK$LIBRARY is DB1:[PROJECT]MYLIB.OLB<RET>
Enter logical name>GRACE<RET>
GRACE not defined
Enter logical name>↑Z
$
```

VAX-11 PASCAL

Introduction

VAX-11 Pascal, a re-entrant native mode compiler, is an extended implementation of the Pascal language as defined by Jensen and Wirth in the *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 VAX-11 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 other Pascal implementations:

- Lexical
  - Upper- and lowercase 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 _ characters in identifiers
- Predefined data types
  - DOUBLE
  - SINGLE
- Predefined procedures
  - CLOSE (f)
  - FIND (f,n)
  - OPEN (f,...)
  - DATE (a)
  - HALT
  - LINELIMIT (f,n)
  - TIME (a)
- Predefined functions
  - LOWER (a,n)
  - SNGL (d)
  - UPPER (a,n)
  - EXPO (r)
  - CARD (s)
  - CLOCK
  - UNDEFINED (r)
- 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
- %INCLUDE directive
- VALUE initialization
- OTHERWISE clause in CASE statement
- External procedure and function declarations
- Dynamic array parameters
- Extended parameter specifications
  - %DESCR
  - %IMMED
  - %IMMED PROCEDURE and %IMMED FUNCTION
  - %STDESCR
- 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 VAX-11 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 stan-

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

Sample VAX-11 Pascal Code

Figure 7-22 illustrates a VAX-11 Pascal program which a user may write to calculate the hypotenuse of a right triangle. The inputs to the program are the length of the two sides of the triangle and the output is the length of the hypotenuse. This version of the program has not yet been debugged to illustrate how error messages are reported.

Compiler Listing Format

Figure 7-22 contains the compiler listing of the hypotenuse program. The compiler listing contains the following three sections:

- Source code listing—When the LIST qualifier is specified, the source code is listed by default.
- Machine code listing—To generate the machine code listing, the MACHINE_CODE qualifier must be specified.
- Cross-reference listing—To generate cross references for all identifiers used in the program, the CROSS_REFERENCE qualifier must be specified.

The following sections describe the format of the user requested listings. Note that the numbers in these sections are keyed to the circled numbers in the listings of Figure 7-22.

Title Line — Each page of the listing contains a title line. The title line lists the module name (1), the date and time of the compilation (2), the Pascal compiler name and version number (3), and the listing page number (4).

Source Code Listing

Each page of the source code listing contains a line under the title line specifying the date and time of source file creation (5) and the VAX/VMS file specification of the source file (6).

```
(1)                          (2)                          (3)                          (4)
EXAMPLE        23-MAY-1980   23:00:05      VAX-11 PASCAL VERSION V1.1-29        PAGE 1
01             11-OCT-1979   11:34:47      _DB1:[200,200]EXAMPLE.PAS;4 (1)
                                   (5)                    (6)
     LINE          LEVEL
   NUMBERS      PROC  STMT     STATEMENT  .  .  .
```

```
EXAMPLE 23-MAY-1980          23:00:05          VAX-11 PASCAL VERSION V1.1-29          PAGE 2
MACHINE-CODE
GENERATED  CODE   (PRIOR TO BRANCH OPTIMIZATION)
LINE        ADDRESS      OPCODE      OPERANDS          BYTESTREAM (HEXADECIMAL; READ FROM RIGHT TO LEFT)

                0002     MOVAB       *VAR(0, 0),R11          58 00 00 00 00 00 9E
                0009     MOVL        R13,*VAR(0, 4)          00 00 00 00 00 5D D0
                0010     CLRL        -(R14)                  7E D4
                0012     CLRD        -(R14)                  7E 7C
                0014     PUSHAL      (R11)B↑8                08 AB DF
                0017     CALLS       #1,PAS$INPUT            00 00 00 00 00 01 FB
                001E     PUSHAL      (R11)B↑8                98 AB DF
                0021     PUSHAL      (R11)W↑236              00 EC CB DF
                0025     CALLS       #2,PAS$OUTPUT (22)      00 00 00 00 00 02 FB
   (23) 8       002C     MOVL        R14,(R13)B↑-12          F4 AD 5E D0
        9       0030     MOVAB       (R11)W↑236,R10          5A 00 EC CB 9E
                0035     SUBL2       #16,R14                 (25) 5E 10 C2
                0038 (24)  PUSHL     R10                     5A DD
                003A     MOVAB       *VAR(2, 0),R9           59 00 00 00 00 00 9E
                0041     MOVL        R9,(R14)B↑4             04 AE 59 D0
                0045     MOVL        #21,(R14)B↑12           0C AE 15 D0
                0049     MOVL        #21,(R14)B↑8            08 AE 15 D0
                004D     CALLS       #5,PAS$WRITESTR         00 00 00 00 00 05 FB
                0054     MOVAB       (R11)W↑236,R10          5A 00 EC CB 9E
```

```
EXAMPLE 23-MAY-1980          23:00:05          VAX-11 PASCAL VERSION V1.1-29          PAGE 4
CROSS_REFERENCE
CROSSREFERENCE LISTING
A                                  4          11          12
B (28)                             4          11          12 (26)
C                                (29) 4       12          13
INPUT                              1          10
OUTPUT                             1
GLOBALLY DEFINED IDENTIFIERS:
EOLN                               0          10
FALSE                              0          14
READLN                             0          11
REAL                               0          4                  (27)
SQR                                0          12          12
WRITELN                            0          9           13          17
```

Figure 7-22 (Con't)
Sample VAX-11 Pascal Code

Source Code Listing — The lines of the source code are printed in the source code listing. In addition, the listing contains the following information pertaining to the source code:

- SOS line numbers (7)—If the source lines were created or edited in a Pascal module with the SOS editor, SOS line numbers appear in the leftmost column of the source code listing. SOS line numbers are irrelevant to the Pascal compiler.
- Line numbers (8)—The compiler assigns unique line numbers to the source lines in a Pascal module. The symbolic traceback that is printed if the program encounters an exception at run time refers to these line numbers.
- Procedure level (9)—Each line that contains a declaration lists the procedure level of that declaration. Procedure level 1 indicates declarations in the outermost block. The procedure level number increases by one for each nesting level of functions or procedures.
- Statement level (10)—The listing specifies a statement level for each line of source code after the first BEGIN delimiter. The statement level starts at 0 and increases by 1 for each nesting level of Pascal structured statements. The statement level of a comment is the same level as that of the statement that follows it.

Errors and Warnings — The source code listing includes information on any errors or warnings detected by the compiler. A line beneath the source code line in which the error is detected specifies whether the diagnostic is a warning or an error. In addition, the error description can contain the following information:

- A circumflex (↑) that points to the character position in the line where the error was detected (11).
- A numeric code, following the circumflex, that specifies

the particular error (12). On the following lines of the source listing, the compiler prints the text that corresponds to each numeric code (13). Note that one source program error often causes the Pascal compiler to detect more than one error (14).

- An asterisk (*) that shows where the compiler resumed translation after the error (15).
- The line number in which the error was detected (16) and the line number of the last line containing an error diagnostic (17). These error line numbers can be used to trace the error diagnostics backwards through the source listing.

Summary — At the end of the source listing, the compiler lists the amount of time required for the compilation (18). If program generated warning or error messages resulted, the compiler prints a summary of all the errors (20) and the source line number of the last message (19). Finally, the compiler lists the status of all the compilation options (21).

Machine Code Listing

The machine code listing (if requested with the MACHINE_CODE qualifier) follows the source listing. The machine code listing contains:

- Symbolic representation (22)—The symbolic representation, similar to a VAX-11 MACRO instruction, appears for each object instruction generated. Because the Pascal compiler operates in one pass, it must generate these instructions before it performs branch optimization. Branch optimization can cause certain BRW instructions to be deleted. Therefore, these instructions will not be identical to those appearing in the executable image.
- Source line number (23)—A source line number marks the first object instruction that the compiler generated for the first Pascal statement on that source line.
- Hexadecimal address (24)—The hexadecimal address is an approximation of the address of the object instruction. Do not use these addresses for debugging purposes because they do not correctly correspond to the locations in the executable image. The branch optimization mentioned above can change the addresses of the object instructions.
- Hexadecimal instruction (25)—This is the hexadecimal representation of the object instruction. The hexadecimal instruction should be read from right to left because the rightmost byte has the smallest address. Again, because of its one-pass operation, the compiler must generate some object instructions before it can determine the address bytes of their operands. The addresses of these operands are printed as zeros. After generating the hexadecimal representation of an instruction, but before writing the object code file, the compiler places the correct values into the binary object code.

Cross-Reference Listing

The cross-reference listing (if requested with the CROSS_REFERENCE qualifier) appears after the machine code listing. It contains two sections:

- User-specified identifiers (26)—This section lists all the user declared identifiers.
- Globally-defined identifiers (27)—This section lists the Pascal predefined identifiers that the program uses.

Each line of the cross-reference listing contains an identifier (28) and a list of the source line numbers where the identifier is used (29). The first line number indicates where the identifier is declared. Predefined identifiers are listed as if they were declared on line 0. The cross-reference listing does not specify pointer type identifiers that are used before they are declared.

VAX-11 BLISS-32

Introduction

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, as is most of the VAX-11 RTL.) 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.
- Operators—BLISS-32 supplies operators which interpret their operands as addresses, signed integers, unsigned integers or character-sequences.
- 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.)
- Condition Handling—the BLISS-32 language fully supports a VAX/VMS condition-handling environment. The ENABLE declaration establishes a condition-handler for exceptions raised within the scope of a routine. The SIGNAL, SIGNAL_STOP and UNWIND predeclared functions allow a programmer to raise conditions and process them.
- GLOBAL and EXTERNAL declarations—allowing code and data 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.
- Conditional Compilation—The programmer may include or exclude source text from compilation through use of %IF-%THEN-%ELSE-%FI. In conjunction with the lexical functions, this provides a powerful capability for tailoring source code for distinct environments.

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, you 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 (or STARLET.L32) 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. LIB.REQ (or LIB.L32) provides the definitions in STARLET, as well as definitions needed for writing ACPs or privileged kernel-mode code.
- 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 7-4).

Table 7-4

VAX-11 Machine Specific Functions

Processor Register Operations	
MTPR	Move to a Processor Register
MFPR	Move from a Processor Register
Parameter Validation Operations	
PROBER	Probe Read accessibility
PROBEW	Probe Write accessibility
Program Status Operations	
MOVPSL	Move from PSL
BISPSW	Bit set PSW
BICPSW	Bit clear PSW
Queue Operations	
INSQUE	Insert entry in Queue
REMQUE	Remove entry from Queue
Bit Operations	
TESTBITSS	Test for Bit Set, then Set bit
TESTBITSC	Test for Bit Set, then Clear bit
TESTBITCS	Test for Bit Clear, then Set bit
TESTBITCC	Test for Bit Clear, then Clear bit
TESTBITSSI	Test for Bit Set, then Set bit Interlocked
TESTBITCCI	Test for Bit Clear, then Clear bit Interlocked
FFS	Find First Set bit
FFC	Find First Clear bit
Extended Arithmetic Operations	
ASHQ	Arithmetic Shift Quad
EDIV	Extended Divide
EMUL	Extended Multiply
INDEX	Index (Subscript) Calculation
CRC	Cyclic Redundancy Calculation
Floating Point Conversion Operations	
CVTLF	Convert Long to Floating
CVTLD	Convert Long to Double
CVTFL	Convert Floating to Long
CVTDL	Convert Double to Long
CVTFD	Convert Floating to Double

Table 7-4 (con't)
VAX-11 Machine Specific Functions

CVTDF	Convert Double to Floating
CVTRDL	Convert Rounded Double to Long
CVTRFL	Convert Rounded Floating to Long
CMPF	Compare Floating
CMPD	Compare Double
String Operations	
MOVTUC	Move Translated Until Character
SCANC	Scan Characters
SPANC	Span Characters
Decimal String Operations	
MOVP	Move Packed
CMPP	Compare Packed
CVTLP	Convert Long to Packed
CVTPL	Convert Packed to Long
CVTPT	Convert Packed to Trailing Numeric
CVTTP	Convert Trailing Numeric to Packed
CVTPS	Convert Packed to Leading Separate Numeric
CVTSP	Convert Leading Separate Numeric to Packed
EDITPC	Edit Packed to Character
Miscellaneous Operations	
HALT	Halt Processor
ROT	Rotate
ADAWI	Add Aligned Word Interlocked
BPT	Breakpoint
CHMx	Change Mode
CALLG	Call with General Argument List
NOP	No Operation

The VAX-11 BLISS-32 Compiler

The VAX-11 BLISS-32 compiler performs a number of optimizations. These include common subexpression elimination, removal of loop invariants, constant folding, block register allocation, peephole replacement, test instruction elimination, jump vs. branch instruction resolution, branch chaining, and cross-jumping.

The VAX-11 BLISS-32 compiler optionally produces source text and generated code in a format closely resembling a VAX-11 assembly listing. Other options allow the programmer to control the degree of optimization, suppress production of object code, determine types and formats of output listings, generate traceback information, and specify the types of information to be listed at the terminal.

BLISS-32 generates shareable, re-entrant and position-independent code. These features make it easy to use BLISS-32 for writing software to be included in shareable libraries for use by any native language. it is also useful for writing connect-to-interrupt device handlers and user-written system services.

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 "precompiled," 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). Several language features enhance transportability:

- The high-level language constructs may be transferred from one machine to another with little or no alteration.
- Machine-specific functions can be separated from the common, mainline code via modularization, macros, and Library and Require files.
- Parameterization allows machine-specific characteristics to be passed to BLISS data structures.
- The compiler can be instructed to perform transportability checking via the "LANGUAGE (COMMON)" module-head switch.

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.

BLISS-32 Sample Program

Figure 7-23 illustrates 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.

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 66 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 66, which was first widely used in military projects beginning in 1970. Her Majesty's Stationery Office publishes the "Official Definition of CORAL 66."

The CORAL 66 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 be-

```
;    0001  MODULE showtime( IDENT='1-1' %TITLE 'SHOW TIME', MAIN=timeout)=
;    0002  BEGIN
;    0003
;    0004  LIBRARY 'SYS$LIBRARY:STARLET';                    ! Defines System Services, etc.
;    0005
;    0006  MACRO
;  M 0007    desc[] = %CHARCOUNT(%REMAINING),                ! A VAX-11 Style String descriptor
;    0008          UPLIT BYTE( %REMAINING) %;
;    0009
;    0010  OWN
;    0011          timebuf:        VECTOR[2],                                  ! 64 bit system time
;    0012          msgbuf:         VECTOR[80,BYTE],                            ! Output msg. buffer
;    0013          msgdesc:        BLOCK[8,BYTE]                               ! String descriptor
;    0014                            PRESET(     [dsc$w_length]= 80,           ! for output buffer
;    0015                                        [dsc$a_pointer] = msgbuf );
;    0016
;    0017  BIND
;    0018          fmtdesc= UPLIT(DESC('At the tone, the time is ', %CHAR(7), '115%T' ));
;    0019
;    0020  EXTERNAL ROUTINE
;    0021          lib$put_output : ADDRESSING_MODE(GENERAL);                  ! From VMS RTL
;    0022
;    0023  ROUTINE timeout=
;    0024          BEGIN
;    0025          LOCAL
;    0026                  RSLT:           WORD;                   ! Resultant string length
;    0027
;    0028          $GETTIM( TIMADR=timebuf );                      ! Get time as 64 bit integer
;    0029
;  P 0030          $FAOL(                  CTRSTR=fmtdesc,         ! Format control-string address
;  P 0031                                  OUTLEN=nslt,            ! Resultant length (only a word!)
;  P 0032                                  OUTBUF=msgdesc,         ! Output buffer descriptor address
;    0033                                  PRMLST=%REF(timebuf));  ! Address of pointer to time block
;    0034
;    0035          MSGDESC[dsc$w_length] = .rslt;                  ! modify output descriptor
;    0036
;    0037          lib$put_output( msgdesc )                       ! print the formatted time
;    0038
;    0039          END;
```

Figure 7-23
Sample VAX-11 BLISS-32 Code

ginning. 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 66 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 languages, and is designed for use on small or medium-size dedicated computers. One of the main intentions is that programs written in CORAL 66 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.

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 optional assembly listing. VAX-11 MACRO is similar to PDP-11 MACRO, but its instruction mnemonics correspond to the VAX native instructions. VAX-11 MACRO is characterized by:

- relocatable object modules
- global symbols for linking separately assembled object programs
- global arithmetic, global assignment operator, global label operator and default global declarations
- user-defined macros with keyword arguments
- 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 instruction mnemonics and MACRO directives and 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 to a source program module or global (externally available). An internal symbol definition is limited to the module in which it appears. Internal symbols are local definitions which are resolved by the assembler.

A global symbol can be defined in one source program module and referenced within another. Global symbols are preserved in the object module and are not resolved until the object modules are linked into an executable program. With some exceptions, all user-defined symbols are internal unless explicitly defined as being global.

Directives

A program statement can contain one of three different operators: a macro call, a VAX instruction mnemonic, or an assembler directive. MACRO includes directives for:

- listing control
- function specification
- data storage
- radix and numeric usage declarations
- location counter control

- program termination
- program boundaries information
- program sectioning
- global symbol definition
- conditional assembly
- macro definition
- macro attributes
- macro message control
- repeat block definition
- macro libraries

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 qualifiers in the command string issued to the MACRO assembler. The use of these qualifiers provides initial listing control options that may be overridden by the 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 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 multiuser 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, reentrant programs.

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-2is also available as an optional language processor for the RSTS/E, RSX-11M, RSX-11M PLUS, 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 a PDP-11 applications implementation language which features many programming facilities found in VAX-11 BASIC. These include:

- program formatting and commenting facilities
- long variable names
- indexed, sequential, and relative file I/O
- virtual arrays
- variable-length strings
- PRINT USING statement
- COMMON statement
- a subprogram CALL statement
- extended debugging facilities
- integrated INQUIRE (HELP) facility

The compiler accepts source programs 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 (OTS) 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 an object module library, the task builder 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 numbering determines the order in which the program is processed; the programmer can enter 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.

Powerful File I/O

The BASIC-PLUS-2 language supports use of RMS-11 block, indexed, sequential, and relative file operations. Although RMS-11 operates in RSX-11 compatibility mode, it does not have support for file sharing under VMS. For applications requiring access to shared files, VAX-11 BASIC should be used.

Powerful 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 concatenate 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 fixed 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 values or string elements of arrays manipulated in memory, other than the amount of available memory.

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 BASIC-PLUS-2 or MACRO-11 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 PDP-11 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.

PDP-11 FORTRAN IV/VAX to RSX

FORTRAN IV is an extended FORTRAN implementation based on American National Standard (ANSI) FORTRAN, X3.9-1966. PDP-11 FORTRAN IV code is executed in compatibility mode under VAX/VMS. The FORTRAN IV language includes the following extensions to the ANSI standard;

- general expressions allowed in all meaningful contexts
- mixed-mode arithmetic
- BYTE data type for character manipulation
- ENCODE, DECODE statements
- PRINT, TYPE, and ACCEPT input/output statements
- direct-access unformatted input/output DEFINE FILE statement
- comments allowed at the end of each source line
- PROGRAM statement
- OPEN and CLOSE file access control statements
- list-directed input/output

Additionally, virtual arrays are supported on target systems with memory management directives. Virtual arrays are memory-resident and require enough main memory to contain all elements of all arrays.

The PDP-11 FORTRAN IV compiler is a fast, one-pass compiler. Compiler options allow program size versus execution speed (threaded code versus inline code) tradeoffs. FORTRAN IV compiler optimizations include:

- common subexpression elimination
- local code tailoring
- array vectoring
- optional inline code generation for integer and logical operations

MACRO-11 assembly language subroutines may be called from FORTRAN IV programs. FORTRAN IV also includes a set of object modules, called the Object Time System (OTS), that are selectively linked with compiler-produced object modules to produce an executable program.

MACRO-11

MACRO-11, the PDP-11 assembly language, is included in the compatibility mode environment. Programs written in MACRO-11 can be assembled to produce relocatable object modules and optional assembly listings. MACRO-11 provides the following features:

- relocateable object modules
- global symbols for linking separately assembled object programs
- device and file name specifications for input and output files
- user-defined macros
- comprehensive system macro library
- program sectioning directives
- conditional assembly directives
- assembly and listing control functions at program and command string levels
- alphabetized, formatted symbol table listing
- default error listing on command output device

Symbols and Symbol Definitions

Three types of symbols can be defined for use within MACRO-11 source programs: permanent symbols, user-defined symbols, and macro symbols. Accordingly, MACRO maintains three types of symbol tables: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST).

Permanent symbols consist of the PDP-11 instruction mnemonics and MACRO directives. The PST contains all the permanent symbols automatically recognized by MACRO and is part of the assembler itself. Since these symbols are permanent, they do not have to be defined by the user in the source program.

User-defined symbols are those used as labels or defined by direct assignment. Macro symbols are those symbols used as macro names. The UST and MST are constructed during assembly by adding the symbols to the UST or MST as they are encountered.

Directives

A program statement can contain one of three different operators: a macro call, a PDP-11 instruction mnemonic, or an assembler directive. MACRO includes directives for:

- listing control
- function specification
- data storage
- radix and numeric usage declarations
- location counter control
- program termination
- program boundaries information
- program sectioning
- global symbol definition
- conditional assembly
- macro definition
- macro attributes
- macro message control
- repeat block definition
- macro libraries

8

Program Development and Support Facilities

VAX/VMS offers the user a powerful and sophisticated program development environment including several support facilities. Described in this section are the interactive and batch text editors, the linker, the Common Run Time Procedure Library, the VAX-11 interactive symbolic debugger, the librarian utility, command language procedures, the differences utility, and VAX-11 RUNOFF.

In addition to the interactive text editor SOS and the SLP batch editor, VAX/VMS now supports the powerful interactive text editor EDT. EDT supports many user oriented features including both line and character editing facilities, an extensive HELP facility, a journaling facility, and the window editing facility.

The VAX/VMS linker is the program development tool that takes the output of a translator or compiler and produces a file that can be executed on the VAX-11 hardware.

The Common Run Time Procedure Library offers the user a set of common language routines that can be called from any of the native mode languages via the VAX-11 calling standard.

The VAX-11 interactive symbolic debugger is extremely useful in isolating program errors by allowing the user to examine and modify the contents of memory locations while the program is executing.

The librarian is a utility that allows the user easy access to the data stored in any of the four libraries (object, macro, help, text). The librarian allows an executing program to initialize and open a library, and to retreive, insert, and delete modules.

The command language procedure section describes the power and flexibility of executing strings of frequently used sequences of DCL commands, or creating new commands from the existing DCL command set. Command procedures can accept up to eight parameters and can include extensive control flow.

By utilyzing the DIFFERENCES utility, the user can determine whether or not two files are identical.

VAX-11 RUNOFF is a document formatter, offering the user such features as page and title formatting, subject-matter formatting, and the ability to produce indexes and tables of contents easily.

INTRODUCTION

VAX/VMS provides a complete program development environment for the user. Development tools supporting this environment are:

- Interactive and batch text editors
- Linker
- Common Run Time Procedure Library
- VAX-11 interactive symbolic debugger
- Librarian Utility
- Command Language Procedures
- Differences Utility
- VAX-11 Runoff

These tools, as well as program language compilers, are available to the programmer via the command language facility. In addition, VAX/VMS supports a complete set of shareable routines collectively known as the common run time procedure library. And finally, VAX/VMS supports the user's ability to write programs utilizing the DCL command set (similar to coding programs in other high level languages). These programs are known as command language procedures.

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.

TEXT EDITORS

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

The user invokes the SOS and EDT text editors interactively, i.e., the user creates and processes files on-line. The SLP text editor, on the other hand, allows direct modification to a file via a command file prepared by the user. SOS is often used to create SLP command files. All editors are invoked by the command EDIT. The default editor is SOS. Therefore, to invoke SOS, enter the command EDIT or EDIT/SOS; to invoke EDT, enter EDIT/EDT; for SLP, use EDIT/SLP.

Before describing the text editors, a short summary of file naming conventions and default file types is presented.

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, several file types have standard meanings. Among these special file types are:

File Type	Default Use
.FOR	FORTRAN language source statements
.MAR	MACRO assembly source statements
.COB	COBOL language source statements
.BAS	BASIC language source statements
.PAS	PASCAL language source statements
.DAT	A data file
.LIS	An output listing from a compiler
.EXE	An executable image
.OBJ	An output file from a compiler

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

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

SOS is line-oriented, so it usually 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. 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.

Advanced features of SOS allow considerable flexibility in searching for a string of text and allow specification of blocks of text by content, or relative position from a known location, instead of by line number. SOS has many operational features under user control.

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>
```

If the user were to omit file-spec, SOS would immediately prompt the user for the missing parameter.

To terminate SOS, enter the command E (EXIT) followed by a carriage return 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 unless otherwise specified by the user.

SOS Examples

Copy command

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

Find command

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

Print command

1) P500:800
   Print lines 500 through 800.
2) P1800
   Print line numbered 1800.

Substitute command

1) Smore<ESC>less<ESC>,500:800
   Change all occurrences of "more" into "less" on lines numbered 500 through 800.

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, 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 system crashes, or loss of carrier on a dial-up line
- an extensive HELP facility
- a 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

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

EDT 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 user may position the window anywhere in the file. Window editing is illustrated in Figure 8-1.

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.

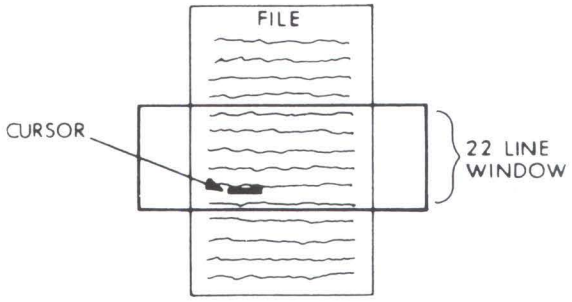


Figure 8-1
Window Editing

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 a picture of the keypad and provides additional information on each of the keypad keys.

The Keypad

The keypad is a special set of keys to the right of the main keyboard. Figure 8-2 illustrates the functions of the VT100 keypad; the VT52 keypad is similar.

SHIFT	HELP	FIND FNDNXT	UND L DEL EL
COMMAND PAGE	SECT	REPL APPEND	UND W DEL EW
BOTTOM ADVANCE	TOP BACKUP	PASTE CUT	UND C DEL C
WORD	DEL EOL EL	INSCOD CHAR	SUBS
OPEN LINE LINE		RESET SELECT	ENTER

Figure 8-2
VT100 Keypad Functions

Keypad functions allow the user to perform a variety of operations. Furthermore, the function of any keypad key can be changed to meet the needs of the user via the DEFINE command.

The commands in the keypad submode let users alter text or change the cursor position in the file. Keypad functions are available to 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.

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.

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, controls EDT's editing capabilities. SET controls such screen parameters as line width or 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 unlikely system crashes. During an editing session, EDT saves all the terminal input 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 EDIT command qualifier. In this way the user can recover edited files to the time of the crash.

The EDT CAI Program

Also available with VAX/VMS V2.0 EDT is a Computer-Assisted Instruction (CAI) program on EDT. 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.

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 redefine these entities to tailor them to specific applications, which can be as diverse as documentation or 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.

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. The SLP command file provides a reliable method of duplicating the changes made to a file, at a later time or on another 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 (line 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 an optional listing file and an updated copy of the corrected input file. SLP provides an optional 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. An interactive text editor is normally used to create SLP indirect command files. If both new and old versions of the file exist, the differences utility 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.

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 in the output file.

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. An audit trail, unless suppressed, is applied to lines changed by the update. The numbers generated by SLP for the listing file do not appear in the output file.

LINKER

The VAX/VMS linker is a program development tool that takes the output of language translators (object files or modules), such as the VAX-11 MACRO assembler or the VAX-11 FORTRAN compiler, and produces a file that can be executed on the VAX-11 hardware. This output file is known as an **image**. To write an application in modules, it is necessary to be able to link together the separately compiled modules. The linker is activated by the DCL LINK command, which can be entered interactively or from within a command procedure. Linking consists of three basic operations:

- allocation of virtual memory addresses
- resolution of intermodule symbolic references
- initialization of the contents of a memory image

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

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.

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 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 similar 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 VAX-11 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

After the linker allocates virtual memory and resolves external references, the linker fills in the actual contents of the image. This image initialization consists mainly of copying the binary data and code that was written by the compiler or assembler. However, the linker must perform two additional functions to initialize the image contents:

- It must insert addresses into instructions that refer to externally defined fields. For example, if a module contains an instruction moving FIELDA to FIELDB, and if FIELDB is defined in another module, the linker must determine the virtual address of FIELDB and insert it into the instruction.
- It must compute values that depend on externally defined fields. For example, if a module defines X as being equal to Y plus Z, and if Y and Z are defined in an external module, the linker must compute the value of Y plus Z and insert it in X.

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 VAX/VMS can initialize the VAX memory management hardware to 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 mechanism used to describe images to VAX/VMS 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:

- the starting block number of the image section in the image file
- the starting virtual page number in the process's virtual address space to which to map the image section
- characteristics of the image section, e.g., read-only, read/write
- 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.

Linker 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

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. Compiled 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.

Object Module Libraries

The librarian creates and updates object module library files. Each library file contains a catalog of the object modules and global symbols 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, and is done by loading modules which contain global symbols that resolve undefined global symbols in the link.

Shareable Image Files

A shareable image is an image that comprises part of a complete program. All references in the shareable image are resolved when the shareable image is created. Shareable images are used as input to a later link to create an executable 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.

Linker Output

The linker can produce 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 given 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 are run. A system image is a special type of image intended for stand-alone operation on the hardware i.e., it does not run under the control of the VAX/VMS operating system.

COMMON RUN TIME PROCEDURE LIBRARY

The VAX-11 Common Run Time Procedure Library (RTL) is composed of a set of general purpose 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 nonconflicting 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 is accessed by users via entry point vectors. This allows:

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

Each procedure entry point in the shareable image is at an address defined relative to the base of the shareable section, and will never change, once it has been assigned. New entry points are always added at the end of the list of entry point vectors. The entry point vector contains the procedure entry mask and a transfer of control to the procedure. 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 also permits a new release of the library to be installed without requiring that user images be relinked.

The VAX-11 Run Time Library is designed as a set of modular re-entrant procedures comprising several functional groups. They are:

- a resource allocation group (virtual memory, logical unit number, and event flags)
- a condition handling group (signaling exception conditions and declaring condition handlers)
- a general utility group (data type conversions)
- a mathematical group (single and double precision trigonometric, logarithmic, and exponential functions)
- a language-independent support group (error handling and Record Management Services support functions)
- language-specific support groups (file handling support functions)
- a string handling group (static and dynamic string functions)

Resource Allocation group (LIB$)

The resource allocation group includes all procedures which allow allocation of process-wide resources. Such resources include:

1. Virtual Memory—one procedure to allocate and one to deallocate arbitrary-sized blocks of process virtual memory.

2. Logical Unit Numbers—allow logical unit numbers to be allocated in a modular manner.
3. Event Flags—allow event flags to be allocated in a modular manner.

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. When 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 can override the standard error messages in order to give a more suitable application-oriented user interface.

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 entry points that are called by the JSB instruction.

Language-Independent Support (OTS$)

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

Language-Specific Support (FOR$, BAS$)

Each of the language support libraries is composed of five principal types of procedures:

- I/O processing procedures
- Language-independent initialization and termination
- System procedures
- Compiled-code support procedures
- 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), LIB$ routines provide easy access to the operating system directives.

Compiled-Code Support Procedures

These routines complement the compiled code by performing operations too complicated or too cumbersome to perform directly with inline 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.

VAX-11 SYMBOLIC DEBUGGER

The VAX-11 SYMBOLIC DEBUGGER is a language-independent, interactive program that can be linked with user code written in all native mode languages supported by VAX/VMS. Current languages with which the debugger can be used are: VAX-11 FORTRAN, VAX-11 BASIC, VAX-11 COBOL, VAX-11 BLISS-32, VAX-11 CORAL 66, VAX-11 PASCAL, 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 isolating program errors. Since user program execution is controlled by DEBUG once it is invoked, modifications may be made to the program while it is executing.

The VAX-11 debugger includes many user oriented functions that facilitate the use of the VAX-11 SYMBOLIC DEBUGGER.

- The debugger is interactive—the user maintains control of the program while conversing with the debugger via the terminal.
- The debugger is symbolic—program locations may be referred to by the symbols the user has created in the program. The debugger is also capable of displaying locations as symbolic expressions.
- The debugger supports all native mode languages—the debugger lets the user converse with the program via the source program's language. Furthermore, the user may change languages during the course of a debugging session by means of a simple command.

- The debugger permits a variety of data forms—the user controls the way in which the debugger accepts and displays addresses and data. For example, an address can be represented symbolically, or as a virtual address in decimal, octal, or hexadecimal. Also, data can be represented by symbols, expressions (X+3), VAX-11 MACRO instructions, ASCII character strings, or numeric strings in decimal, octal, or hexadecimal.

DEBUG Commands

DEBUG commands direct the execution of the program and can be entered interactively from a terminal or from an indirect command file. Typically, the DEBUG commands can:

- Specify points at which execution will be suspended, when and if they are encountered, by using the SET BREAK command.
- Trace the sequence of program execution by means of the SET TRACE command. This command establishes tracepoints in the program.
- Display before-and-after values of a location whenever that location is stored into, by means of the SET WATCH command.
- Initiate or resume execution, by means of the GO command or the STEP command.
- Determine the location of breakpoints, tracepoints, and watchpoints by means of the commands SHOW BREAK, SHOW TRACE, and SHOW WATCH, respectively.
- Erase breakpoints, tracepoints, and watchpoints in the program, through use of the CANCEL command.
- Display the contents of memory locations, by using the EXAMINE command.
- Change the value of the contents of memory locations, by using the DEPOSIT command.
- 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.
- Call a subroutine at DEBUG time, by means of the CALL command.
- Change values of parameters for LANGUAGE, SCOPE, MODE, and TYPE.
- Specify an arbitrary file name for the DEBUG log file by means of the SET LOG command.
- Control DEBUG I/O at debug time, via the SET OUTPUT command. This includes normal terminal output, log file output, and command file verification.
- 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.
- Instruct DEBUG to take commands from a specified file by means of @filespec.

THE LIBRARIAN UTILITY

Libraries are indexed files that contain frequently used modules of code or text. There are four types of libraries; object, macro, help, and text. The library type indicates the type of module that the library contains. Each library contains indexes that store information regarding the library's content, including type, and location. The librarian is a utility that allows the user easy access to the data stored in libraries.

The librarian may be invoked in one of two ways; via a set of librarian routines that can be called from the user program directly, or interactively via the DCL command LIBRARY issued from the terminal or from within an indirect command file. The DCL LIBRARY command enables the user to replace and maintain modules in an existing library, or to create a new library. The librarian routines enable an executing program to initialize and open a library, and to retrieve, insert, and delete modules.

The four library types are defined as follows:

- Object libraries (file type OLB) contain frequently called routines and are used as input to the linker. The linker searches the object module library whenever it encounters a reference it cannot resolve from the specified input files.
- Macro libraries (file type MLB) contain macro definitions used as input to the MACRO assembler. The assembler searches the macro library whenever it encounters a macro that is not defined in the input source file.
- Help libraries (file type HLB) contain help modules; that is, modules that provide user information concerning a program. The help message can be retreived by calling the appropriate librarian routines.
- Text libraries (file type TLB) contain any sequential record files required by the user program. A user program can call library routines directly to retrieve text modules.

Librarian Routines

The Librarian utility provides a set of 18 user-callable routines that:

- initialize a library
- open a library
- look up a key in a library
- insert a new key in a library
- return the names of the keys
- delete a key and its associated text
- read text records
- write text records

The user program can call the librarian routines using the VAX-11 standard calling sequence supported in all languages producing VAX-11 native mode code.

DCL LIBRARY Command

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

To invoke the LIBRARY command, enter the following format:

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

For example, to create an object library named TESTLIB, and insert entries ERRMSG, and STARTUP, the user would proceed as follows:

```
$LIBRARY/CREATE TESTLIB        ERRMSG,STARTUP
```

COMMAND LANGUAGE PROCEDURES

A command procedure is a file containing DCL commands, command or program input data, or both. Command procedures may be used to catalog sequences of commands frequently used during an interactive session or to submit all jobs for batch processing.

In its simplest form, a command procedure consists of one or more command lines that the command interpreter executes. In its most complex form, a command procedure resembles a program written in a high level programming language: it can establish loops and error checking procedures, call other procedures, pass values to other procedures and test values set in other procedures, perform arithmetic calculations and input/output operations, and manipulate character string data.

Passing Parameters to Command Procedures

The user can write generalized command procedures that may perform differently each time they are executed. The command interpreter defines eight special symbols for use as parameters within command procedures. These symbols are named P1, P2, P3...P8; they are all initially equated to null strings. Either numeric or character string values for these parameters may be passed when executing the procedure with the @ command or the SUBMIT command when entering a batch job.

For example, the procedure named EXECUTE contains the following lines:

```
$ IF P2 .EQS. "" THEN $P2:="FORTRAN"
$ 'P2' 'P1'
$ LINK 'P1'
$ RUN 'P1'
```

The command procedure EXECUTE accepts both the language compiler and the user program name as input. If the user executes the procedure with the @ command, the values for the command parameters P1 and P2 would be entered as follows:

```
$ @EXECUTE PAYROLL COBOL
```

In this sample run, the user chose the program name PAYROLL, and the COBOL compiler.

It is also possible to define a symbol as a local symbol, using a single equals sign (=) in an assignment statement. For example, the user might have equated the symbol EXE to the execution command @EXECUTE as follows:

```
$ EXE*CUTE:=@EXECUTE
```

The asterisk (*) specifies that EXE, EXEC, EXECU, etc. are abbreviations of EXECUTE. The minimum abbreviation is three characters (in this case, EXE). A colon (:) in an assignment statement indicates a character string assignment. Now to execute the command procedure the use can enter the following:

```
$ EXE STRESS
```

In this run, STRESS is the user program name and the compiler is the default compiler, FORTRAN (i.e., the second parameter in the EXE command was left blank).

Logical Commands

Normally, the command interpreter executes each command in a command procedure in sequential order, and terminates processing when it reaches the end of the command procedure file. However, by using combinations of the logical commands, the user can alter the flow of execution of the command procedure. By using the IF, GOTO, ON, EXIT, and STOP commands, the user can control the execution sequence, conditionally execute lines, construct loops, and handle errors.

Lexical Functions

The command interpreter recognizes a set of functions, called lexical functions, that return information about character strings and attributes of the current process. Lexical functions may be used in any context in which symbols and expressions are used. Within command procedures, lexical functions are used to translate logical names, perform character string manipulations, and determine the current processing mode of the procedure.

Command Procedure Example

The command procedure described in Figure 8-3, when invoked, locks up a user terminal as being in use. If the current user must leave the terminal for some time and does not wish to have it disturbed, the user can invoke the command procedure INUSE.COM, rendering the terminal inaccessible to any other user not knowing the access password.

This command procedure illustrates several of the powerful features of DCL, including:

- Trapping of the Control-Y function.
- Calling a command procedure from within a command procedure (i.e., @COMMANDS:INUSE.TXT). ERASE, ERASELINE, and TEXT are user-defined symbols that also invoke command procedures.
- Referencing lexical functions (i.e., 'F$TIME, 'F$LOCATE, and 'F$EXTRACT).

Upon invoking the command procedure INUSE.COM:

- The current setting of VERIFY is retrieved via the lexical function 'F$VERIFY, and stored in local variable VER (line 100).
- The procedure will set NOVERIFY (line 200), i.e., the command lines are not echoed on the terminal during execution (if verify was on when the command procedure was invoked, it will be turned on when the procedure exits successfully).
- The address of the password routine (line 900) is stored for later use if Control-Y is typed.
- The address of ERR_HNDLR (line 950) is stored for processing any errors that might occur.

Execution then proceeds with the BEGIN code block (lines 1000-1300). The ERASE procedure (line 1100) is called, which clears the screen of all text. INUSE.COM then calls the command procedure INUSE.TXT (line 1300). This procedure prints in block letters, "IN USE," across the video screen. Execution then proceeds to the LOOP section of code (lines 1500-2300). This block of code retreives the current date and time of day from VMS, using the lexical function 'F$TIME. The date and time of day normally appear as follows:

```
dd-mmm-yyyy hh:mm:ss.cc
```

The 'F$LOCATE and 'F$EXTRACT lexical functions operate upon the date and time of day, reducing the time quantity to hours and seconds only. Therefore the final date and

```
100     $     VER='F$VERIFY()
200     $     SET NOVERIFY
300     $
400     $!    THIS COMMAND PROCEDURE LOCKS A TERMINAL AS BEING IN USE. IT DISABLES
500     $!    CONTROL Y, SETS A CONTROL Y ENTRY POINT AND LOOPS ON A SHOW TIME
600     $!    COMMAND. A CONTROL Y TYPED AT THE TERMINAL WILL TRANSFER CONTROL
700     $!    TO A CHECK FOR A PASSWORD TO EXIT FROM THIS PROCEDURE.
800     $
900     $     ON CONTROL Y THEN $GOTO PASSWORD
950     $     ON ERROR THEN GOTO ERR_HNDLR
1000    $BEGIN:
1100    $     ERASE
1200    $
1300    $     COMMANDS:INUSE.TXT
1400    $
1500    $LOOP:
1600    $     TIMSTR:='F$TIME()
1700    $     DOT='F$LOCATE(".",TIMSTR)
1800    $     DOT=DOT-3                                  !SHOW TIME DOWN TO MINUTES
1900    $     TIMSTR:='F$EXTRACT(0,DOT,TIMSTR)
2000    $     TEXT 5 32 "'TIMSTR'"
2100    $     TEXT 1   1
2200    $     WAIT 00:01
2300    $     GOTO LOOP
2400    $
2500    $PASSWORD:
2600    $     TEXT 1   1
2700    $     INQUIRE MAGIC "ENTER THE PASSWORD TO CONTINUE"
2800    $     IF "'MAGIC'" .EQS. "'P1'" THEN $GOTO EXIT
2900    $     IF "'MAGIC'" .EQS. "REFRESH" THEN $GOTO BEGIN
3000    $     ERASELINE 1
3100    $     ERASELINE 2
3200    $     ERASELINE 3
3300    $     ERASELINE 4
3400    $     ERASELINE 5
3500    $     ERASELINE 6
3600    $     ERASELINE 7
3700    $     GOTO LOOP
3800    $
3801    $   ERR_HNDLR:
3802    $     ON ERROR THEN GOTO ERR_HNDLR
3803    $     GOTO PASSWORD
3900    $EXIT:
4000    $     SET CONTROL Y
4100    $     ERASE
4200    $     IF VER THEN $SET VERIFY
4300    $     EXIT
4400    $
4500    $!    END OF INUSE.COM
```

Figure 8-3

Example Command Procedure

time of day appear as:

dd-mmm-yyyy hh:mm

This date and time of day function are printed on the screen above the "IN USE" message. The date and time of day are refreshed once every minute. The PASSWORD code (lines 2500-3700) is entered only if a Control-Y is typed at the terminal while the terminal is locked up. The command procedure prompts for a password. If the entered password matches the initial password (declared by the current user of the terminal), the flow of execution drops to the EXIT code block (lines 3900-4300). If the passwords do not match, execution drops through to line 3000, which clears the top seven lines of the screen.

Another added feature is the REFRESH statement (line 2900). The REFRESH statement directly follows the PASSWORD check statement (line 2900). If the screen gets cluttered with garbage characters, any user may enter a CONTROL Y, and in response to the system prompt for a password (line 2700), type REFRESH. REFRESH is recognized in line 2900, clearing the entire screen of unwanted characters. This is followed by a new IN USE and date and time of day message.

DIFFERENCES UTILITY

By invoking the DIFFERENCES command, the user can determine if two files are identical and, if not, how they dif-

fer. The DIFFERENCES utility compares the contents of two disk files on a record-by-record basis and creates a listing of the records that do not match. By default, the DIFFERENCES utility compares every character in each file, and by default, the DIFFERENCES utility writes the output in ASCII.

VAX-11 RUNOFF

VAX-11 RUNOFF is a document formatter. A RUNOFF-processed document can be updated without extensive retyping because text changes, via 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. Executing in default mode, RUNOFF provides:

- a standard typewriter page size of 8½″ × 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

The output file is the print-ready document. After RUNOFF has processed the file, the original file remains available for further editing.

VAX-11 RUNOFF contains commands to perform the following functions:

- filling and justifying text
- page formatting
- title formatting
- subject-matter formatting
- graphic, list and note formatting
- index and table of contents
- miscellaneous formatting

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.

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.

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; and determining the number of header levels (up to six) that the document will have are all provided by the title formatting commands.

Subject-Matter Formatting

Subject-matter formatting commands include 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.

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 indexes and tables of contents.

Miscellaneous Formatting

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

9 Data Management Facilities

```
ENVIRONMENT DIVISION.
[INPUT-OUTPUT SECTION.]
FILE-CONTROL.
SELECT file-name
    ASSIGN TO device-name-1    [, device-name-2]    . . .
    ; ORGANIZATION IS INDEXED
[ ; ACCESS MODE IS { SEQUENTIAL } ]
                   { RANDOM     }
                   { DYNAMIC    }
    ; RECORD KEY IS data-name-1
[ ; ALTERNATE RECORD KEY IS data-name-2  [WITH DUPLICATES] ]

DATA DIVISION.
[FILE SECTION.
[FD file-name
[ ; BLOCK CONTAINS   [integer-1 TO]   integer-2  { RECORDS    } ]
                                                { CHARACTERS }
[ ; RECORD CONTAINS   [integer-3 TO]   integer-4 CHARACTERS ]
    ; LABEL { RECORD IS  } { STANDARD }
            { RECORDS ARE } { OMITTED  }
[ ; DATA { RECORD IS  } data-name-3 [, data-name-4] . . . ]
         { RECORDS ARE }

PROCEDURE DIVISION.
OPEN { INPUT file-name-1   [, file-name-2]   . . . }
     { OUTPUT file-name-3  [, file-name-4]  . . . } . . .
     { I-O file-name-5     [, file-name-6]  . . . }
```

VAX/VMS data management includes a file system that provides volume structuring and protection, and record management services that provide device-independent access to the VAX peripherals.

The VAX/VMS on-disk structure provides a multilevel hierarchy of named directories and subdirectories. Files can extend across multiple volumes and be as large as the volume set on which they reside. Volumes are mounted to identify them to the system. VAX/VMS also supports multivolume ANSI format magnetic tape files with transparent volume switching.

The VAX/VMS record management input/output system (RMS) provides device-independent access to disks, tapes, unit record equipment, terminals, and mailboxes. RMS allows user and application programs to create, access, and maintain data files with efficiency and economy. Under RMS, records are regarded by the user program as logical data units that are structured and accessed in accordance with application requirements.

RMS provides sequential record access to sequential file organizations, sequential, random, or combined record access to relative file organizations and sequential, random, or a combination using index key access to multikey indexed files. Multikey indexed file processing includes incremental reorganization.

VAX/VMS also supports several other data management facilities: DATATRIEVE, VAX-11 SORT, and the Forms Management System (FMS) utility package.

INTRODUCTION

The operating system's data management services are provided by the following facilities:

- utilities for data and file manipulation and inquiry
- file system
- record management services
- device drivers
- command interpreter

Utilities which VAX offers include the VAX-11 SORT/MERGE for reordering data, DATATRIEVE for data inquiry and report writing, and FMS for screen formatting and forms generation.

The file system provides volume structuring and directory access to disk and magnetic tape files. Programmers can use the file system as a base to build their own record processing system, or they can use the VAX/VMS record management services.

The record management services (RMS) provide device-independent access to all types of I/O peripherals. The RMS procedures enable a program to access records within files, and provide the same programming interface regardless of device characteristics. The system includes utilities for RMS file creation and maintenance.

The device drivers provide the basic I/O device handling for all of the other data management services. Device drivers and their features are described in the Peripherals and Operating System sections.

As described in the Users section, the command interpreter enables a user to reserve devices for exclusive use, set device and directory name defaults, and assign logical names to file specifications. The command interpreter also enables the user to execute file management utilities that provide file copy, transfer, and conversion operations.

The following paragraphs discuss some of the features and functions of the file system, including the file structures, file naming facilities, and the file management utility programs. The remainder of this section describes the record management services programming environment, and utilities for high-level data and file manipulation.

FILE MANAGEMENT

VAX/VMS provides two file structures: one for disk volumes and one for magnetic tape volumes. From the user's point of view, the only differences between the two file structures are those imposed by the capabilities of the media. Volumes are mounted for identification, and files can extend across multiple volumes. The practical limit to file size is that they can be only as large as the volume set on which they reside.

Volume and file protection are based on User Identification Codes (UICs) assigned to accessors and the file or volume. The UICs establish the accessor's relationship to the data structure as owner, the owner's group, the system, or the world (all others). Depending on the relationship, the accessor may or may not have read, write, execute, or delete access to any given file.

Disk volumes are multiuser volumes. They can contain a multilevel directory hierarchy that is defined dynamically by the users of the volume. The on-disk file structure appears to a program to be a virtually contiguous set of blocks. The blocks of the file, however, may be scattered anywhere on a volume. Mapping information is maintained to identify all the blocks constituting a file. Figure 9-1 illustrates the file structure.

Disk files can be extended easily. The blocks of the file are allocated in physically contiguous sets, called **extents**. Users are not required to preallocate space, although they can do so. Users can specify placement on an allocation request, and they can control automatic allocation. For example, when a file is automatically extended, it can be extended by any given number of contiguous blocks. If desired, a file can be created as a contiguous file, in which case it is both virtually and physically contiguous.

The disk structure includes duplicates of its critical volume information. The system detects bad disk blocks dynamically and prevents re-use once the files to which they are allocated are deleted.

Magnetic tape volumes are single-user volumes. Magnetic tape files consist of physically contiguous blocks. Record blocking is under program control. Files have ANSI format labels. VAX/VMS also supports unlabeled (non-file-structured) magnetic tapes.

File Directories and Directory Structures

A directory is a file containing a list of files on a given volume. A directory entry contains the name, type, version, and unique file ID for a particular file. A directory can list files having the same owner UIC or files having different owner UICs. The entries are listed alphabetically.

A disk volume contains at least one directory, called the master file directory. The system manager is responsible for creating a volume's master file directory. The master file directory can (and normally does) contain a list of directory files which form a second level of directories. The second level of directory files can list data files and/or other directory files, called **subdirectories**. Users can create subdirectories within the directories they own. The subdirectories can also list other directory files and/or data files. Figure 9-2 illustrates a multilevel directory structure.

Since directories of files on volumes are files themselves, they are assigned owner UICs and can be protected from certain kinds of access depending on the relationship established by an accessor's UIC. In the special case of directory files, the file protection fields control an accessor's ability to:

- look up files
- enter new files in the directory, including new versions of existing files
- remove files from the directory

File Specifications

A **file specification** identifies which file is to be used in a file processing operation. Programs use file specifications to identify the file they want to create, access, delete, or extend, and users supply the command interpreter with a file specification to identify the file they want to edit, compile, copy, delete, etc. A complete file specification is a well-defined character string composed of the following fields:

- *Node Name* — The node of the network in which the volume containing the file is stored. The node name is

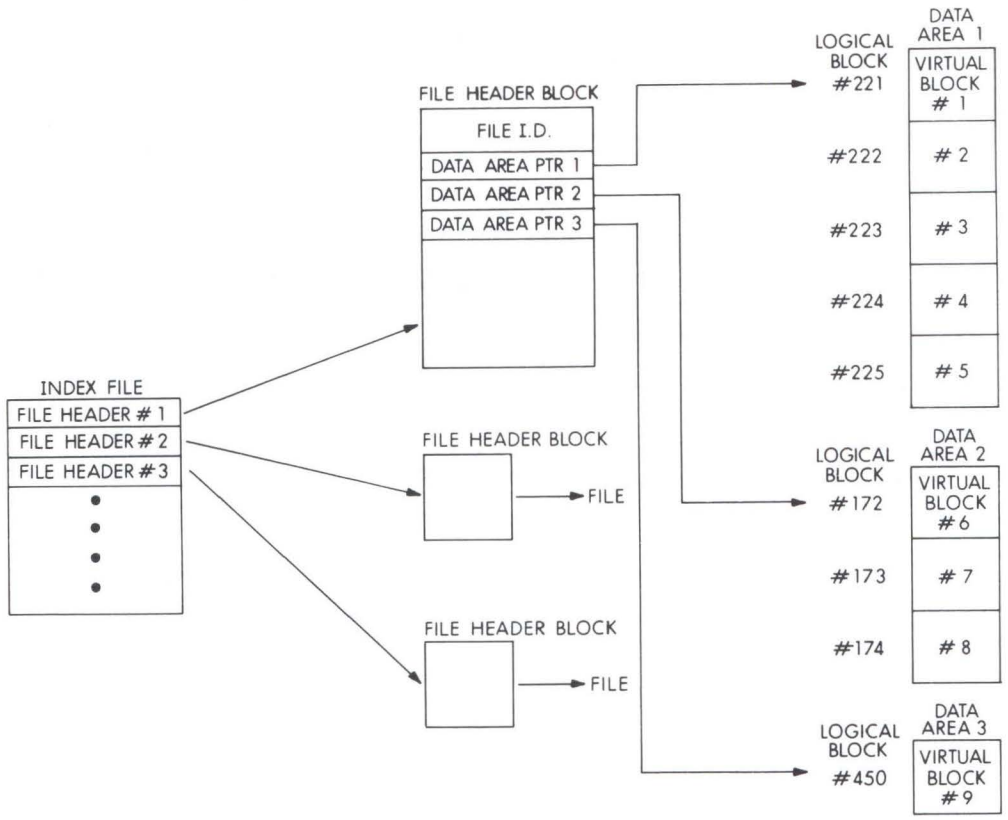


Figure 9-1

Disk File Structure

followed by two colons (::) to delimit it from the remainder of the file specification.

- *Device Name* — The device on which the volume containing the file is mounted. The device name is followed by a single colon (:) to delimit it from the remainder of the file specification.
- *Directory Name* — The directory in which the file is listed. A directory name begins with an opening bracket (< or [) and ends with a closing bracket (> or]). If the file is listed in a subdirectory, the directories to be searched are listed in the desired search order, with the names separated by periods, e.g.:

  [name1.name2.name3]

- *File Name* — The user-assigned name of the file.
- *File Type* — The type identification for the file. The type is preceded by a period (.) to delimit it from the remainder of the file specification.
- *File Version* — the generation number of the file. The file version is preceded by a semicolon (;) or period (.) to delimit it from the remainder of the file specification.

For example, a complete file specification might be:

NODE47::DBA1:[JONES]HANOI.FOR;2

In this case, NODE47 is the name of the network node, DBA1 is the name of the device (DB for disk pack device, A for disk controller, 1 for drive unit number), [JONES] is the directory name, HANOI is the file name, FOR is the file type (meaning that the file is a FORTRAN source file), and 2 is the version number.

Neither programs nor command language users need to provide a complete file specification to identify files. The system applies defaults to most fields of a file specification when they are not present. For example, if the node name is not present, the node is assumed to be the node on which the program is executing. If the version number is not present, the version is always assumed to be the latest version. Device name and directory name defaults for users and the programs they execute are supplied by the system manager in the user authorization file, and users can change the standard defaults at any time during their session on the system.

Some commands (such as COPY, PRINT, and DELETE) accept a wild card in one or more fields of a file specification. A wild card is an asterisk appearing in a file specification field and it means "all."

File specifications also apply to non-file-structured devices such as line printers, card readers, and terminals. In these cases, however, the user or program needs to supply only the node name and device name, as appropriate.

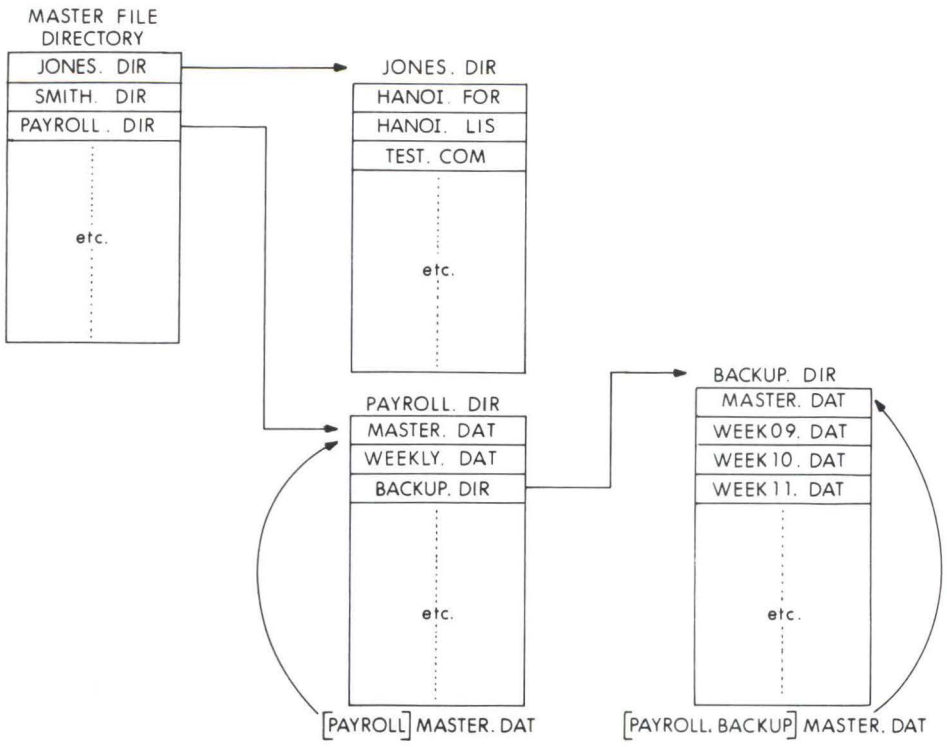


Figure 9-2

Multilevel Directory Structure

Logical File Naming

To provide both system and device independence, users and programs are not limited to identifying files by their file specifications. They can use **logical names** in place of a complete file specification, or in place of a portion of a file specification. For example, a user can assign a logical name to the left-most three fields of a file specification:

$ ASSIGN NODE47::DBA4:[JONES] *to* VOL

And then use the logical name VOL in a subsequent command:

$ TYPE VOL:HANOI.FOR

Defaults also apply when translating logical names, so that the user could have made the assignment:

$ ASSIGN NODE47::[JONES] *to* VOL

In this case, the user's default device name would be used to derive the complete file specification.

Logical name assignments can be made on a process, group, or system-wide basis. Logical names can also be recursive, that is, a logical name can be assigned to another logical name, or to a logical name and a portion of a file specification.

For example, suppose a company's weekly payroll production run includes an application program that uses the current week's payroll changes data file. That data file may be located in the directory named [PAYROLL] one week, or in the payroll backup subdirectory, [PAYROLL.BACKUP], another week. The volume on which the file is stored may be mounted on disk pack drive unit number 1 one week, or on unit 2 another week.

The application programmer can write the program without knowing which directory the data file is listed in, or which device the volume is mounted on. A series of logical name assignments provides the complete file specification. The assignments are the responsibility of the people who know what directory the file is listed in, and what drive the volume is mounted on.

In the example shown in Figure 9-3, the application program contains an OPEN statement for the payroll data file using the logical name WEEKLY_PAYROLL_CHANGES (note that underscore is a legal character). The application systems designer has created a command procedure file called PAYRUN that controls the production run. The command procedure file includes a logical name assignment that obtains the file name as a parameter supplied by the operator or production clerk who starts the production run. The logical name used by the application program is given a value that consists of another logical name (WEEKLY_PAYROLL) and the file name and type specifications.

To complete the series of logical name assignments, the payroll group operations manager makes a group-wide logical name assignment: the payroll data files this week are stored in the PAYROLL.BACKUP subdirectory. The logical name assignment provides the directory name, using another logical name (PAY_PACK) known to the oper-

Application Programmmer:

```
OPEN ("WEEKLY_PAYROLL_CHANGES")
```

Application System Programmer:

```
Command Procedure: PAY_RUN.COM
accepts one parameter (P1): Week Number
$ ASSIGN WEEKLY_PAYROLL:'P1'.WPY     WEEKLY_PAYROLL_CHANGES
$ RUN APPLICATION
```

Production Clerk:

```
$ @PAY_RUN WEEK09
```

Payroll Group Operations Manager:

```
$ ASSIGN/GROUP PAY_PACK:[PAYROLL.BACKUP]     WEEKLY_PAYROLL
```

Local Operator:

```
$ ASSIGN/SYSTEM DBA2:     PAY_PACK
```

Figure 9-3
Logical Naming

ator who mounts the payroll data files volume. The operator makes the system-wide logical name assignment when mounting the pack before the production run. Given the assignments shown in the example, the logical name used to open the file is translated to:

DBA2:[PAYROLL.BACKUP]WEEK09.WPY

(The local system node name and the latest version number are used as defaults to complete the file specification.) Should the directory name change, or the pack be mounted on another device that day, the only changes made are the logical name assignments. There is no need to modify either the application program or the command procedure controlling the production run.

File Management

The VAX/VMS system includes many services that aid in data management and maintenance. Some of these are described in the following paragraphs.

Sorting Files — The SORT/MERGE program allows the user to rearrange, delete, and reformat records in a file. The user can arrange the records in the ascending or descending sequence of one or more fields within the records for subsequent sequential processing. SORT can also create several different index files for accessing a file according to these indexes without reordering the data itself.

Comparing Files — A file differences command contrasts two files by automatically aligning matching text, and optionally ignoring comments, empty records, trailing blanks, or multiple blanks. The output can be a file-by-file list of differences, an interleaved list of differences, a list with change bars, or a batch editor command input file.

Backing Up Files and Volumes — The Disk Save and Compress (DSC) utility enables a user to back up entire disk volumes to magnetic tape or to other disks. When backing up disk volumes to other disk volumes, or restoring disk volumes from magnetic tape, DSC combines unused blocks on disks into contiguous areas.

Verifying File Structures — The file verification utility checks the consistency and accuracy of the file structure on a Files-11 disk volume. It can also display the number of available blocks in a volume, locate files that could not otherwise be accessed, and list the names of files on the volume.

Bad Block Locator — The bad block locator utility determines the number and location of bad blocks on Files-11 disk volumes and stores this information in the bad block file on the volume so that the blocks can not be allocated. Running this utility before initializing a Files-11 volume is useful in ensuring a disk's integrity.

RMS Utilities — The record management services procedures are complemented by a number of utilities designed especially for RMS file creation and maintenance. They allow the user to:

- create an RMS file and define the attributes of the file
- list the attributes of a single file or a group of files, or list the contents of a backup magnetic tape
- convert a file with any file organization or record format to a file with any other file organization or record format
- back up a single file or group of files in a compact format (optionally by creation or revision date)
- restore files previously backed up (optionally by creation or revision date)

RECORD MANAGEMENT SERVICES

The record management services (RMS) are a set of system procedures that provide efficient and flexible facilities for data storage, retrieval, and modification. When writing programs, the user can select processing methods from among the RMS file organizations and accessing techniques. The following sections discuss RMS:

- file organizations
- file attributes
- program operations
- run-time environment

The manner in which RMS builds a file is called its organization. RMS provides three file organizations:

- sequential
- relative
- indexed

All three file organizations are available in both compatibility mode (using RMS-11) and in native mode (using VAX-11 RMS).

The organization of a file establishes the techniques one can use to retrieve and store data in the file. These techniques are known as record access modes. The record access modes that RMS supports are:

- sequential
- random
- Record's File Address (RFA)

An application program or a RMS utility can be used when creating a RMS file to specify the organization and characteristics of the file. Among the attributes specified are:

- storage medium
- file name and protection specifications
- record format and size
- file allocation information

After RMS creates a file according to the specified attributes, application programs can store, retrieve and modify data. These program operations take place on the logical records in a file or the blocks comprising the file.

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 might constitute a single record. Each record in the personnel file would be subdivided into discrete pieces of information known as fields. The user defines the number, locations within the record, and logical interpretations of these fields.

The user can completely control the grouping of fields into records and records into files. The relationship among fields and records is embedded in the logic of the programs. RMS does not know the logical relationships that exist within the information in the files.

RMS ensures that every record written into a file can subsequently be retrieved and passed to a requesting program as a single logical unit of data. The structure, or organization, of a file establishes the manner in which RMS stores and retrieves records. The way a program requests the storage or retrieval of records is known as the record access mode. The organization of a file determines which record access modes can be used.

Sequential File Organization

In sequential file organization, records appear in consecutive sequence. The order in which records appear is the order in which the records were originally written to the file by an application program or RMS utility. Sequential organization is the only file organization permitted for magnetic tape and unit record devices. Most VAX/VMS system utilities that deal with files, deal with sequentially organized files. All system editors and language processors, for instance, operate on sequentially organized files. Figure 9-4 illustrates sequential file organization.

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 maximum length permitted for a record in the file. These cells are numbered from 1 (the first) to n (the last). A cell's number represents its location relative to the beginning of the file.

Each cell in a relative file can contain a single record. There is no requirement, however, that every cell contain a record. Empty cells can be interspersed among cells containing records. Figure 9-5 illustrates a 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 forth. 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

Figure 9-4

Sequential File Organization

Figure 9-5

Relative File Organization

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 field present in every record of an indexed file. The location and length of this field are identical in all records. When creating an indexed file, the user decides which field or fields in the file's records are to be a key. Selecting such fields indicates to RMS that the contents (i.e., key value) of those fields in any particular record written to the file can be used by a program to identify that record for subsequent retrieval.

At least one key must be defined for an indexed file: the primary key. 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 programs 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. Stored 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 9-6 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.

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 record access mode during the processing of a file. RMS permits only certain combinations of file organization and record access mode. Table 9-1 lists these combinations.

Sequential Record Access Mode

Sequential record access mode can be used to access all RMS files and all record-oriented devices, including mailboxes. Sequential record access means that records are retrieved or written in the sequence established by the organization of the file.

Sequential Access to Sequential Files — When using sequential record access mode in a sequentially organized file, physical arrangement establishes the order in which records are retrieved. To read a particular record in a file, say the fifteenth record, a program must open the file and access the first fourteen records before accessing the desired record. Thus each record in a sequential file can be retrieved only by first accessing all records that physically precede it. Similarly, once a program has retrieved the fifteenth record, it can read all the remaining records (from the sixteenth on) in physical sequence. It cannot, however, read any preceding record without closing and reopening the file and beginning again with the first 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 record 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 record 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 rec-

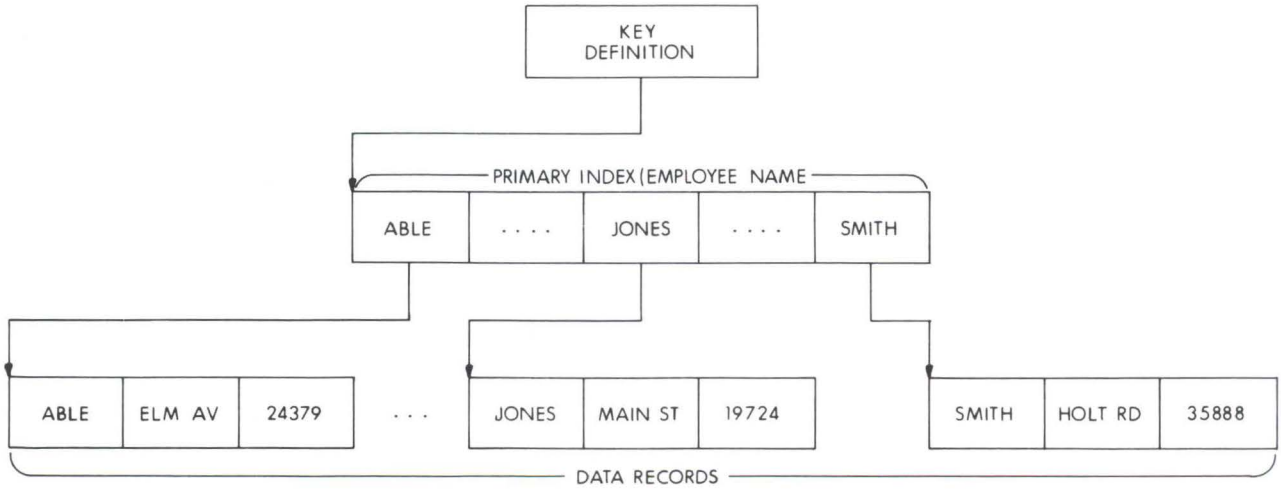


Figure 9-6

Indexed File Organization

Table 9-1
Record Access Modes and File Organizations

File Organization	Record Access Mode			
	Sequential	Random		RFA
		Record #	Key Value	
Sequential	Yes	Yes[2]	No	Yes[1]
Relative[1]	Yes	Yes	No	Yes
Indexed[1]	Yes	No	Yes	Yes

1 Disk files only.
2 Fixed length record format disk files only.

ords from an indexed file in the order represented by any index. The entries in an index are arranged in ascending order by key values. If more than one key is defined for the file, each separate index associated with a key represents a different logical ordering of the records in the file.

When reading records in sequential record access mode from an indexed file, a program initially specifies a key (primary key, first alternate key, second alternate key, etc.) to RMS. Thereafter, RMS uses the index associated with that specified key to retrieve records in the sequence represented by the entries in the index. Each successive record RMS returns in response to a read request contains a value in the specified key field that is equal to or greater than that of the previous record returned.

When writing records to an indexed file, RMS uses the definition of the primary key field to place the record in the file.

Random Record Access Mode

In random record access mode, the program establishes the order in which records are processed. Each program request for access to a record operates independently of the previous record accessed. Each request in random record access 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 notifies 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 notifies the program.

Random Record Access to Indexed Files — For indexed files, a key value rather than a relative record number identifies the record. Each program read request in random record 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 indexes. In this way RMS ensures that the record can be retrieved by any of its key values.

Record's File Address (RFA) Record Access Mode

Record's File Address (RFA) record access mode can be used to retrieve records in any file organization as long as the file resides on a disk volume. Like random record access mode, RFA record access allows a specific record to be identified for retrieval, using the record's unique address. The actual format of this address depends on the organization of the file.

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. 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 record access mode to another while processing a file. For example, a program can access a record randomly, then switch to sequential record 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 record access mode selected.

FILE AND RECORD ATTRIBUTES

When creating an RMS file, a program or user defines its logical and physical characteristics, or attributes. 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. The program or user also defines or selects other attributes, including:

- device
- file size
- file location
- record format and size
- keys (for indexed files only)

Selection of device 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 line printers. 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 to specific locations on the disk to optimize disk access time. The file's starting location can be specified optionally using a volume-relative block number, or a physical cylinder address.

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, 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 particular data fields within 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 key field data types.

- string
- signed 15-bit integer
- unsigned 16-bit binary
- signed 31-bit integer
- unsigned 32-bit binary
- packed decimal

The string key field can be composed of simple or segmented keys. A simple key is a single, contiguous string of characters in the record; in other words, 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 only be simple keys.

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.

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. If key values can change, the user must also specify that the duplicate

key values are allowed. If the primary key value can change, the user may not change the record length.

Figures 9-7 and 9-8 show excerpts from a COBOL program which operates upon an indexed customer information file via the dynamic access method. The program searches through the file and generates various reports based upon the customer's financial status and additional input typed in by the user at the terminal. In Figure 9-7, the program describes the organization of the file and specifies the access method to be used. In Figure 9-8, the program searches for the first non-zero customer number. Using the "approximate key" match facility (greater than), the program searches for the first non-zero customer. When RMS has located the first non-zero customer number, the program changes access method and the file is read sequentially.

```
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT CUSTOMER-FILE
        ASSIGN TO "CUSTOM.DAT"
        ORGANIZATION IS INDEXED
        ACCESS MODE IS DYNAMIC
        RECORD KEY IS CUST-CUSTOMER-NUMBER
        ALTERNATE RECORD IS KEY IS CUST-CUSTOMER-NAME
        FILE STATUS IS CUSTOMER-FILE-STATUS.
```

Figure 9-7

ISAM File Description

```
        OPEN INPUT CUSTOMER-FILE.
        MOVE "000000" TO CUST-CUST-NUMBER.
        START CUSTOMER-FILE.
                KEY IS > CUST-CUST-NUMBER.
        OPEN OUTPUT STATEMENT-REPORT.
**************************************************
MAINLINE SECTION.
SBEGIN.
        READ CUSTOMER-FILE NEXT
                AT END
                        GO TO END-PROCESS.
        ADD 1 TO RECORD-COUNT.
```

Figure 9-8

Dynamic Access Processing

PROGRAM OPERATIONS ON RMS FILES

After RMS has created a file according to the user's description of file characteristics, 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, that is, independent of record processing, a program can:

- create a file
- open an existing file
- modify file attributes
- extend a file
- close the 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.

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:

- Read a record. RMS returns an existing record within the file to the program.
- Write 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 the sequential file organization.
- Update a record. The program modifies the contents of a record in 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, RFA, or, if the file contains fixed-length records, random record access mode. New records can be added only to the end of the file and only through the use of sequential or random record access mode.

The find operation is supported in both sequential and RFA record access modes. In sequential record access mode the program can use a find operation to skip records. In RFA record access mode, the program can use the find operation to establish a random starting point in the file for sequential read operations.

The sequential file organization does not support the delete operation, since the structure of the file requires that records be adjacent in and across virtual blocks. A program can, however, update existing records in sequential disk files as long as the modification of a record does not alter its size.

Relative File Record I/O — Relative file organization permits programs greater flexibility in performing record operations than does sequential organization. A program can read existing records from the file using sequential, random, or RFA record access mode.

New records can be sequentially or randomly written as long as the intended record cell does not already contain a record. Similarly, any record access mode can be used to perform a find operation. After a record has been found or read, RMS permits the delete operation. Once a record has been deleted, the record cell is available for a new record. A program can also update records in the file. If the format of the records is variable, update operations can modify record length up to the maximum size specified when the file was created.

Indexed File Record I/O — Indexed file organization provides the greatest flexibility in performing record operations. A program can read existing records from the file in sequential, RFA, or random record access mode. When reading records in random record access mode, the program can choose one of four types of matches that RMS performs using the program-provided key value. The four types of matches are:

- exact key match
- approximate key match
- generic key match
- approximate and generic key match

Exact key match requires that the contents of the key in the record retrieved precisely match the key value specified in the program read operation.

The approximate match facility allows the program to select either of the following relationships between the key of the record retrieved and the key value specified by the program:

- equal to or greater than
- greater than

The advantage of this kind of match is that if the requested key value does not exist in any record of the file, RMS returns the record that contains the next higher key value. This allows the program to retrieve records without knowing an exact key value.

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

RMS also allows any number of new records to be written into an indexed file. It rejects a write operation only if the value contained in a key of the record violates a user-defined key characteristic (e.g., duplicate key values not permitted).

The find operation, similar to the read operation, can be performed in sequential, RFA, or random record access mode. When finding records in random record 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. The only restriction RMS applies during an update operation is that the contents of the modified record must not violate any user-defined key characteristic (e.g., key values cannot change and duplicate key values are not permitted).

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 record access modes, a program can process a file as a structure consisting solely of blocks.

Using block I/O, a program reads or writes blocks by identifying a starting virtual block number in the file and a transfer length. Regardless of the organization of the file, RMS accesses the identified block or blocks on behalf of the program.

Since RMS files, particularly relative and indexed files, contain internal information meaningful only to RMS itself, DIGITAL does not recommend that a file be modified by using block I/O. The presence of the block I/O facility, however, does permit 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 has two levels, the file processing level and the record processing level.

At the file processing level, RMS and the operating system provide an environment permitting 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 allowing programs to exercise as little or as much control over buffer space requirements for file processing as desired.

At the record processing level, RMS allows programs to access records in a file through one or more record access streams. 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 to/from an I/O buffer from/to a program-provided location. Locate mode allows programs to process retrieved records directly in an I/O buffer.

Run Time File Processing

RMS allows executing programs to share files rather than requiring them to process files serially. The manner in which a file can be shared depends on the organization of the file. Program-provided information further establishes the degree of sharing of a particular file.

File Organization and Sharing — With the exception of magnetic tape files, which cannot be shared, an RMS file can be shared by any number of programs that are reading, but not writing, the file. Sequential disk files can be shared by multiple readers and multiple writers, but they are responsible for any record locking required to handle multiple readers and writers properly.

Program Sharing Information — A program specifies what kind of 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 this program.

These two types of information allow 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.

Buffer Handling — To a program, record processing under RMS appears as the direct movement of records between a file and the program itself. Transparently to the program, however, RMS reads or writes the blocks of a file into or from internal memory areas known as I/O buffers. Records within these buffers are then made available to the program. Users can control the number and size of buffers. For sequential record access, users can choose an optional I/O read-ahead and write-behind buffer management. For magnetic tape file access, they can control the number of buffers for multiple buffering. For sequential disk files, users can specify the number of blocks that are to be transferred whenever RMS performs an I/O operation.

Run Time Record Processing

After opening a file, a program can access records in the file through the RMS record processing environment. This environment provides three facilities:

- record access streams
- synchronous or asynchronous record operations
- record transfer modes

Record Access Streams — In the record processing environment, a program accesses records in a file through a record access stream. A record access stream is a serial sequence of record operation requests. For example, a program can issue a read request for a particular record, receive the record from RMS, modify the contents of the record, and then issue an update request that causes RMS to write the record back into the file. The sequence of read and update record operation requests can then be performed for a different record, or other record operations can be performed, again in a serial fashion. Thus, within a record access stream, there is at most one record being processed at any time.

For relative and indexed files, RMS permits a program to establish multiple record access streams for record operations to the same file. The presence of such multiple record access streams allows programs to process in parallel more than one record of a file. Each stream represents an independent and concurrently active sequence of record operations.

As an example of multiple record access streams, a program could open an indexed file and establish two record access streams to the file. The program could use one record access stream to access records in the file in random access mode through the primary index. At the same time, the program could use the second record access stream to access records sequentially in the order specified by an alternate index.

Synchronous and Asynchronous Record Operations — Within each record access stream, a program can perform any record operation either synchronously or asynchronously. When a record operation is performed synchronously, RMS returns control to a program only after the record operation request has been satisfied (e.g., a record has been read and passed on to the program).

If the programming language allows asynchronous processing, RMS can return control to a program before the

record operation request has been satisfied. A program can use the time required for the physical transfer to perform other computations. A program cannot, however, issue a second record operation through the same stream until the first record operation has completed. To ascertain when a record operation has actually been performed, a program can specify completion routines or issue a wait request and regain control when the record operation is complete.

Record Transfer Modes — A program can use either of two record transfer modes to gain access to each record in memory:

- move mode
- locate mode

Move mode means that the individual records are copied between the I/O buffer and a program. For read operations, RMS reads a block into an I/O buffer, finds the desired record within the buffer, and moves the record to a program-specified location in its work space. For write operations, the program builds or modifies a record in its own work space and RMS moves the record to an I/O buffer. RMS supports move mode record operations for all file organizations.

Locate mode enables programs to read records directly in an I/O buffer. Locate mode reduces the amount of data movement, thereby saving processing time. RMS provides the program with the address and size of the record in the I/O buffer. RMS supports locate mode record transfers on all file organizations for read operations only.

RMS Record Locking

VAX-11 RMS provides a record locking capability for files that use the relative and indexed organization. In addition, RMS record locking is supported for sequential files with 512-byte fixed length records. This provides control over operation when the file is being accessed simultaneously by more than one program and/or more than one stream in a program. Record locking makes certain that when a program is adding, deleting, or modifying a record on a given stream, another program or stream is not allowed access to the same record or record cell. RMS-11 executing in compatibility mode does not support record locking and file sharing. There are two varieties of record locking and unlocking:

- Automatic Record Locking — The lock occurs on every execution of a $FIND or $GET macro instruction, and the lock is released when the next record is accessed, the current record is updated or deleted, the record stream is disconnected, or the file is closed. The $FREE macro instruction explicitly unlocks all records previously locked for a particular record stream. The $RELEASE macro instruction explicitly unlocks a specified record in a record stream.
- Manual Record Locking — In manual record locking, varying degrees of locking may be specified by setting bits in the record processing options field (ROP) of the RAB. The ULK bit specifies manual (as opposed to automatic) locking and unlocking. This bit specifies that locking will occur on the execution of a $GET, $FIND, or $PUT macro instruction and that unlocking may take place explicitly only via a $FREE or $RELEASE macro instruction. The NLK bit specifies that the record accessed with either a $GET or $FIND macro instruction is not to be locked, while the RLK bit specifies that a record may be accessible for read purposes but may not otherwise be accessed.

UTILITY LANGUAGES

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 System) 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 formatting 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 — reorders 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 line printer 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 single 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.

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 Hierarchies
  Use of hierarchies allows manipulation of complex data containing lists and sublists. 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 sublist) and the ANY Boolean expression, which allows DATATRIEVE to search a sublist 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 subtotals.
- 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 (uppercase/lowercase) 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 (sublist). The sublist may be of fixed or variable length.
- 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 lookup.
- 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.

VAX-11 SORT/MERGE

VAX-11 SORT/MERGE is a native mode utility that may be run interactively, as a batch job, or it can be callable from a user-written VAX-11 native mode program.

The SORT utility 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. A user can specify up to ten input files and SORT will produce one sorted output file. The sorting sequence is determined by user-specified control fields, also known as key fields, within the data themselves. 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.

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 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.

The MERGE utility permits the user to merge data from as few as two, to as many as ten similarly sorted input files. The MERGE utility merges the data according to key field(s) defined by the user and generates a single output file. The input files to be merged must be in sorted order, i.e., the SORT and MERGE key fields must be the same.

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

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

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

$ SORT/KEY=(POSITION=29,SIZE=30) SALES.DAT BILLING.LIS <cr>

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

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

$ TYPE BILLING.LIS

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

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

STORE1.FIL

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

STORE2.FIL

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

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

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

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

$ TYPE CENTR.FIL <cr>

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

VAX-11 SORT/MERGE FEATURES

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)
- merge up to ten sorted input files into one sorted output file
- create reordered address files of RFAs and keys for software use
- SORT/MERGE fixed, variable, and VFC records
- SORT/MERGE ASCII character keys in ASCII or EBCDIC sequence
- SORT/MERGE sequential, relative, indexed-sequential files
- SORT/MERGE character, decimal, binary, unsigned binary, F_, D_, G_, and H_floating data types
- SORT can determine its own work file requirements based on input file RMS information received

- SORT can be controlled by a command string or specification file
- SORT can be tuned for maximum efficiency
- SORT provides four processing techinques: record, tag, address, index
- SORT/MERGE input files from any VAX/VMS input device
- Output sorted data files to any VAX/VMS output device
- SORT automatically prints 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/MERGE error messages in VAX/VMS format
- Optional sequence checking of input files on merge

VAX-11 SORT/MERGE supports the following key formats:

- character data are ASCII
- ASCII and EBCDIC collating sequence
- binary data are VAX representation
- packed decimal data are VAX representation
- zoned decimal data are VAX representation
- unsigned binary and F_, D_, G_, and H_floating
- string decimal data format can be:
  - leading separate sign
  - leading overpunched sign
  - trailing separate sign
  - trailing overpunched sign

SORT/MERGE as a Set of Callable Subroutines

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

SORT and MERGE subroutines are callable from VAX-11 COBOL using the standard COBOL SORT and MERGE verbs.

For either interface, the user can supply a key comparison routine. This feature allows the user the flexibility of departing from the key types supported by SORT/MERGE.

With this release of VAX/VMS, full MERGE capability has entered the list of SORT features callable as a subroutine. This allows a much increased file flexibility.

File I/O Interface

The file I/O interface allows the user to specify the input files and an output file to SORT or MERGE. SORT then reads the data from the input file(s) and sorts the data into the output file. MERGE also reads the data form the input file(s) and merges it into one output file.

Record I/O Interface

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

Programming Considerations

Any program can use either SORT/MERGE subroutine interface with any of the VAX-11 native mode languages.

SORT/MERGE PERFORMANCE FEATURES

- SORT/MERGE compiles a key comparison routine specific to each sort or merge. This results in a substantial reduction of CPU usage.
- Work files are not created until they are needed. This reduces overhead when sorting small files.
- The internal record size has been reduced, therefore, less I/O is required to do intermediate merge passes for SORT.
- For some sorts, the number of intermediate merge passes has been reduced, thereby providing a substantial increase in speed of the SORT.

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 any native mode language processor.

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 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.

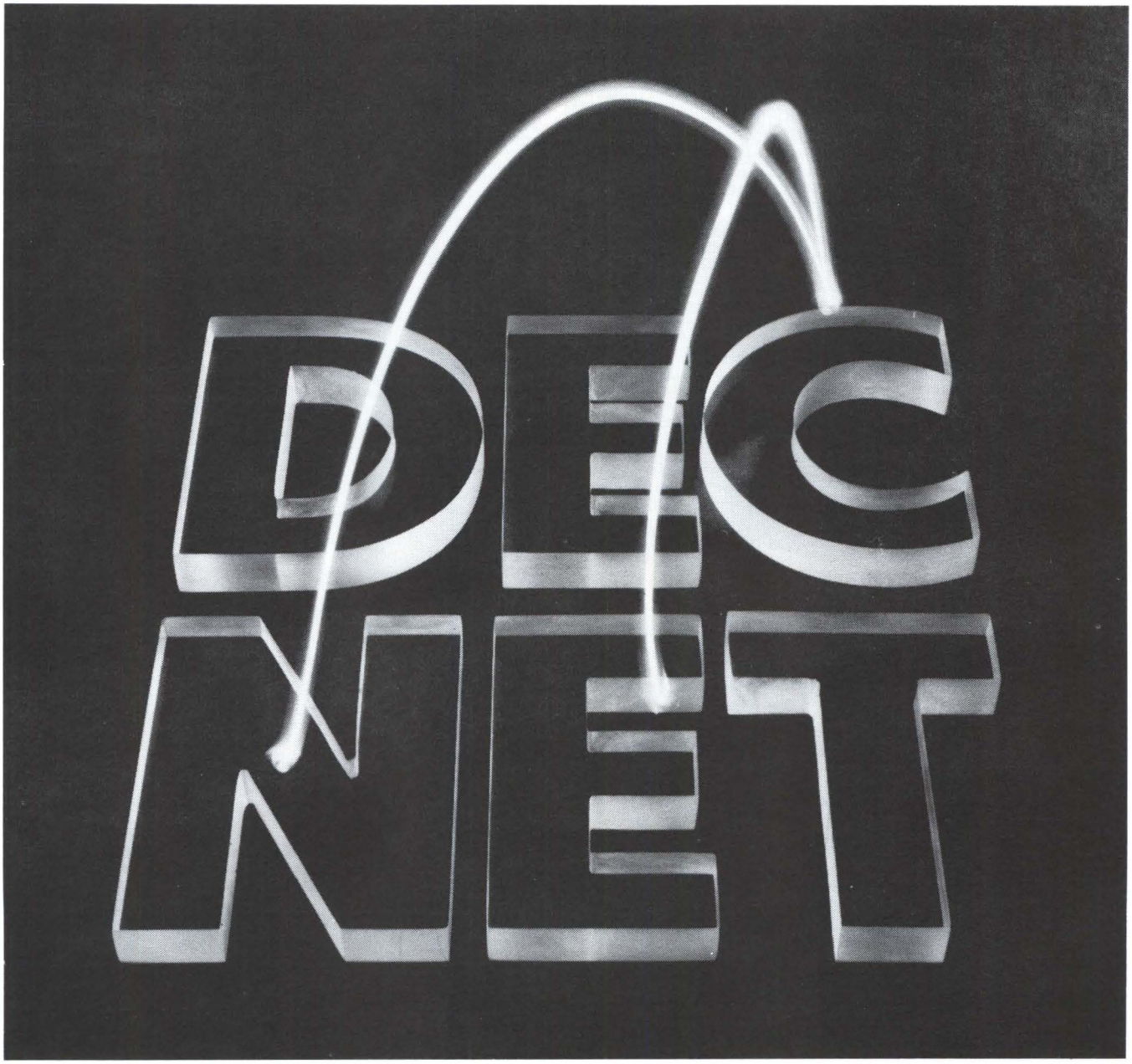
Maintaining VAX-11 FMS Applications

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.

10 Data Communications Facilities



DECnet is a family of network products developed by Digital Equipment Corporation that adds networking capability to DIGITAL's computer families and operating systems. Using DECnet, various DIGITAL computer systems can be linked together to facilitate remote communications, resource sharing, and distributed computation. DECnet is highly modular and flexible. It can be viewed as a set of tools or services from which a user selects those appropriate to build a network to satisfy the requirements of a particular application.

DIGITAL Network Architecture (DNA) provides the common network architecture upon which all DECnet products are built. The architecture is designed to handle a broad range of application requirements because all the functions of the network from the user interface to physical link control are completely modular. DNA allows nodes to operate as switches, front-ends, terminal concentrators, or hosts.

DECnet-VAX:

- provides an interprocess communication facility that is highly transparent and easy to use
- provides a higher-level language programming interface
- allows programs to access files at other systems
- allows users and programs to transfer files between systems
- allows users to transmit command files to be executed on other systems
- allows an operator to down-line load RSX-11S system images into other systems

VAX/VMS also supports protocol emulators (Internets), which enable DIGITAL systems to communicate with other vendors' systems.

INTRODUCTION

DIGITAL computers can communicate with other DIGITAL computers either remotely or locally via a network. By utilizing protocol emulators (Internets), they can communicate with computers from other suppliers.

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. A task is an image running in the context of a process.

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 10-1 illustrates such a network.

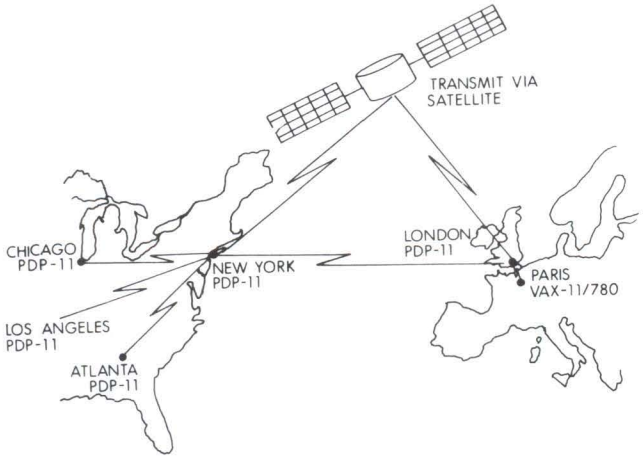


Figure 10-1
Communications Network

- 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 10-2 illustrates a resource-sharing network.

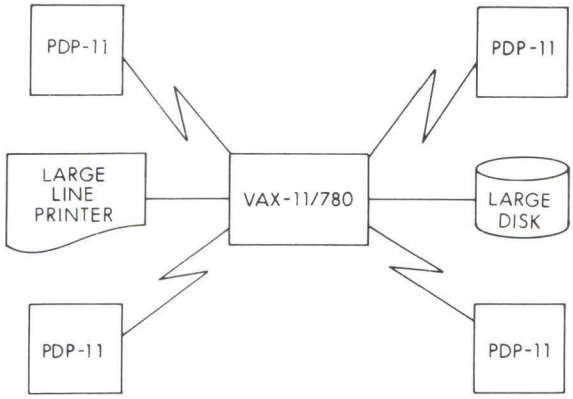


Figure 10-2
Resource-Sharing Network

- 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 10-3 and 10-4 illustrate two such networks.

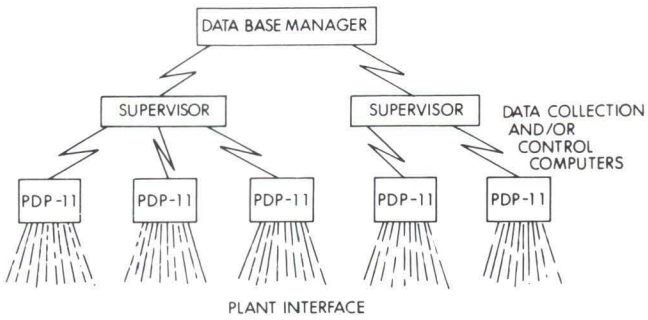


Figure 10-3
Typical Manufacturing Network

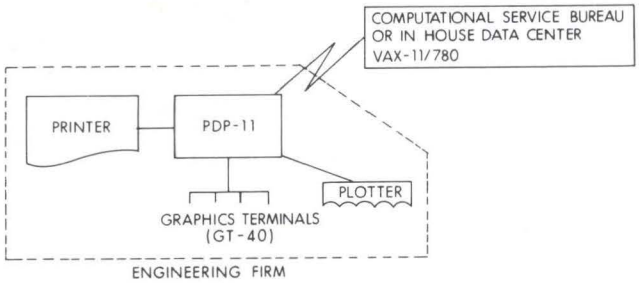


Figure 10-4

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 node's file system restrictions.
- 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 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.

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 four distinct layers: User Layer, Network Service 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 and DIGITAL supplied functions. 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 the user 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).

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, take directories of files on remote nodes, 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. Also, at the record level, RMS can be used to read, write, update, and delete records stored on a remote node.

Network Command Terminal

The network command 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 higher-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 you don't do AST processing or attempt to accept multiple connects, you 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 initiation
- RECEIVE—receive a message (the analog of GET or READ)
- SEND—transmit a message (the analog of PUT or WRITE)
- SEND INTERRUPT MESSAGE—transmit a high-priority message
- DISCONNECT—terminate a conversation (the analog of CLOSE)

The process can send optional data along with the connect request; for example, the size or number of messages that it wants to send. The receiving process or task can accept or reject the connect initiation. A process can accept multiple connect requests.

A process can send or receive mailbox messages to or from another process or task. Mailbox message traffic is essentially no different from data message traffic except that it uses a mailbox associated with the I/O channel over which the logical link was created. (This is the same mechanism used, for example, for telling programs that unsolicited terminal data are 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 access control string as follows:

nodename"access control string"::filename.filetype;version

where:

nodename =	A 1- to 6-character name (numerics or uppercase alphabetics) identifying the remote network node.
access control string =	Typically, a "username password." If omitted, default login information comes from an entry located in the local configuration data base. The double colon (::) following the nodename separates the nodename from the file specifier.
filename = filetype version	Use the following format for a DECnet-VAX node: device:[directory]filename.filetype;version

DECnet-VAX supports a subset 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 access control string, just as with any other remote file application.

Much of the VAX-11 RMS functionality is supported by DECnet-VAX, including managing sequential, relative, and indexed file organizations. A large number of the VAX-11 RMS macros are available to network users.

SAMPLE VAX-11 FORTRAN INTERTASK COMMUNICATION

This section describes the basic communication protocol involving VAX-11 FORTRAN intertask communications. 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. Similar capabilities exist in any of the native mode languages.

Three major steps in VAX-11 FORTRAN intertask communication are:

1. Creating a logical link between tasks.
2. Sending and receiving messages (each message can be 1 to 512 bytes in length).
3. Destroying 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.

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

Figure 10-5 illustrates intertask communications using normal VAX-11 FORTRAN I/O instructions.

MACRO TRANSPARENT INTERTASK COMMUNICATION

This section describes the fundamentals of coding a MACRO program for transparent intertask communications utilizing a subset of the existing macro calls available under VAX/VMS system services. These macro calls allow the user to perform intertask communications in much the same way as normal I/O operations are performed.

The term "transparent" simply implies that the calls are identical in format to all other I/O calls.

Thus, communication with another task is performed in much the same way as an I/O channel is assigned to a device ($ASSIGN). Reads and writes are then performed as if to a pair of sequential devices (that is, $QIO with the IO$_WRITEVBLK function or $OUTPUT, and $QIO with the IO$_READVBLK function or $INPUT). Finally, $DASSGN the device when communication is complete.

Three major functions in transparent intertask communication are:

1. Create a logical link between tasks.
2. Send and receive messages (each message can be 0 to 65,535 bytes long).
3. Delete the link at the end of the message dialogue.

Creating a Logical Link Between Tasks

A logical link between tasks can be created only if the tasks agree to cooperate with each other. That is, one task must request that a logical link be created, and the other task must agree to accept the request.

A logical link is requested by including a task specifier in the source task's $ASSIGN call. A task specifier identifies the remote node and the target task to which it will be connected.

The target task identifies the source task requesting the logical link connect request by specifying SYS$NET as the

Source Task Code

```
        PROGRAM DEMO2.FOR
C
C       This program prompts the user for a request, communicates
C       with a remote task to obtain the requested data, and displays
C       the answer for the user. The remote task is referenced by
C       the logical name TASK. If the remote task is named DEMO3.EXE
C       at node TULSA, the following procedure is used to run the
C       two programs:
C
C       $ DEFINE TASK TULSA::""TASK=DEMO3""
C       $ RUN DEMO2
C
        LOGICAL*1 CODE(4),BUFFER(20)
C
100     FORMAT          ('$Enter 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
C       link into it.
C
        OPEN            (UNIT=1,NAME='TASK,'ACCESS='SEQUENTIAL',FORM='FORMATTED')
C
C       Prompt the user for a request code, send the code to the
C       remote task, read the reply from the remote task, and display it to
C       the user.
C       Repeat the cycle until the user enters 'Exit' as his request code.
C
10      ACCEPT          100,CODE
        IF(CODE(1).EQ.'E') GOTO 20
        WRITE           (1,200END=20)CODE
        READ            (1,300) NCHAR,(BUFFER(K),K=1,NCHAR)
        TYPE            400,CODE,(BUFFER(K),K=1,NCHAR
        GOTO            10
C
C       Finished.
C
20      CLOSE           (UNIT=1)
        END
```

Target Task Code

```
        PROGRAM DEMO3.FOR
C
C       This is the companion task for DEMO2. For each request it
C       receives from the remote task it replies with a 1- to 20-
C       character response. This program does not know the name of the
C       requesting task. To complete the logical link with its initiator, it
C       opens the 'file' specified by the logical name SYS$NET.
C
        LOGICAL*1 CODE(4),BUFFER(20)
C
100     FORMAT          (4A1)
200     FORMAT          (20A1)
C
```

Figure 10-5

VAX-11 FORTRAN Intertask Communications

```
C       Establish a communication path with the remote task.
C
        OPEN        (UNIT=1,NAME='SYS$NET,:'ACCESS='SEQUENTIAL',FORM='FORMATTED')
C
C       Process requests until end-of-file encountered.
C
10      READ        (100,END=20)CODE
C
C       Perform appropriate processing to obtain result to
C       transmit back to the requesting task.
C
        WRITE       (1,200) (BUFFER(K),K=1,NCHAR)
        GOTO        10
C
C       Finished.
C
20      CLOSE       (UNIT=1)
        END
```

Figure 10-5 (Con't)
VAX-11 FORTRAN Intertask Communications

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 NONTRANSPARENT INTERTASK COMMUNICATION

Nontransparent intertask communication may consist of two or more tasks interacting to establish a logical link. After establishing the logical link, the tasks exchange messages over the link, then disconnect the link when communication is complete.

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 mailbox 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 op-

tional data to the remote task at the same time it disconnects or aborts a logical link.

Task Messages

There are two types of messages in nontransparent intertask communications: data messages and mailbox messages.

Data Messages — A data message is a message sent by one task, and expected by the cooperating task. That is, for each message sent by a task $QIO with the IO$_WRITEBLK function or $OUTPUT, the receiving task must issue a receive $QIO with the IO$_READVBLK function or $INPUT.

Thus, a data message in nontransparent intertask communications is the same as a data message sent in transparent communication.

Mailbox Messages — All other messages received by a task employing a nontransparent interface are classified as mailbox messages. These include any one of the following message types:

1. A logical link connect request—this message is received by the target task. It requests a logical link connection to the source task.
2. A connect accept—this message is received by the source task. The message confirms that the target task accepted the logical link connect request.
3. A connect reject—this message is also received by the source task. The message informs the source task that the target rejected the logical link connect request.
4. An interrupt message—either task can receive a 1- to 16-byte interrupt message sent by a cooperating task. The 1 to 16 bytes of data are placed in the task's mailbox.
5. A synchronous disconnect—this message informs the task that the cooperating task synchronously disconnected the logical link.
6. A disconnect abort—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 mailbox message into the mailbox associated with the channel representing the logical link to which the mailbox message applies.

In the case of a task that can accept multiple inbound connect requests, inbound connect requests are placed into the mailbox associated with the I/O channel over which the network name was declared.

Note that the mailbox was previously created using the $CREMBX system service call. The task must then explicitly retrieve the unsolicited message from the mailbox using the $QIO(IO$_READVBLK) system service call.

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, multiline basis. The product can concurrently run up to four physical lines, each with a different set of attributes (e.g., some may be 2780, 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 sup-

ports 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.

Figure 10-6 illustrates a schematic of the MUX200.

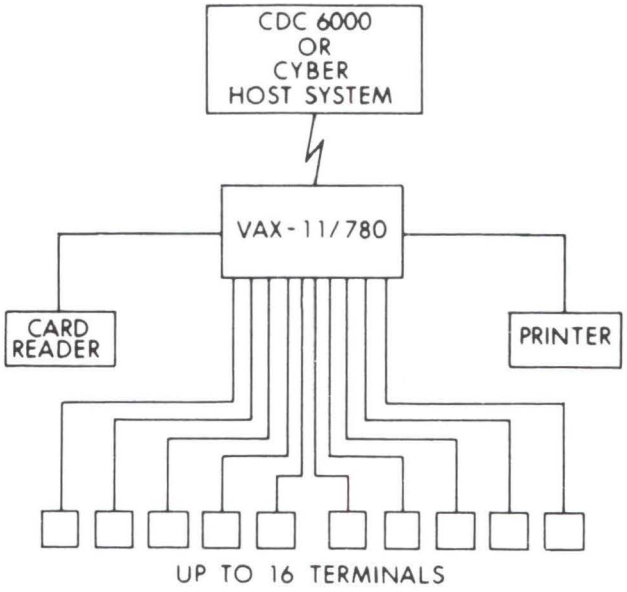


Figure 10-6
MUX200 Schematic

APPENDIX A

COMMONLY USED MNEMONICS

ACP	Ancillary Control Process
ANS	American National Standard
ASCII	American Standard Code for Information Interchange
AST	Asynchronous System Trap
ASTLVL	Asynchronous System Trap level
CCB	Channel Control Block
CM	Compatibility Mode bit in the hardware PSL
CRB	Channel Request Block
CRC	Cyclic Redundancy Check
DAP	Data Access Protocol
DDB	Device Data Block
DDCMP	DIGITAL Data Communications Message Protocol
DDT	Driver Data Table
DV	Decimal Overflow trap enable bit in the PSW
ECB	Exit Control Block
ECC	Error Correction Code
ESP	Executive Mode Stack Pointer
ESR	Exception Service Routine
F11ACP	Files-11 Ancillary Control Process
FAB	File Access Block
FCA	Fixed Control Area
FCB	File Control Block
FCS	File Control Services
FDT	Function Decision Table
FP	Frame pointer
FPD	First Part (of an instruction) Done
FU	Floating Underflow trap enable bit in the PSW
GSD	Global Section Descriptor
GST	Global Symbol Table
IDB	Interrupt Dispatch Block
IPL	Interrupt Priority Level
IRP	I/O Request Packet
ISECT	Image Section
ISD	Image Section Descriptor
ISP	Interrupt Stack Pointer
IS	Interrupt Stack bit in PSL
ISR	Interrupt Service Routine
IV	Integer Overflow trap enable bit in the PSW
KSP	Kernel Mode Stack Pointer
MBA	MASSBUS Adapter
MBZ	Must Be Zero
MCR	Monitor Console Routine
MFD	Master File Directory
MFPR	Move From Process Register instruction
MME	Memory Mapping Enable
MTPR	Move To Process Register instruction
MUTEX	Mutual Exclusion semaphore
NSP	Network Services Protocol
OPCOM	Operator Communication Manager
P0BR	Program region base register
P0LR	Program region length register
P0PT	Program region page table
P1BR	Control region base register
P1LR	Control region limit register
P1PT	Control region page table
PC	Program Counter
PCB	Process Control Block
PCBB	Process Control Block Base register
PFN	Page Frame Number
PID	Process Identification Number
PME	Performance Monitor Enable bit in PCB
PSECT	Program Section
PSL	Processor Status Longword
PSW	Processor Status Word
PTE	Page Table Entry
QIO	Queue Input/Output Request system service
RAB	Record Access Block
RFA	Record's File Address
RMS	Record Management Services
RWED	Read, Write, Execute, Delete
SBI	Synchronous Backplane Interconnect
SBR	System Base Register
SCB	System Control Block
SCBB	System Control Block Base register
SLR	System Length Register
SP	Stack Pointer
SPT	System Page Table
SSP	Supervisor Mode Stack Pointer
SVA	System virtual address
TP	Trace trap Pending bit in PSL
UBA	UNIBUS Adapter
UCB	Unit Control Block
UETP	User Environment Test Package
UFD	User File Directory
UIC	User Identification Code
USP	User Mode Stack Pointer
VCB	Volume Control Block
VPN	Virtual Page Number
WCB	Window Control Block
WCS	Writable Control Store
WDCS	Writable Diagnostic Control Store

APPENDIX B

```
IDENTIFICATION DIVISION.
PROGRAM-ID.         MERGE EXAMPLE.
*****************************************************************************
*       This program MERGEs three identically sequenced                     *
*       regional sales files into one total sales file.                     *
*       The program adds sales amounts and writes one                       *
*       record for each product-code.                                       *
*****************************************************************************
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        VAX-11.
OBJECT-COMPUTER.        VAX-11.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
            SELECT REGION1-SALES        ASSIGN TO "REG1SLS".
            SELECT REGION2-SALES        ASSIGN TO "REG2SLS".
            SELECT REGION3-SALES        ASSIGN TO "REG3SLS".
            SELECT MERGE-FILE           ASSIGN TO "MRGFILE".
            SELECT TOTAL-SALES          ASSIGN TO "TOTLSLS".
DATA DIVISION.
FILE SECTION.
FD          REGION1-SALES
            LABEL RECORDS ARE STANDARD.
01          REGION1-RECORD                          PIC X(100).
FD          REGION2-SALES
            LABEL RECORDS ARE STANDARD.
01          REGION2-RECORD                          PIC X(100).
FD          REGION3-SALES
            LABEL RECORDS ARE STANDARD.
01          REGION3-RECORD                          PIC X(100).
SD          MERGE-FILE.
            01          MERGE-REC.
                        03 M-REGION-CODE           PIC XX.
                        03 M-PRODUCT-CODE          PIC X(10).
                        03 M-SALES-AMT             PIC S9(7)V99.
                        03 FILLER                  PIC X(79).
FD          TOTAL-SALES
            LABEL RECORDS ARE STANDARD.
01          TOTAL-RECORD                            PIC X(100).
WORKING-STORAGE SECTION.
01          INITIAL-READ            PIC X           VALUE "Y".
01          THE-COUNTERS.
            03 PRODUCT-AMT                         PIC S9(7)V99.
            03 REGION1-AMT                         PIC S9(9)V99.
            03 REGION2-AMT                         PIC S9(9)V99.
            03 REGION3-AMT                         PIC S9(9)V99.
            03 TOTAL-AMT                           PIC S9(11)V99.
01          SAVE-MERGE-REC.
            03 S-REGION-CODE                       PIC XX.
            03 S-PRODUCT-CODE                      PIC X(10).
            03 S-SALES-AMT                         PIC S9(7)V99.
            03 FILLER                              PIC X(79).
PROCEDURE DIVISION.
000-START SECTION.
```

```
010-MERGE-FILES.
      OPEN OUTPUT TOTAL-SALES.
      MERGE MERGE-FILE ON ASCENDING KEY M-PRODUCT-CODE
            USING REGION1-SALES REGION2-SALES REGION3-SALES
            OUTPUT PROCEDURE IS          020-BUILD-TOTAL-SALES
                  THRU                   100-DONE-TOTAL-SALES.
      DISPLAY "TOTAL SALES FOR REGION 1 "
REGION1-AMT.
      DISPLAY "TOTAL SALES FOR REGION 2 "
REGION2-AMT.
      DISPLAY "TOTAL SALES FOR REGION 3 "
REGION3-AMT.
      DISPLAY "TOTAL ALL SALES "          TOTAL-AMT.
      CLOSE TOTAL-SALES.
      DISPLAY "END OF PROGRAM MERGE01".
      STOP RUN.
020-BUILD-TOTAL-SALES SECTION.
030-GET-MERGE-RECORDS.
      RETURN MERGE-FILE AT END
            MOVE PRODUCT-AMT TO S-SALES-AMT
            WRITE TOTAL-RECORD FROM SAVE-MERGE-REC
            GO TO 100-DONE-TOTAL SALES.
      IF INITIAL-READ = "Y"
            MOVE "N" TO INITIAL-READ
            MOVE MERGE-REC TO SAVE-MERGE-REC
            PERFORM 050-TALLY-AMOUNTS
            GO TO 030-GET-MERGE-RECORDS.
040-COMPARE-PRODUCT-CODE.
      IF M-PRODUCT-CODE = S-PRODUCT-CODE
            PERFORM 050-TALLY-AMOUNTS
            GO TO 030-GET-MERGE-RECORDS.
      MOVE PRODUCT-AMT TO S-SALES-AMT.
      MOVE ZEROES TO PRODUCT-AMT.
      WRITE TOTAL-RECORD FROM SAVE-MERGE-REC.
      MOVE MERGE-REC TO SAVE-MERGE-REC.
      GO TO 040-COMPARE-PRODUCT-CODE.
050-TALLY-AMOUNTS.
      ADD M-SALES-AMT TO PRODUCT-AMT TOTAL-AMT.
      IF M-REGION-CODE = "01"
            ADD M-SALES-AMT TO REGION1-AMT.
      IF M-REGION-CODE = "02"
            ADD M-SALES-AMT TO REGION2-AMT.
      IF M-REGION-CODE = "03"
            ADD M-SALES-AMT TO REGION3-AMT.
100-DONE-TOTAL-SALES SECTION.
120-DONE.
      EXIT.
```

APPENDIX C

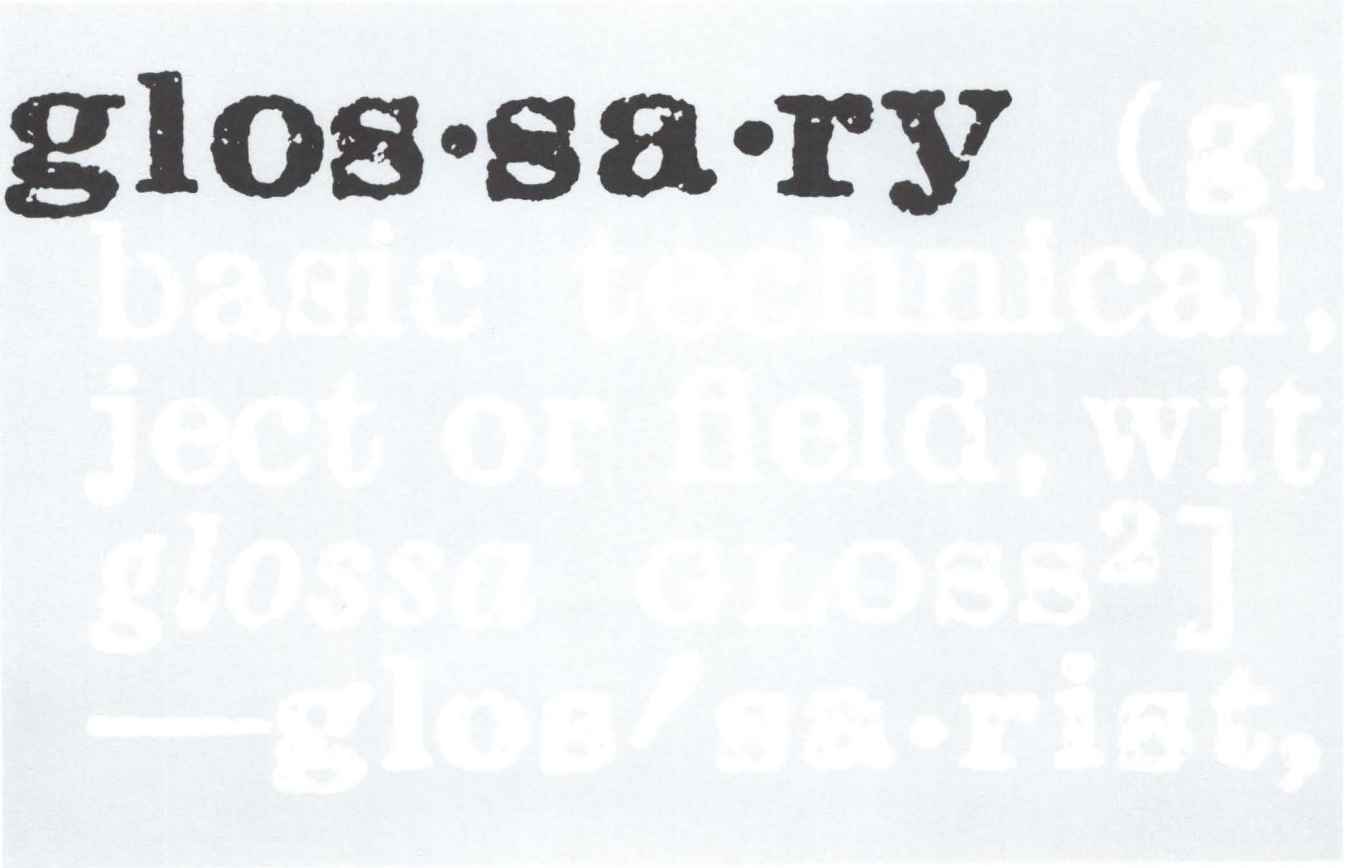
VT100 examples:

```
IDENTIFICATION DIVISION.
PROGRAM-ID.         VIDEO3.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.          VAX-11.
OBJECT-COMPUTER.          VAX-11.
SPECIAL-NAMES.
          SYMBOLIC CHARACTERS       ESCAPER    PARM-1    PARM-2    PARM-3    PARM-4

                                ARE         28         92         60        103         75.

*****************************************************************************************
*      ESCAPER = ESC      PARM-1 = [ PARM-2 = ;   PARM-3=f   PARM-4 = J              *
*****************************************************************************************
INPUT-OUTPUT SECTION.
FILE-CONTROL.
          SELECT NAME-FILE ASSIGN TO "NAMFIL".
DATA DIVISION.
FILE SECTION.
FO              NAME - FILE
                LABEL RECORDS ARE STANDARD.
01              NAME-REC.
                03      N-CUST-NUM                      PIC X(8).
                03      N-CUST-NAME                     PIC X(25)
                03      N-ADDRESS                       PIC X(25)
                03      N-CITY                          PIC X(20)
                03      N-STATE                         PIC XX.
                03      N-ZIP                           PIC X(5).
WORKING-STORAGE SECTION.
01              HOME-UP.
                03              FILLER                  PIC X       VALUE ESCAPER.
                03              FILLER                  PIC X       VALUE PARM-1.
                03              H-LINE                  PIC 99      VALUE 00.
                03              FILLER                  PIC X       VALUE PARM-2.
                03              H-COL                   PIC 99      VALUE 00.
                03              FILLER                  PIC X       VALUE PARM-3.
01              CLEAR-SCREEN.
                03              FILLER                  PIC X       VALUE ESCAPER.
                03              FILLER                  PIC X       VALUE PARM-1.
                03              FILLER                  PIC X       VALUE PARM-4.
01              THE-FORM.
                03              FORM-LINE-1.
                    04          FILLER                  PIC X       VALUE ESCAPER.
                    04          FILLER                  PIC X       VALUE PARM-1.
                    04          LINE-4                  PIC 99      VALUE 04.
                    04          FILLER                  PIC X       VALUE PARM-2.
                    04          COL-3                   PIC 99      VALUE 03.
                    04          FILLER                  PIC X       VALUE PARM-3.
                    04          FILLER                  PIC X(24)   VALUE
                                "CUSTOMER NUMBER:_______".
    03              FORM-LINE-2.
                    04          FILLER                  PIC X       VALUE ESCAPER.
                    04          FILLER                  PIC X       VALUE PARM-1.
                    04          LINE-6                  PIC 99      VALUE 06.
                    04          FILLER                  PIC X       VALUE PARM-2.
                    04          COL-3                   PIC 99      VALUE 03.
                    04          FILLER                  PIC X       VALUE PARM-3.
                    04          FILLER                  PIC X(39)   VALUE
                                "CUSTOMER NAME:".______________
```

```
    03          FORM-LINE-3.
                04      FILLER                  PIC X           VALUE ESCAPER.
                04      FILLER                  PIC X           VALUE PARM-1.
                04      LINE-8                  PIC 99          VALUE 08.
                04      FILLER                  PIC X           VALUE PARM-2.
                04      COL-3                   PIC 99          VALUE 03.
                04      FILLER                  PIC X           VALUE PARM-3.
                04      FILLER                  PIC X(42)       VALUE
                        "CUSTOMER ADDRESS:_____".
    03          FORM-LINE-4.
                04      FILLER                  PIC X           VALUE ESCAPER.
                04      FILLER                  PIC X           VALUE PARM-1.
                04      LINE-10                 PIC 99          VALUE 10.
                04      FILLER                  PIC X           VALUE PARM-2.
                04      COL-3                   PIC 99          VALUE 03.
                04      FILLER                  PIC X           VALUE PARM-3.
                04      FILLER                  PIC X(48)       VALUE
                        "CITY:_____  STATE:__  ZIP:____".
01          CURSOR-POSITIONER.
            03      FILLER                  PIC X           VALUE ESCAPER.
            03      FILLER                  PIC X           VALUE PARM-1.
            03      LINE-POS                PIC 99.
            03      FILLER                  PIC X           VALUE PARM-2.
            03      COL-POS                 PIC 99.
            03      FILLER                  PIC X           VALUE PARM-3.
01  INPUT-AREA                                             PIC X(25).
PROCEDURE DIVISION.
000-OPEN.
        OPEN OUTPUT NAME-FILE.
005-BEGIN.
        DISPLAY HOME-UP WITH NO ADVANCING.
010-CLEAR-SCREEN.
        DISPLAY CLEAR-SCREEN WITH NO ADVANCING.
020-PAINT-FORM.
        DISPLAY THE-FORM.
        GO TO 050-GET-INPUT-DATA.
050-GET-INPUT-DATA.
        MOVE SPACES TO INPUT-AREA.
*********************************************
* Position cursor to accept customer number.*
*********************************************
        MOVE 4 TO LINE-POS.
        MOVE 19 TO COL-POS.
        DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
        ACCEPT INPUT-AREA.
        IF INPUT-AREA = "DONE"
            PERFORM 005-BEGIN THRU 010-CLEAR-SCREEN
            CLOSE NAME-FILE STOP RUN.
        MOVE INPUT-AREA TO N-CUST-NUM.
        MOVE SPACES TO INPUT-AREA.
*********************************************
* Position cursor to accept customer name.*
*********************************************
        MOVE 6 TO LINE-POS.
        MOVE 17 TO COL-POS.
        DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
        ACCEPT INPUT-AREA.
        MOVE INPUT-AREA TO N-CUST-NAME.
        MOVE SPACES TO INPUT-AREA.
```

```
************************************************
* Position cursor to accept customer address.*
************************************************
           MOVE 8 TO LINE-POS.
           MOVE 20 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-ADDRESS.
           MOVE SPACES TO INPUT-AREA.
************************************
* Position cursor to accept city.    *
************************************
           MOVE 10 TO LINE-POS.
           MOVE 8 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-CITY.
           MOVE SPACES TO INPUT-AREA.
************************************
* Position cursor to accept state.   *
************************************
           MOVE 38 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-STATE.
           MOVE SPACES TO N-STATE.
************************************
* Position cursor to accept zip.     *
************************************
           MOVE 46 TO COL-POS.
           DISPLAY CURSOR-POSITIONER WITH NO ADVANCING.
           ACCEPT INPUT-AREA.
           MOVE INPUT-AREA TO N-ZIP.
           WRITE NAME-REC.
           GO TO 005-BEGIN.
```

The Glossary



abort An exception that occurs in the middle of an instruction and potentially leaves the registers and memory in an indeterminate state, such that the instruction cannot necessarily be restarted.

absolute indexed mode An indexed addressing mode in which the base operand specifier is addressed in absolute mode.

absolute mode In absolute mode addressing, the PC is used as the register in autoincrement deferred mode. The PC contains the address of the location containing the actual operand.

absolute time Time values expressing a specific date (month, day, and year) and time of day. Absolute time values are always expressed in the system as positive numbers.

access mode 1. Any of the four processor access modes in which software executes. Processor access modes are, in order from most to least privileged and protected: kernel (mode 0), executive (mode 1), supervisor (mode 2), and user (mode 3). When the processor is in kernel mode, the executing software has complete control of, and responsibility for, the system. When the processor is in any other mode, the processor is inhibited from executing privileged instructions. The Processor Status Longword contains the current access mode field. The operating system uses access modes to define protection levels for software executing in the context of a process. For example, the Executive runs in kernel and executive mode and is most protected. The command interpreter is less protected and runs in supervisor mode. The debugger runs in user mode and is no 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 also 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 UIC.

address A number used by the operating system and user software to identify a storage location. See also virtual address and physical address.

address access type The specified operand of an instruction is not directly accessed by the instruction. The address of the specified operand is the actual instruction operand. The context of the address calculation is given by the data type of the operand.

addressing mode The way in which an operand is specified; for example, the way in which the effective address of an instruction operand is calculated using the general registers. The basic general register addressing modes are called: register, register deferred, autoincrement, autoincrement deferred, autodecrement, displacement, and displacement deferred. In addition, there are six indexed addressing modes using two general registers, and literal mode addressing. The PC addressing modes are called: immediate (for register deferred mode using the PC), absolute (for autoincrement deferred mode using the PC), and branch.

address space The set of all possible addresses available to a process. Virtual address space refers to the set of all possible virtual addresses. Physical address space refers to the set of all possible physical addresses sent out on the 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).

alternate key An optional key within the data records in an indexed file; used by VAX-11 RMS to build an alternate index. See key (indexed files) and primary key.

American Standard Code for Information Interchange (ASCII) A set of 8-bit binary numbers representing the alphabet, punctuation, numerals, and other special symbols used in text representation and communications protocol.

Ancillary Control Process (ACP) A process that acts as an interface between user software and an I/O driver. An ACP provides functions supplemental to those performed in the driver, such as file and directory management. Three examples of ACPs are: the Files-11 ACP (F11ACP), the magnetic tape ACP (MTACP), and the networks ACP (NETACP).

area Areas are VAX-11 RMS maintained regions of an indexed file. They allow a user to specify placement and/or specific bucket sizes for particular portions of a file. An area consists of any number of buckets, and there may be from 1 to 255 areas in a file.

Argument Pointer General register 12 (R12). By convention, AP contains the address of the base of the argument list for procedures initiated using the CALL instructions.

assign a channel To establish the necessary software linkage between a user process and a device unit before a user process can transfer any data to or from that device. A user process requests the system to assign a channel and the system returns a channel number.

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 highest access mode for which an AST is pending. The AST does not occur until the current access mode drops in priority (raises in numeric value) to a value greater than or equal to ASTLVL. 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.

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. The scheduler never schedules a process below its base priority. The base priority can be modified only by the system manager or the process itself.

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 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 (512 contiguous bytes for most disk devices). 2. An arbitrary number of contiguous bytes used to store logically related status, control, or other processing information.

block I/O The set of VAX-11 RMS procedures that allow you direct access to the blocks of a file regardless of file organization.

bootstrap block A block in the index file on a system disk that 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, used for building and processing relative and indexed files. A bucket contains one or more records or record cells. Buckets are the unit of contiguous transfer between VAX-11 RMS buffers and the disk.

buffered I/O See system buffered I/O.

bug check The operating system's internal diagnostic check. The system logs the failure and crashes 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, significance increases from bits 0 through 7 and the value of the unsigned integer is in the range 0 to 255 decimal. A byte can be used to store one ASCII character.

cache memory A small, high-speed memory placed between slower main memory and the processor. A cache increases effective memory transfer rates and processor speed. It contains copies of data recently used by the processor, and fetches several bytes of data from memory in anticipation that the processor will access the next sequential series of bytes.

call frame See stack frame.

call instructions The processor instructions CALLG (Call Procedure with General Argument List) and CALLS (Call Procedure with Stack Argument List).

call stack The stack, and conventional stack structure, used during a procedure call. Each access mode of each process context has one call stack, and interrupt service context has one call stack.

channel A logical path connecting a user process to a physical device unit. A user process requests the operating system to assign a channel to a device so the process can transfer data to or from that device.

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.

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, syntax check, 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 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 A FORTRAN term for a program section that contains only data.

common event flag cluster A set of 32 event flags that enables cooperating processes to post event notification to each other. Common event flag clus-

ters 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 programming environment for an RSX-11M task image. The operating system compatibility mode procedures reside in the control 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.

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 previously executed instructions.

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 found, initiates the handler immediately. The condition handler may perform some action 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 context, 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' hardware context in its hardware process control block (PCB) using the Save Process Context instruction, and loads another process' hardware PCB into the hardware context using the Load Process Context instruction, scheduling that process for execution.

continuation character A hyphen at the end of a command line signifying that the command string continues on to the next command line.

console The manual control unit integrated into the central processor. The console includes an LSI-11 microprocessor and a serial line interface connected to a hard copy terminal. It enables the operator to start and stop the system, monitor system operation, and run diagnostics.

console terminal The hard copy terminal connected to the central processor console.

control region The highest-addressed half of per-process space (the P1 region). Control region virtual addresses refer to the process-related information used by the system to control the process, such as: the kernel, executive, and supervisor stacks, the permanent I/O channels, exception vectors, and dynamically used system procedures (such as the command interpreter and RSX-11M programming environment compatibility mode procedures). The user stack is also normally found in the control region, although it can be relocated elsewhere.

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 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.

D_floating (point) datum A floating point datum consisting of B contiguous bytes (64 bits) starting on an arbitrary byte boundary. The value of the D_floating datum is in the approximate range (+ or −) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{55} or typically sixteen decimal digits.

data base 1. All the occurrences of data described by a data base management system. 2. 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.

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 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. The 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 interrupt An interrupt received on interrupt priority level 16 through 23. Device interrupts can be requested only by devices, controllers, and memories.

device name The field in a file specification that identifies the device unit on which a file is stored. Device names also include the mnemonics that identify an I/O peripheral device in a data transfer request. A device name consists of a mnemonic followed by a controller identification letter (if applicable), followed by a unit number (if applicable), and ends with a colon (:).

device queue See spool queue.

device register A location in device controller logic used to request device functions (such as I/O transfers) and/or report status.

device unit One drive, and its controlling logic, of a mass storage device system. A mass storage system can have several drives connected to it.

diagnostic A program that tests logic and reports any faults it detects.

direct I/O An I/O operation in which the system locks the pages containing the associated buffer in memory for the duration of the I/O operation. The I/O transfer takes place directly from the process buffer. Contrast with 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 type 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. The directory name begins with a left bracket ([or <) and ends with a right bracket (] or >).

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.

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 record access mode to another while processing a file.

echo A terminal handling characteristic in which the characters typed by the terminal user on the keyboard are also displayed on the screen or printer.

effective address The address obtained after indirect or indexing modifications are calculated.

entry mask A word whose bits represent the registers to be saved or restored on a subroutine or procedure call using the call and return instructions.

entry point A location that can be specified as the object of a call. It contains an entry mask and exception enables known as the entry point mask.

equivalence name The string associated with a logical name in a logical name table. An equivalence name can be, for example, a device name, another logical name, or a logical name concatenated with a portion of a file specification.

error logger A system process that empties the error log buffers and writes the error messages into the error file. Errors logged by the system include memory system errors, device errors and timeouts, and interrupts with invalid vector addresses.

escape sequence An escape is a transition from the normal mode of operation to a mode outside the normal mode. An escape character is the code that indicates the transition from normal to escape mode. An escape sequence refers to the set of character combinations starting with an escape character that the terminal transmits without interpretation to the software set up to handle escape sequences.

event A change in process status or an indication of the occurrence of some activity that concerns an individual process or cooperating processes. An incident reported to the scheduler that affects a process' ability to execute. Events can be synchronous with the process' execution (a wait request), or they can be asynchronous (I/O completion). Some other events include: swapping; wake request; 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 which 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 faults, compatibility mode faults, breakpoint instruction execution, and arithmetic faults such as floating point overflow, underflow, and divide by zero.

exception condition A hardware- or software-detected event other than an interrupt or jump, branch, case, or call instruction that changes the normal flow of instruction execution.

exception dispatcher An operating system procedure that searches for a condition handler when an exception condition occurs. If no exception handler is found for the exception or condition, the image that incurred the exception is terminated.

exception enables See trap enables.

exception vector See vector.

executable image An image that is 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 provides the basic control and monitoring 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 programmed 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. 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.

extension The amount of space to allocate at the end of a file each time a sequential write exceeds the allocated length of the file.

extent The contiguous area on a disk containing a file or a portion of a file. Consists of one or more clusters.

F_floating (point) datum A floating point datum consisting of 4 contiguous bytes (32 bits) starting on an arbitrary byte boundary. The value of the F_floating datum is in the approximate range (+ or −) 0.29×10^{-38} to 1.7×10^{38}. The precision is approximately one part in 2^{23} or typically seven decimal digits.

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 and 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 An organized collection of related items (records) maintained in an accessible storage area, such as disk or tape.

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 The field preceding a file type in a file specification that contains a 1- to 9-character logical name for a file.

filename extension See file type.

file organization The physical arrangement of data in the file. You select the specific organization from those offered by VAX-11 RMS, based on your individual needs for efficient data storage and retrieval. See indexed file organization, relative file organization, and sequential file organization.

Files-11 The name of the on-disk structure used by the RSX-11, IAS and VAX/VMS operating systems. Volumes created under this structure are transportable between these 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, the file type, 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 in which the data in the file is processed.

file system A method of recording, cataloging, and accessing files on a volume.

file type The field in a file specification that is preceded by a period or dot (.) and consists of a zero- to three-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 Property of a file in which all records are of the same size. This format provides simplicity in determining the exact location of a record in the file and eliminates the need to prefix a record size field to each record.

floating (point) datum A numeric data type in which the number is represented by a fraction (less than 1 and greater than or equal to ½) multiplied by 2 raised to a power. There are four floating point data types: F_floating (4 bytes), D_floating (8 bytes), G_floating (8 bytes), and H_floating (16 bytes)

foreign volume Any volume other than a Files-11 formatted volume which may or may not be file structured.

fork process A dynamically created system process such as a process that executes device driver code or the timer process. Fork processes have minimal context. Fork processes are scheduled by the hardware rather than by the software. The timer process is dispatched directly by software interrupt. I/O driver processes are dispatched by a fork dispatcher. Fork processes 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.

fully associative cache A cache organization in which any block of data from main memory can be placed anywhere in the cache. Address comparision must take place against each block in the cache to find any particular block. Contrast with direct mapping cache.

G_floating (point) datum A floating point datum consisting of 8 contiguous bytes (64 bits) starting on an arbitrary byte boundary. The value of the G_floating datum is in the approximate range (+ or −) 0.56×10^{-308} to 9×10^{308}. The precision is approximately one part in 2^{52} or typically fifteen.

general register Any of the sixteen 32-bit registers used as the primary operands of the native mode instructions. The general registers include 12 general purpose registers which can be used as accumulators, as counters, and as pointers to locations in main memory, and the Frame Pointer (FP), Argument Pointer (AP), Stack Pointer (SP), and Program Counter (PC) registers.

generic device name A device name that identifies the type of device but not a particular unit; a device name in which the specific controller and/or unit number is omitted.

giga Metric term used to represent the number 1 followed by nine zeros.

global page table The page table containing the master page table entries for global sections.

global section A data structure (e.g., FORTRAN global common) or shareable image section potentially available to all processes in the system. Access is protected by privilege and/or group number of the UIC.

global symbol A symbol defined in a module that is potentially available for reference by another module. The linker resolves (matches references with definitions) global symbols. Contrast with local symbol.

global symbol table (GST) In a library, an index of strongly defined global symbols used to access the modules defining the global symbols. The linker will also put global symbol tables into an image. For example, the linker appends a global symbol table to executable images that are intended to run under the symbolic debugger, and it appends a global symbol table to all shareable images.

group 1. A set of users who have special access privileges to each other's directories and files within those directories (unless protected otherwise), as in the context "system, owner, group, world," where group refers to all members of a particular owner's group. 2. A set of jobs (processes and their subprocesses) who 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).

H_floating (point) datum A floating point datum consisting of 16 contiguous bytes (128 bits) starting on an arbitrary byte boundary. The value of the H_floating datum is in the approximate range (+ or −) 0.84×10^{4932} to 0.59×10^{4932}. The precision is approximately one part in 2^{112} or typically 33 decimal digits.

hardware context The values contained in the following registers while a process is executing: the Program Counter (PC); the Processor Status Longword (PSL); the 14 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 known to the processor that contains the hardware context when a process is not executing. A process' 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 prepare 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 privileges The privileges assigned to an image when it is linked. See process privileges.

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.

immediate mode In immediate mode addressing, the PC is used as the register in autoincrement mode addressing.

index The structure which allows retrieval of records in an indexed file by key value. See key (indexed files).

index file The file on a Files-11 volume that contains the access information for all files on the volume and enables the operating system to identify and access the volume.

index file bit map A table in the index file of a Files-11 volume that indicates which file headers are in use.

index register A register used to contain an address offset.

indexed addressing mode In indexed mode addressing, two registers are used to determine the actual instruction operand: an index register and a base operand specifier. The contents of the index register are used as an index (offset) into a table or array. The base operand specifier supplies the base address of the array (the base operand address or BOA). The address of the actual operand is calculated by multiplying the contents of the index register by the size (in bytes) of the actual operand and adding the result to the base operand address. The addressing modes resulting from index mode addressing are formed by adding the suffix "indexed" to the addressing mode of the base operand specifier: register deferred indexed, autoincrement indexed, autoincrement deferred indexed (or absolute indexed), autodecrement indexed, displacement indexed, and displacement deferred indexed.

indexed file organization A file organization which allows random retrieval of records by key values and sequential retrieval of records in sorted order by key value. See key (indexed files).

indirect command file See command procedure.

input stream The source of commands and data. One of: the user's terminal, the batch stream, or an indirect command file.

instruction buffer A 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 continously fetches data from memory to keep the buffer full.

interleaving Assigning consecutive physical memory addresses alternately between two memory controllers.

interlocked The property of a read followed by a write to the same datum with no possibility of an intervening reference by a second processor or I/O device. Examples are the Branch on Bit Interlocked and Add Aligned Word Interlocked instructions.

interprocess communication facility A common event flag, mailbox, or global section 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, 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 (IPL) The interrupt level at which the processor executes when an interrupt is generated. There are 31 possible interrupt priority levels. IPL 1 is lowest, 31 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 the interrupt priority level of the device's interrupt service routine.

interrupt service routine The routine executed when a device 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 with kernel privileges, 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 interrupt stack. Unlike the stack pointers for process context stacks, which are stored in the hardware PCB, the interrupt stack pointer is stored in an internal register.

interrupt vector See vector.

I/O driver See driver.

I/O function An I/O operation that is interpreted by the operating system and typically results in one or more physical I/O operations.

I/O function code A 6-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).

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.

key
indexed files: A character string, a packed decimal number, a 2- or 4-byte unsigned binary number, or a 2- or 4-byte signed integer within each data record in an indexed file. You define the length and location within the records; VAX-11 RMS uses the key to build an index. See primary key, alternate key, and random access by key value.

relative files: The relative record number of each data record in a data file; VAX-11 RMS uses the relative record numbers to identify and access data records in a relative file in random access mode. See relative record number.

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, image 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 6-bit field of the instruction. If the operand data type is byte, word, longword, quadword, or octaword, the operand is zero-extended and can express values in the range 0 through 63 (decimal). If the operand data type is F_, D_, G_, or H_floating, the 6-bit field is composed of two 3-bit fields, one for the exponent and the other for the fraction. The operand is extended to F_, D_, G_, or H_floating format.

locality See program locality.

local symbol A symbol 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.

locate mode Technique used for a record input operation in which the data records are not copied from the I/O buffer. See move mode.

locking a page in memory Making a page in an image 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 in an image ineligible for paging out of the working set for the image. The page can be swapped when the process is swapped. A page stays locked in a working set until it is specifically unlocked.

logical block number A number used to identify a block on a mass storage device. The number is 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 allow 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 I/O functions A set of I/O functions that allow restricted direct access to device level I/O operations.

logical record A group of related fields treated as a unit.

longword Four contiguous bytes (32 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 through 31. The address of the longword is the address of the byte containing bit 0. When interpreted arithmetically, a longword is a 2's complement integer with significance increasing from bit 0 to bit 30. When interpreted as a signed integer, bit 31 is the sign bit. The value of the signed integer is in the range −2,147,483,648 to 2,147,483,-647. When interpreted as an unsigned integer, significance increases from bit 0 to bit 31. The value of the unsigned integer is in the range 0 through 4,294,967,295.

macro A statement that requests a language processor to generate a predefined set of instructions.

mailbox A software data structure that is treated as a record-oriented device for general interprocess communication. Communication using a mailbox is similar to other forms of device-independent I/O. Senders perform a write to a mailbox, the receiver performs a read from that mailbox. Some system-wide mailboxes are defined: the error logger and OPCOM read from system-wide mailboxes.

main memory See physical memory.

mapping window A subset of the retrieval information for a file that is used to translate virtual block numbers to logical block numbers.

mass storage device A device capable of reading and writing data on mass storage media such as a disk pack or a magnetic tape reel.

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 *im-*

age 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 Console Routine (MCR) The command interpreter in an RSX-11 system.

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 online (an activity accomplished by a system operator).

move mode Technique used for a record transfer in which the data records are copied between the I/O buffer and your program buffer for calculations or operations on the record. See 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, and octaword integer, F_, D_, G_ and H_floating format, 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.

null process A small system process that is the lowest priority process in the system and takes one entire priority class. One function of the null process is to accumulate idle processor time.

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.

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.

octaword Sixteen contiguous bytes (128 bits) starting on an addressable byte boundary. Bits are numbered from right to left, 0 to 127. An octaword is identified by the address of the byte containing the low-order bit (bit 0).

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.

operator's console Any terminal identified as a terminal attended by a system operator.

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 four bits are required to represent the numbers 0 through 9.

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 repre-

sents 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 used as the unit of memory mapping and protection. 2. The data between the beginning of 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 are needed in physical memory. In this system, a process is paged only when it 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 least recently faulted pages as needed. A process pages 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 a 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 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 allow 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.

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 key The mandatory key within the data records of an indexed file; used by VAX-11 RMS to determine the placement of records within the file and to build the primary index. See key (indexed files) and alternate key.

primary vector A location 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.

private section An image section of a process that is not shareable among processes. See also global section.

privilege See process privilege, user privilege, and image 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 contexts 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.

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.

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 fault 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.

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, and perform direct I/O to a file-structured device.

process section See private section.

process space The lowest-addressed half of virtual address space, where per-process instructions and data reside. Process space is divided into a program region and a control region.

Program Counter (PC) General register 15 (R15). At the beginning of an instruction's execution, the PC normally contains the address of a location in memory from which the processor will fetch the next instruction it will execute.

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.

programmer number See member number.

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.

project number See group number or account number.

pure code See re-entrant code.

quadword Eight contiguous bytes (64 bits) starting on an addressable 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 1. n. A circular, doubly-linked list. See system queues. v. To make an entry in a list or table, perhaps using the INSQUE instruction. 2. See job queue.

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 key Indexed files only: Retrieval of a data record in an indexed file by either a primary or alternate key within the data record. See key (indexed files).

random access by record's file address The retrieval of a record by its unique address, which is provided to the program by RMS. This method of access is the only means of randomly accessing a sequentially organized file containing variable length records.

random access by relative record number Retrieval of a record by its relative record number. See relative record number. For relative files, random access by relative record number is synonymous with random access by key. See random access by key (relative files only).

read access type An instruction or procedure operand attribute indicating that the specified operand is only read during instruction or procedure execution.

record A set of related data that your program treats as a unit.

record access block (RAB) An RMS user data structure that represents a request for a record access stream. A 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 three methods: sequential, random, and record's file address.

record blocking The technique of grouping multiple rcords into a single block. On magnetic tape, an IRG is placed after the block rather than after each record. This technique reduces the number of I/O transfers required to read or write the data; and, in addition (for magnetic tape), increases the amount of usable storage area. Record blocking also applies to disk files.

record cell A fixed-length area in a relative file that can contain a record. The concept of fixed-length record cells lets VAX-11 RMS directly calculate the record's actual position in the file.

record format The way a record physically appears on the recording surface of the storage medium. The record format defines the method for determining record length.

record length The size of a record; that is, the number of bytes in a record.

record locking A facility that prevents access to a record by more than one record stream or process until the initiating record stream or process releases the record.

Record Management Services A set of operating system procedures that are called by programs to process files and records within files. RMS allows programs to issue READ and WRITE 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 a program can access in one I/O operation is the device's physical record.

record's file address The unique address of a record in a file, which is returned by RMS whenever a record is accessed, that allows records in disk files to be access randomly regardless of file organization. This address is valid only for the life of the file. If an indexed file is reorganized, then the RFA of each record will typically change.

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.

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 The arrangement of records in a file where each record occupies a cell of equal length within a bucket. Each cell is assigned a successive number, called a relative record number, which represents the cell's position relative to the beginning of the file.

relative record number An identification number used to specify the position of a record cell relative to the beginning of the file; used as the key during random access by key mode to relative files.

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.

return status code See status code.

RMS-11 A set of routines which is linked with compatibility mode programs, and provides similar functional capabilities to VAX-11 RMS. The file organizations and record formats used by RMS-11 are identical to those of VAX-11 RMS.

Run Time Procedure Library The collection of procedures available to native mode images at run time. These library procedures (such as trigonometric functions, etc.) are common to 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.

secondary storage Random access mass storage.

secondary vector A location that identifies the starting address of a condition handler to be executed when a condition occurs and the primary vector contains zero or the handler to which the primary vector points chooses not to handle the 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.

self-relative queue A circularly linked list whose forward and backward links use the address of the entry in which they occur as the base address for the link displacement to the linked entry. Contrast with absolute addresses used to link a queue.

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.

sequential record access mode Record storage or retrieval which starts at a designated point in the file and continues in one-after-the-other fashion through the file. That is, records are accessed in the order in which they physically appear in the file.

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 sharable image cannot be executed. A shareable image file can be used to contain a library of routines. A shareable image can be used to create 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' 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 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 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' eligibility to execute. They include: processes waiting for a common event flag, suspended processes, and executable 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' 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 adapters, and the UNIBUS adapter.

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," the system refers to the group numbers that are used by operating system and its controlling 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 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 queue A queue used and maintained by operating system procedures. See also state queues.

system region The third quarter of virtual address space. The lowest-addressed half of system space. Virtual addresses in the system region are shareable between processes. Some of the data structures mapped by system region virtual addresses are: system entry vectors, 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 processes.

system space The highest-addressed half of virtual address space. See also system region.

system virtual address A virtual address identifying a location mapped by an address 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 the keyboard and receive messages on the video screen or printer. Examples of terminals are the LA36 DECwriter hard-copy terminal and VT100 video display terminal.

time-critical process A process assigned to a software priority level between 16 and 31, inclusive. The scheduling priority assigned to a time-critical process is never modified by the scheduler, although it can be modified by the system manager or process itself.

timer A system fork process that maintains the time of day and the date. It also 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).

translation buffer An internal processor cache containing translations 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 the FP register contents stored on the stack for each call frame.

urgent interrupt An interrupt received on interrupt priority levels 24 through 31. These can be generated only by the processor for the interval clock, serious errors, and power fail.

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 that 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. It consists of a group number and a member number separated by a comma.

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.).

value return registers The general registers R0 and R1 used by convention to return function values. These registers are not preserved by any called procedures. They are available as temporary registers to any called procedure. All other registers (R2, R3,...,R11, AP, FP, SP, PC) are preserved across procedure calls.

variable-length bit field A set of 0 to 32 contiguous bits located arbitrarily with respect to byte boundaries. A variable bit field is specified by four attributes: 1) the address A of a byte, 2) the bit position P of the starting location of the bit field with respect to bit 0 of the byte at address A, 3) the size, in bits, of the bit field, and 4) whether the field is signed or unsigned.

variable-length record format A file format in which records are not necessarily the same length.

variable with fixed-length control record format Property of a file in which records of variable-length contain an additional fixed control area capable of storing data that may have no bearing on the other contents of the record. Variable with fixed-length control record format is not applicable to indexed files.

VAX-11 Record Management Services (VAX-11 RMS) The file and record access subsystem of the VAX/VMS operating system for VAX. VAX-11 RMS helps your application program process records within files, thereby allowing interaction between your application program and its data.

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 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 (.) and is followed by a number which generally identifies it as the latest file created of all files having the identical file specification but for version number. 2. The number used to identify the revision level of program.

virtual address A 32-bit integer identifying a byte "location" in virtual address space. The memory management hardware translates a virtual address to a physical address. The term "virtual address" may also refer to the address used to identify a virtual block on a mass storage device.

virtual address space The set of all possible virtual addresses that an image executing in the context of a process can use to identify the location of an instruction or data. The virtual address space seen by the programmer is a linear array of 4,294,967,296 (2^{32}) byte addresses.

virtual block A block on a mass storage device referred to by its file-relative address rather than its logical (volume-oriented) or physical (device-oriented) address. The first block in a file is always virtual block 1.

virtual I/O functions A set of I/O functions that must be interpreted by an ancillary control process.

virtual memory The set of storage locations in physical memory and on disk that are referred to by virtual addresses. From the programmer's viewpoint, the secondary storage locations appear to be locations in physical memory. The size of virtual memory in any system depends on the amount of physical memory available and the amount of disk storage used for non-resident virtual memory.

virtual page number The virtual address of a page of virtual memory.

volume
Disks: An ordered set of 512-byte blocks. The basic medium that carries a Files-11 structure.

Magnetic tape: A reel of magnetic tape, which may contain a part of a file, a complete file, or more than one file.

volume set A collection of related volumes.

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 by the timer process, if the hibernating process or another process scheduled a 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 of the unsigned integer is in the range 0 through 65,535.

working set The set of pages in process 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 are 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 are copied in both cache and main memory. Cache and main memory data are always consistent. Contrast with write back.

Index

Absolute mode addressing 4-7
Access modes 2-1, 4-17, 4-18
Accounting statistics 3-8
ACP (ancillary control process) 6-20, 6-21
Address
  manipulation instructions 4-10, 4-13
  mapping registers 4-17
  physical 4-17 to 4-22
Address
  virtual 4-17 to 4-22
Addressing modes 4-1, 4-5 to 4-7
Address sort 9-14
Address space
  physical 4-18 to 4-20
  virtual 4-1, 4-4, 4-5, 4-18 to 4-20, 6-5
Address translation buffer 4-29
Ancillary control process (ACP) 6-20, 6-21
Application programming 3-1 to 3-5, 3-10 to 3-14
Arbitration
  Memory Interconnect 4-28, 4-29
Argument Pointer (AP) 4-5, 4-7, 4-8
Arguments
  definitions 4-7
  passing 4-4, 4-7, 4-8
Arithmetic exceptions 4-8
Assembler (see also VAX-11, MACRO; (instruction set))
  MACRO-11 (PDP-11) 7-36
  VAX-11 MACRO 3-4, 7-33, 7-34
Asynchronous system trap processing services 6-8, 6-11
Autodecrement addressing mode 4-5, 4-6
Autoincrement addresssing mode 4-5, 4-6
Autoincrement Deferred addressing mode 4-5, 4-6
Automatic recovery
  power failure 3-9
Automatic restart 2-4
Backup disks 2-4
Backup files 9-4
Bad block locator 9-4
Bad blocks 2-3, 5-1, 5-2
Balance Set 6-19
Bandwidth
  memory 4-30
Base priority 6-18, 6-19
BASIC
  PDP-11 2-1, 2-2, 7-1, 7-34, 7-35
  VAX-11 1-1,2-1, 2-2, 7-1, 7-14 to 7-21
Batch processing 2-1, 2-5, 3-3, 3-6, 3-9, 8-1, 8-3, 8-4
Battery backup 2-4, 4-29
Bit field 4-4
BLISS-32 1-1, 2-1, 2-2, 3-4, 3-5, 6-11, 7-29 to 7-32
Block I/O 5-1, 5-2, 6-10, 9-10, 9-11
Blocks
  bad 2-3, 5-1, 5-2
Bootstrap 2-3, 2-4
Branch instructions 4-10, 4-13 to 4-15
Buffers 4-29, 4-32
BUS
  MASSBUS 2-2, 4-29, 4-30, 4-33, 5-1, 5-2
  Memory Interconnect 2-1, 2-2, 4-28 to 4-29
  UNIBUS 2-2, 4-27, 4-30, 4-33, 5-1, 5-2, 5-5
Byte 4-1 to 4-4
Cache memory 2-1, 4-29, 4-32
CALL
  facility 7-1, 7-5, 7-11, 7-18, 7-23
  instructions 4-10, 4-15
Call frame 4-8
Card reader 5-3
Case instruction 4-10, 4-13, 4-14
Character data 4-2, 4-4
Character string instructions 4-9, 4-12
Clocks 2-1
COBOL 1-1, 2-1, 2-2, 3-4, 7-1, 7-7 to 7-14, 9-9
Command language 2-2, 2-5, 3-1 to 3-4
Command procedures 2-2, 2-5, 3-1 3-3, 8-1, 8-8, 8-9
Command terminal 5-3, 5-4
Commercial system
  example 3-10, 3-11, 3-13
Communication
  between processes 2-2, 3-6, 3-7, 6-12 to 6-14
  interprocessor 5-6
  network 10-1
Compatibility mode 2-5, 3-4, 3-5, 4-1, 4-15, 4-16, 6-22, 6-23, 7-1, 7-34 to 7-36
Condition codes 4-4, 4-8
Condition handlers 4-8, 6-11, 6-12
Connect-to-interrupt 6-11
Console 2-1, 4-25, 4-28, 4-31, 4-32, 5-6, 5-7
Context 4-1, 6-2, 6-3
Context switching 4-16
Control region 4-5, 6-5, 6-6
CR11 card reader 5-3
Data Communications Facilities 10-1 to 10-8
Data
  integrity 2-3
  managment facilities 9-1 to 9-17
  protection 6-4, 6-5
  throughput 4-30, 5-5, 5-6
  types
    VAX-11 4-1 to 4-4
    VAX-11 BASIC 7-15
    VAX-11 COBOL 7-9
    VAX-11 FORTRAN 7-2
DATATRIEVE 9-12 to 9-14
DDCMP (DIGITAL Data Communications Message Protocol) 5-6, 10-2
Debugger 2-2, 3-6, 7-1, 8-1, 8-6, 8-7
Debugger
  BLISS-32 7-31
  COBOL 7-11
  FORTRAN 7-7
Decimal data 4-2, 4-3
DECnet 10-1 to 10-7
Default device name 3-7
Definitions G-1 to G-21
Detached process 6-3
Device
  drivers 3-13, 4-23, 4-24, 5-1, 6-2, 6-11, 6-22, 9-1
  independence 3-6, 6-20
  name 6-11
Diagnostics
  on-line 3-9, 3-10

peripherals 2-3, 2-4
remote 2-4, 3-10
DIGITAL Data Communications Message Protocol (DDCMP) 5-6, 10-2
DIGITAL Network Architecture (DNA) 10-2
Direct memory access (DMA) devices 5-1 to 5-3, 5-5
Directory 3-7, 9-1, 9-2
Disks
backup 2-4, 9-4
supported 5-1, 5-2
Disk Save and Compress (DSC) utility 9-4
Displacement addressing mode 4-5, 4-6
Displacement Deferred addressing mode 4-6
Distributed computing network 10-1, 10-2
DMA (direct memory access) devices 5-1 to 5-3, 5-5
DMC11 communications link 5-6
DNA (DIGITAL Network Architecture) 10-2
Down-line loading 2-5, 3-10, 10-2
DR11-B direct memory access digital interface 5-5
DR780 (High Performance Interface) 5-5, 5-6
Drivers
device 3-13, 4-23, 4-24, 5-1, 6-2, 6-11, 6-22, 9-1
DSC (Disk Save and Compress) utility 9-4
Dynamic access 9-7
DZ11 terminal line interface 5-5
Edit instruction 4-9, 4-11, 4-12
Editors 2-1, 8-1 to 8-4
Error
logging 2-4, 3-9
read 5-2
Error Correcting Code (ECC) MOS memory 2-1, 2-3, 4-27, 4-32, 5-2
Event 6-18
Event flag
clusters 6-11, 6-13
common 6-13
local 6-11
system services 6-8
Exception condition handling services 6-8, 6-11, 6-12
Exceptions 4-4, 4-8, 4-16, 6-11, 6-12
Eception vectors 4-22
Fault detection 2-3
Files
backup 9-4
comparing 9-4
directories 9-1
logical names 9-3, 9-4
management 2-2, 9-1 to 9-4
manipulation
DECnet-VAX 10-2 to 10-4
PDP-11 BASIC-PLUS-2/VAX 7-35
protection 3-6, 3-7
RMS-11 2-5, 6-22, 6-23
sorting 9-4, 9-14 to 9-16
specificaions 9-1, 9-2
VAX-11 BASIC 7-17
VAX-11 COBOL 7-9, 7-10
VAX-11 FORTRAN 7-2
VAX-11 RMs 2-2, 3-5, 6-20, 7-1, 7-2, 9-1 to 9-12
Files-11 On-Disk Structure Level 2 (ODS-2) 2-2
Flight simulation
example 3-10, 3-12, 3-13
Floating point
accelerator 2-3, 4-30
data 4-2, 4-3
instructions 4-9, 4-11
Foreign volume 6-10
FORTRAN
PDP-11 FORTRAN IV/VAX to RSX 2-1, 2-2, 3-4, 3-5, 7-1, 7-36
VAX-11 FORTRAN 3-4, 7-1, 7-2 to 7-7
Frame Pointer (FP) 4-5, 4-7, 4-8
General register manipulation instructions 4-10, 4-13
General registers 4-1, 4-5
Global sections 6-14
Glossary G-1 to G-21
Hard copy terminal 5-4
Hardware
context 4-16, 4-25
process control block 4-23
High performance interface (DR780) 5-5, 5-6
Host development mode 7-1
Image 4-1, 6-2, 6-14
Immediate mode addressing 4-7
Indexed addressing mode 4-6
Indexed files 7-2, 7-9, 7-10, 9-5 to 9-10
Index instruction 4-10, 4-12
Index register 4-6
Index sort 9-14
Instruction buffer 4-29, 4-32
Instruction set
compatibility mode 4-1, 4-15, 4-16
native mode 4-1, 4-8 to 4-15
Integer instructions 4-9, 4-11
Interactive terminals 5-3 to 5-5
Interactive text editor 8-1 to 8-3
Interleaving 4-29
Internets (protocol emulators) 10-7, 10-8
Interprocess communication 2-2, 3-6, 3-7, 6-12 to 6-14
Interprocessor communications link 5-5, 5-6, 6-14
Interrupts 4-16, 4-23
Interrupt vectors 4-22
I/O
controller interfaces 4-29 to 4-33
device drivers 3-13, 4-23, 4-24, 5-1, 6-2, 6-11, 6-22, 9-1
processing 6-19 to 6-22
RMS 9-9 to 9-11
space 4-23
system services 6-7, 6-10 to 6-12, 6-20
Jump instruction 4-10, 4-13 to 4-15
Key
indexed files 9-8, 9-9
Known image 6-4
LA11 line printer 5-3
LA120 hard copy terminal 5-4
LA36 hard copy terminal 5-4
Languages (see also names of specific languages) 1-1, 2-1, 2-2, 3-4, 3-5, 7-1 to 7-36
Librarian 2-1, 8-1, 8-7
Libraries 3-6, 7-5, 7-11, 7-31, 7-34
Line printers 5-3
Linker 2-1, 8-1, 8-4, 8-5
Literal mode addressing 4-6
Local event flag 6-11
Local event flag clusters 6-11
Logical names 6-11, 9-3, 9-4
Longword 4-2, 4-3
Loop control instructions 4-10, 4-13, 4-14
LPA11-K direct memory access controller 5-5
LP11 line printers 5-3
MACRO assembler 2-1, 2-2, 3-4, 7-1, 7-33, 7-34
Macros
BLISS-32 7-31
MACRO 7-34
Magnetic tape 5-2, 5-3
Mailbox 2-2, 3-5, 3-6, 3-11, 6-10, 6-11, 6-13, 6-14
Main memory (see also Memory) 2-1 to 2-4, 4-27 to 4-29, 4-32
Manager
system 3-6 to 3-8
Map-to-I/O page 6-11
Mapping 4-18 to 4-22, 6-14, 6-15
Mapping registers 4-17
MASSBUS 2-2, 4-29, 4-30, 4-33, 5-1, 5-2

Mass storage devices 2-2, 5-1 to 5-3, 6-10

Mathematics library 2-3, 8-6

Memory
  bandwidth 4-30
  battery backup 2-4, 4-29
  interleaving 4-29
  main 2-1 to 2-4, 4-29, 4-32
  management 2-1, 4-18 to 4-22, 6-2, 6-14 to 6-18
  Management control system services 6-9, 6-10, 6-17, 6-18
  mapping 4-18 to 4-22, 6-14, 6-15
  multiported 5-6, 6-14
  physical 2-1 to 2-4, 4-29, 4-32
  protection 2-1, 4-17, 4-18
  shared areas 5-6, 6-14
  virtual (see also Virtual addressing; Virtual memory) 4-17

Modified pages 6-16, 6-17

MOS (Main) memory 2-1 to 2-4, 4-29, 4-32

Multiprogramming 2-1, 3-5, 3-6, 4-16

Native mode
  instruction set 4-1, 4-8 to 4-15
  programming environment 7-1 to 7-36, 8-1 to 8-10

Network Ancillary Control Process (NETACP) 10-3

Network services 2-1, 2-2, 2-5, 10-1 to 10-8

Network Services Protocol (NSP) 10-2

Non-processor request (NPR) devices 5-1 to 5-3

Non-transparent interprocess communication 10-3
  macro 10-6

Octaword 4-2, 4-3

On-line diagnostics 3-9, 3-10

Operating system
  compatibility mode 6-22, 6-23
  interprocess communication and control 6-12 to 6-14
  I/O processing 6-19 to 6-22
  memory management 6-2, 6-14 to 6-18
  overview 2-1 to 2-5, 6-1 to 6-5
  process scheduling 6-2, 6-18, 6-19
  system services 6-5 to 6-14
  user environment 6-5 to 6-14

Operator
  system 3-8 to 3-10

Optimizations
  VAX-11 FORTRAN 7-5 to 7-7

Owner process 6-3

Packed decimal
  data 4-2
  instructions 4-9, 4-11

Page
  description 2-1, 6-15 to 6-17
  fault 6-16
  mapping 4-20 to 4-22
  tables 4-20, 4-22, 6-14

Paging 2-1, 3-5, 3-6, 6-15 to 6-17

PDP-11
  BASIC-PLUS-2/VAX 2-1, 2-2, 3-4, 3-5, 7-1, 7-34, 7-35
  Compatibility 2-5, 3-4, 3-5, 4-1, 4-15, 4-16, 6-22, 6-23, 7-1 7-34 to 7-36
  DATATRIEVE 9-12 to 9-14
  FORTRAN IV/VAX to RSX 2-1, 2-2, 3-4, 3-5, 7-1, 7-36
  instructions 4-1, 4-16
  MACRO-11 2-1, 2-2, 3-4, 3-5, 7-1, 7-36

Performance 2-3

Performance analysis statistics 3-8

Peripheral devices 2-2, 5-1 to 5-7

Per-process space 4-4, 4-5, 6-6

Physical address 4-18 to 4-20

Physical memory 2-1 to 2-4, 4-29, 4-32

Position independent code 4-7

Power failure 3-9

Priority 2-1, 2-2, 2-5, 4-16, 4-17, 4-22, 6-18, 6-19

Private Pages 2-3

Privilege 3-7, 6-4

Privileged instructions 4-10, 4-18

Procedure
  control instructions 4-10, 4-15
  definition 4-4

Process
  communication 2-2, 3-6, 6-12 to 6-14, 10-3, 10-4
  control blocks 4-23 to 4-25
  control system services 6-8, 6-9
  context 4-1
  description 3-5, 4-1, 4-16, 6-2 to 6-4
  mapping 6-14, 6-15
  page table 4-20, 4-22, 4-23
  scheduling 6-18, 6-19
  virtual address space 4-1, 4-4, 4-5
  virtual memory 6-15

Processor
  access modes 4-17, 4-18
  description 2-1, 4-1 to 4-33

Process-oriented paging (see also Paging) 2-1

Processor Status Longword (PSL) 4-17

Processor Status Word 4-8

Process control system services 6-8, 6-9, 6-12 to 6-14

Program Counter (PC) 4-1, 4-6, 4-7

Program
  application 3-10 to 3-13
  development tools 2-2, 3-6, 8-1 to 8-10
  region 4-5, 6-5, 6-6

Programmable real-time clock 2-1

Programmed interrupt request devices 5-1

Programming
  interfaces 2-5
  languages 1-1, 2-1, 2-2, 3-4, 3-5, 7-1 to 7-36

Protection 2-1, 2-3, 4-17, 4-18

Protection code 3-6

Protocol Emulators (Internets) 10-7, 10-8

PSL (Processor Status Longword) 4-17

Quadword 4-2, 4-3

Queue control 3-8, 3-9

Queue instructions 4-10, 4-13

Queue I/O Request processing 6-21, 6-22

Quota 3-7, 3-8

Random record access mode 9-7

Read error 5-2

Real-time clock 2-1

Real-time flight simulation example 3-10, 3-12, 3-13

Real-time Interface Extensions
  connect-to-interrupt 6-11
  map-to-I/O page 6-11

Real-time processes
  memory management 3-6
  priority 2-1, 2-2, 2-5, 4-16, 4-17
  resource allocation 3-7, 3-8, 6-1, 6-18, 6-19

Record
  access modes
    RMS 9-6, 9-7
  attributes 9-7 to 9-9
  processing
    RMS 9-9 to 9-12

Record I/O 9-9, 9-10

Record Management Services (RMS) 2-2, 2-5, 3-5, 6-20, 7-1, 7-2, 7-9, 7-17, 7-23, 9-1 to 9-12

Record's File Address (RFA access mode) 9-7, 9-9

Record sort 9-14

Regions 4-5

Register addressing mode 4-5

Register Deferred addressing mode 4-5, 4-6

Register Deferred Indexed Addressing mode 4-6

Register manipulation instructions 4-10, 4-13

Registers 4-1, 4-5

Relative Files 9-5 to 9-7, 9-9, 9-10

Reliability features 2-3, 2-4

Remote diagnostics 2-4, 3-10

Resource
allocation 6-4
quota 3-7, 3-8
Resource-sharing network 10-1
Restart 2-4
RFA (Record's File Address) access mode 9-7, 9-9
RK06 disk drive 5-2
RK07 disk drive 5-1, 5-2
RL02 disk drive 5-1, 5-2
RM03 disk drive 5-1, 5-2
RM05 disk drive 5-1, 5-2
RMS-11 2-5, 6-22, 6-23
RMS (Record Management Services) 2-2, 2-5, 3-5, 6-20, 7-1, 7-2, 7-9, 7-17, 7-23, 9-1 to 9-12
RP06 disk drive 5-1, 5-2
RSX-11M (see also PDP-11, Compatibility) 6-22, 6-23
RX01 Floppy disk 5-6, 5-7
RX02 disk drive 5-1, 5-2
Scatter/gather transfers 4-29
Scheduling 2-1, 2-2, 2-5, 3-5, 3-6, 6-2, 6-18, 6-19
Sequential files 9-5 to 9-7
Sequential record access mode 9-6, 9-7, 9-9
Serial line multiplexer 5-5
Sharable image 8-5
SLP text editor 8-1, 8-3, 8-4
Software process control block 4-23
Sort/MERGE 9-14 to 9-16
SOS interactive text editor 8-1, 8-2
Special function instructions 4-10, 4-15
Spooling 2-3, 3-8, 3-9
Stack 4-4
Stack frame 4-4, 4-8
Stack Pointer (SP) 4-5, 4-7, 4-8
String handling
BASIC-PLUS-2/VAX 7-35
Subprocess 6-3
Subroutine control instructions 4-10, 4-14, 4-15
Swapping 6-19
Symbolic debugger 2-2, 3-6, 7-1, 8-1, 8-6, 8-7
Symbolic Traceback Facility 7-1, 7-7, 7-11, 7-12
System
automatic recovery
power failure 2-4, 3-9
event 6-18
manager 3-6 to 3-8
operator 3-8 to 3-10
page table 4-20, 4-21
programming 4-17 to 4-26
region 4-5, 6-5, 6-6
services 6-5 to 6-14
space 4-5
System Control Block 4-22, 4-24
Tag sort 9-14
TE16 tape storage system 5-2, 5-3
Terminals 5-3 to 5-5
Terms
definitions G-1 to G-21
Text editors 8-1
Interactive
EDT 8-1 to 8-3
SOS 8-1,8-2
Batch
SLP 8-1, 8-3, 8-4
Time-of-year clock 2-1
Traceback 2-2, 3-6, 7-1, 8-1, 8-6, 8-7
Trace faults 4-8
Transparent interprocess communication
10-3
macro 10-4
Traps 4-8
TS11 tape storage subsystem 5-2
TU45 tape storage system 5-2
TU58 tape storage system 5-7
TU77 tape storage system 5-2
Type-ahead 5-3
UETP (User Environment Test Package) 3-10
UIC (user identification code) 3-6 to 3-8, 6-4,6-5
UNIBUS 2-2, 4-27, 4-30, 4-31, 4-33
Unit record peripherals 5-3
User authorization 3-10
User authorization file 3-10, 6-4, 6-5
User Environment Test Package (UETP) 3-10
User identification code (UIC) 3-6 to 3-8, 6-4, 6-5
Variable-length bit field instructions 4-9, 4-12, 4-13
VAX-11
750 Processor 4-31 to 4-33
780 Processor 4-27 to 4-30
BASIC 1-1, 2-1, 2-2, 3-4, 7-1, 7-14 to 7-21
BLISS-32 1-1, 2-1, 2-2, 3-4, 3-5, 7-1, 7-29 to 7-32
COBOL 1-1, 2-1, 2-2, 3-4, 7-1, 7-7 to 7-14
Common language environment 7-1, 7-2
CORAL 66 2-1, 2-2, 3-4, 3-5, 7-1, 7-32, 7-33
data types 4-1 to 4-4
Forms Management System (FMS) 9-16, 9-17
FORTRAN 1-1, 2-1, 2-2, 3-4, 7-1, 7-2 to 7-7
Interactive Symbolic debugger 2-2, 3-6, 7-1, 8-1, 8-6, 8-7
MACRO 2-1, 2-2, 3-4, 7-1, 7-33, 7-34
PASCAL 1-1, 2-1, 2-2, 3-4, 3-5, 7-1, 7-26 to 7-29
PL/I 1-1, 2-1, 2-2, 3-4, 7-1, 7-21 to 7-26
Procedure Calling Standard 2-5, 7-1, 7-23
Processor architecture 4-1 to 4-26
Record Management System (RMS) 2-2, 2-5, 3-5, 6-20, 7-1, 7-2, 7-9, 7-17, 7-23, 9-1 to 9-12
RUNOFF 8-10
SORT/MERGE 9-14 to 9-16
VAX-11/750 4-31 to 4-33
VAX-11/780 4-25 to 4-28
VAX/VMS
command language 2-2, 2-5, 3-1 to 3-4
DEBUG program 2-2, 3-6, 7-1, 8-1, 8-6, 8-7
operating system (see also Operating system) 2-1 to 2-5, 6-1 to 6-23
Vectors 4-22, 4-24
Video terminal 5-4, 5-5
Virtual addressing 4-1, 4-4, 4-5, 4-17 to 4-22, 6-5
Virtual memory
operating system 2-1 to 2-5, 6-1 to 6-23
process 2-1 to 2-3, 6-2, 6-3, 6-12
programming considerations 6-17 to 6-19
VT100 video terminal 5-4, 5-5
Word 4-2, 4-3
Working set 6-15, 6-16

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, MA 01754, Tel. (617) 897-5111 — SALES AND SERVICE OFFICES; UNITED STATES — ALABAMA, Birmingham, Huntsville ARIZONA, Phoenix, Tucson ARKANSAS, Little Rock CALIFORNIA, Costa Mesa, El Segundo, Los Angeles, Oakland, Sacramento, San Diego, San Francisco, Monrovia, Santa Barbara, Santa Clara, Sherman Oaks COLORADO, Colorado Springs, Denver CONNECTICUT, Fairfield, Meriden DELAWARE, Newark FLORIDA, Miami, Orlando, Pensacola, Tampa GEORGIA, Atlanta HAWAII, Honolulu IDAHO, Boise ILLINOIS, Chicago, Peoria INDIANA, Indianapolis IOWA, Bettendorf KENTUCKY, Louisville LOUISIANA, New Orleans MARYLAND, Baltimore MASSACHUSETTS, Boston, Springfield, Waltham MICHIGAN, Detroit, Kalamazoo MINNESOTA, Minneapolis MISSOURI, Kansas City, St. Louis NEBRASKA, Omaha NEW HAMPSHIRE, Manchester NEW JERSEY, Cherry Hill, Parsippany, Princeton, Somerset NEW MEXICO, Albuquerque, Los Alamos NEW YORK, Albany, Buffalo, Long Island, New York City, Rochester, Syracuse, Westchester NORTH CAROLINA, Chapel Hill, Charlotte OHIO, Cincinnati, Cleveland, Columbus, Dayton OKLAHOMA, Tulsa OREGON, Portland PENNSYLVANIA, Harrisburg, Philadelphia, Pittsburgh RHODE ISLAND, Providence SOUTH CAROLINA, Columbia, Greenville TENNESSEE, Knoxville, Nashville TEXAS, Austin, Dallas, El Paso, Houston, San Antonio UTAH, Salt Lake City VERMONT, Burlington VIRGINIA, Fairfax, Richmond WASHINGTON, Seattle, Spokane WASHINGTON D.C. WEST VIRGINIA, Charleston WISCONSIN, Milwaukee INTERNATIONAL — EUROPEAN AREA HEADQUARTERS: Geneva, Tel: [41] (22)-93-33-11 INTERNATIONAL AREA HEADQUARTERS: Acton, MA 01754, U.S.A., Tel: (617) 263-6000 AUSTRALIA, Adelaide, Brisbane, Canberra, Hobart, Melbourne, Perth, Sydney, Townsville AUSTRIA, Vienna BELGIUM, Brussels BRAZIL, Rio de Janeiro, Sao Paulo CANADA, Calgary, Edmonton, Halifax, Kingston, London, Montreal, Ottawa, Quebec City, Regina, Toronto, Vancouver, Victoria, Winnipeg DENMARK, Copenhagen ENGLAND, Basingstoke, Birmingham, Bristol, Ealing, Epsom, Leeds, Leicester, London, Manchester, Reading, Welwyn FINLAND, Helsinki FRANCE, Bordeaux, Lyon, Paris, Puteaux, Strasbourg HOLLAND, Amstelveen, Delft, Utrecht HONG KONG IRELAND, Dublin ISRAEL, Tel Aviv ITALY, Milan, Rome, Turin JAPAN, Osaka, Tokyo MEXICO, Mexico City, Monterrey NEW ZEALAND, Auckland, Christchurch, Wellington NORTHERN IRELAND, Belfast NORWAY, Oslo, PUERTO RICO, San Juan SCOTLAND, Edinburgh REPUBLIC OF SINGAPORE SPAIN, Barcelona, Madrid SWEDEN, Gothenburg, Stockholm SWITZERLAND, Geneva, Zurich, WEST GERMANY, Berlin, Cologne, Frankfurt, Hamburg, Hannover, Munich, Nurnberg, Stuttgart

PRINTED IN USA EB 18816 10/81 02B 07 20.0 COPYRIGHT (C) 1981 DIGITAL EQUIPMENT CORPORATION