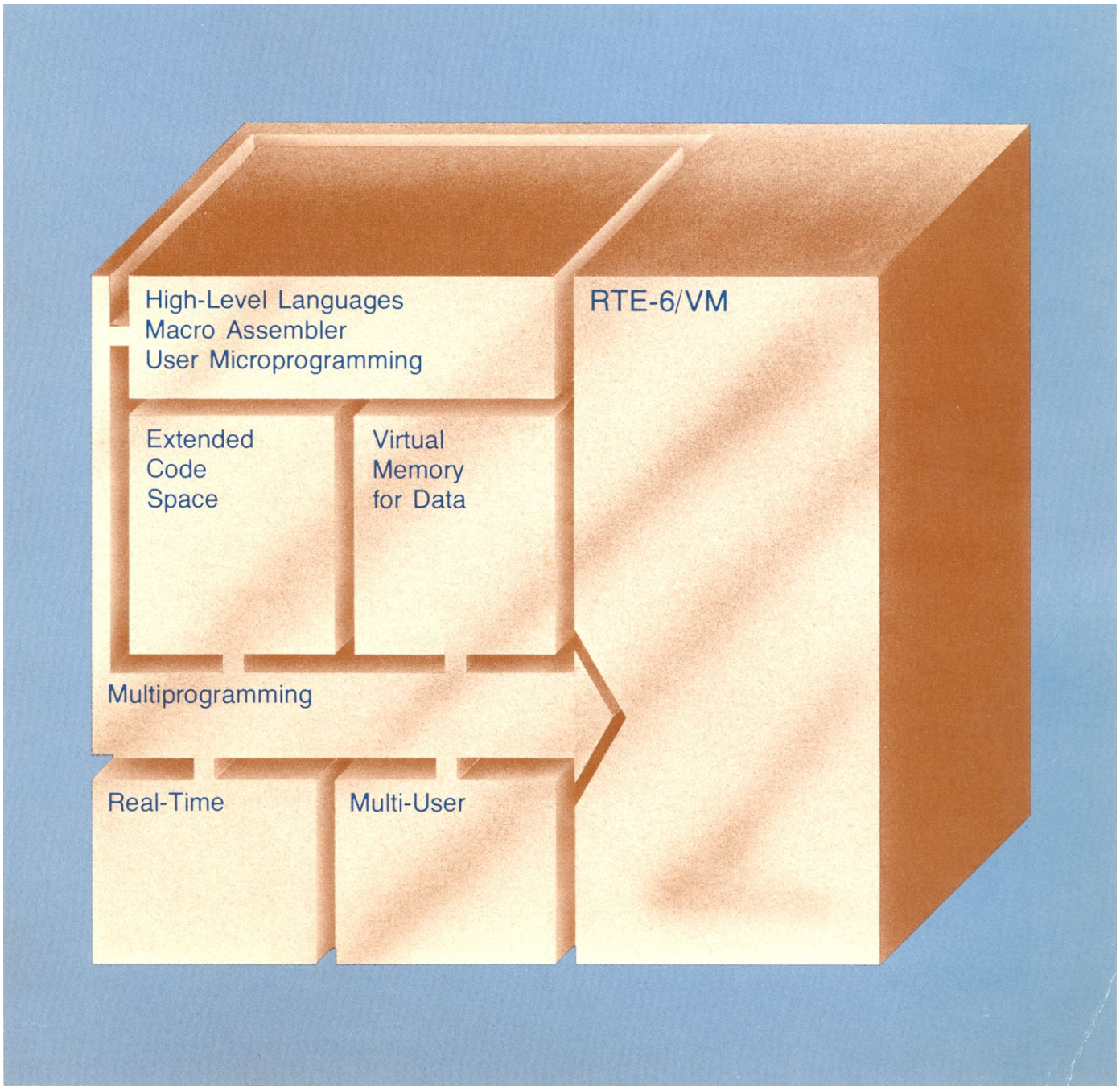
HEWLETT PACKARD

RTE-6/VM Programmer's

Reference Manual



HEWLETT PACKARD

RTE-6/VM Programmer's

Reference Manual

Data Systems Division
11000 Wolfe Road
Cupertino, CA 95014-9974

Part No. 92084-90005
U0186

Printed in U.S.A. January, 1985
Updated January, 1986

Printing History

The Printing History below identifies the Edition of this Manual and any Updates that are included. Periodically, Update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this Printing History page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past Updates, however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all Updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the appropriate Software Numbering Catalog, Software Product Catalog, or Diagnostic Configurator Manual.

First Edition	Dec 1981	
Update 1	Jul 1982	
Update 2	Jan 1983	Corrections and Additions to text
Update 3	Dec 1983	
Reprint	Dec 1983	Updates 1,2,& 3 Incorporated
Update 4	Jan 1985	
Reprint	Jan 1985	Update 4 Incorporated
Update 5	Jan 1986	

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company.

Copyright © 1981-1983,1985-1986 by HEWLETT-PACKARD COMPANY

Preface

This manual describes the scope, format, and use of the RTE-6/VM operating system services available to user-written programs. It is intended to be the primary reference source for programmers who will be responsible for writing and maintaining software within the RTE-6/VM operating environment.

This manual is divided into five sections as follows:

Chapter 1 gives a general description of the RTE-6/VM operating system features.

Chapter 2 describes the use of the Executive communication module of RTE-6/VM that provides user-written programs with the ability to communicate with the operating system. The Executive allows programs to perform I/O operations, manage programs, manage system resources, and obtain status information about the system.

Chapter 3 describes the use of files and the File Management Package (FMP) by user-written programs. By calling routines contained in the File Management Package (FMP library), user-written programs can define, access, position within, and purge disc or non-disc files.

Chapter 4 defines and describes the use of the Virtual Memory Area (VMA) and the Extended Memory Area (EMA) features of RTE-6/VM.

Chapter 5 describes the format and use of the several routines contained in the System Library. The System Library contains user-callable subroutines that provide a variety of utility and special purpose functions.

Appendix A Error Messages
Appendix B VMA/EMA Mapping Management Subroutines
Appendix C System Communication Area and System Tables
Appendix D Program Related Tables
Appendix E I/O Tables and Processing
Appendix F Memory Management and Related Tables
Appendix G DCB and Directory Formats
Appendix H Session Monitor Tables
Appendix I Record Formats
Appendix J Scheduling FMGR Programmatically

For additional information on the RTE-6/VM operating system or any of its subsystems, refer to the RTE-6/VM Index and Glossary Manual. This guide contains a complete glossary, commonly used tables, a general index to all RTE-6/VM manuals, and a documentation map.

Two reference manuals that are directly related to the RTE-6/VM operating system are briefly described below:

* RTE-6/VM Terminal User's Reference Manual. This manual describes the features of the RTE-6/VM operating system that are available to the user in an interactive mode.

* RTE-6/VM System Manager's Manual. This manual contains the information necessary to plan, generate, and maintain the RTE-6/VM operating system.

NOTE

File management in this manual is based on the File Management Package (FMP). Refer to the CI User's Manual for details on the command interpreter hierarchical file system.

Table of Contents

Chapter 1 General Description

Introduction 1-1
Multiprogramming and Timeslicing 1-3
Program Types 1-6
Memory Management 1-6
Memory Maps 1-7
Physical Memory 1-8
Common Areas. 1-12
Memory Protection 1-13
Program Partitions. 1-15
Program Segmentation. 1-16
Input/Output Processing 1-17
Hardware Considerations 1-19
Logical Unit Numbers. 1-20
Power Fail. 1-21
I/O Controller Time-Out 1-21
Privileged Interrupt Processing 1-22
Resource Management 1-23
Session Monitor 1-24
Language Support 1-25
Executive Communication 1-27
File Management System 1-28
System Library 1-29
Spooling System 1-30
System Utility Programs 1-31

Chapter 2 Executive Communication

Introduction 2-1
EXEC Call Formats 2-3
XLUEX I/O Extended Logical Unit EXEC 2-8
EXEC Call Error Returns 2-9
   No-Abort Option 2-10
   No-Suspend I/O Option 2-14
EXEC Description Conventions 2-16
Standard I/O EXEC Calls 2-18
   Read or Write Call - EXEC 1 or 2 2-19
   I/O Control Call - EXEC 3 2-24
Class I/O EXEC Calls 2-27
   Class I/O Operation 2-30
   Class Read, Write, and Write/Read 2-33
   Class I/O Control - EXEC 19 2-39
   Class Get - EXEC 21 2-41
Program Management EXEC Calls 2-49

Program Completion - EXEC 6 2-50
Program Suspend - EXEC 7 2-53
Program Segment Load - EXEC 8 2-55
Program Scheduling - EXEC 9,10,23, and 24. 2-57
Program Time Scheduling - EXEC 12 2-63
String Passage - EXEC 14 2-67
Program Swapping Control - EXEC 22 2-70
Status EXEC Calls 2-72
Time Request - EXEC 11 2-72
I/O Status - EXEC 13 2-74
Partition Status - EXEC 25 2-79
Memory Size - EXEC 26 2-81
Disc Track Management EXEC Calls 2-84
Disc Track Allocation - EXEC 4 or 15 2-85
Disc Track Release - EXEC 5 or 16 2-87

Chapter 3 File Management via FMP

Introduction 3-1
Files 3-1
Type 0 Files 3-2
Type 1 Files 3-3
Type 2 Files 3-3
Type 3 Files 3-3
Type 4 Files 3-4
Type 5 Files 3-4
Type 6 Files 3-4
Type 7 Files 3-4
File Types Greater Than 7 3-4
File Access 3-5
File Extents 3-5
File Updates 3-6
Cartridges 3-7
Cartridge and File Directories 3-10
File Security 3-11
Cartridges in the Session Environment 3-12
FMP Calls 3-15
The Data Control Block 3-15
Data Transfer 3-17
FMP Call Formats 3-19
Common Parameters 3-21
IDCB 3-21
IERR 3-23
INAM 3-23
IBUF 3-23
Optional Parameters 3-24
ISC 3-24
ICR 3-25
IDCBZ 3-26

FMP Call Description Conventions 3-27
File Definition FMP Calls 3-28
CREAT and ECREA Calls 3-29
CRETS Call 3-34
OPEN and OPENF Calls 3-37
CLOSE and ECLOS Calls 3-44
PURGE Call 3-47
File Access 3-49
READF and EREAD Calls 3-50
WRITF and EWRIT Calls 3-56
File Positioning 3-60
LOCF and ELOCF Calls 3-61
APOSN and EAPOS Calls 3-64
POSNT and EPOSN Calls 3-67
RWNDF Call 3-71
Special Purpose FMP Calls 3-72
FCONT Call 3-73
FSTAT Call 3-76
IDCBS Call 3-80
NAMF Call 3-81
POST Call 3-82
Examples Using FMP Calls 3-84

Chapter 4 VMA and EMA Programming

Introduction 4-1
Virtual Memory Area (VMA) 4-3
Extended Memory Area (EMA) 4-6
Using Shareable EMA 4-7
Shareable EMA Program Considerations 4-8
Partition Considerations 4-9
Mother Partitions 4-9
Shareable EMA Partitions 4-11
Programming with VMA/EMA 4-12
Declaring Extended Memory Area (EMA) 4-13
General Purpose VMA/EMA Subroutines 4-16
EMAST Subroutine (Return Information on VMA/EMA) 4-16
VMAST Subroutine (Return Size of VMA/EMA) 4-18
VMAIO Subroutine (Perform Large VMA/EMA Data Transfers) . 4-20
EIOSZ Subroutine (Determine Maximum Length of Transfer) . 4-22
LKEMA/ULEMA (Lock/Unlock a Shareable EMA Partition) . . . 4-23
VMA File Subroutines 4-25
CREVM Subroutine (Create a VMA Backing Store File) 4-26
OPNVM Subroutine (Open a VMA Backing Store File) 4-28
PURVM Subroutine (Purge VMA Backing Store File) 4-30
PSTVM Subroutine (Post Working Set to Disc) 4-31
CLSVM Subroutine (Close the VMA Backing Store File) . . . 4-32
VREAD Subroutine (Read Data From a File to a VMA/EMA) . . 4-33
VWRIT Subroutine (Write Data From a VMA/EMA to a File) . . 4-35
Examples Using VMA File Subroutines 4-37

Chapter 5 Applications Using System Routines

Introduction. 5-1
Calling Library Subroutines 5-1
Reentrant Subroutine Structure. 5-1A
Privileged Subroutine Structure 5-1C
Memory-Resident Library 5-1E
Utility Subroutine Structure. 5-1F
Format of Routines. 5-1F
Routines Callable from FORTRAN. 5-1H
Routines Callable from Pascal 5-1H
Microcoded Subroutines. 5-1J
  Fast FORTRAN Processor. 5-1J
  Floating Point Library. 5-1J
Syntax Conventions. 5-2
Class I/O Applications 5-3
   CLRQ - Class I/O Management 5-3
Resource Numbers and Logical Unit Lock Applications 5-14
   RNRQ - Resource Management 5-16
   LURQ - Logical Unit Lock 5-20
Parameter Passage Applications 5-30
   RMPAR - Retrieve Parameters 5-31
   PRTN, PRTM - Parameter Return 5-33
   GETST - Recover Parameter String 5-35
I/O Requests and Related Routines 5-38
   .STIO - Configures Driver 5-38
   BINRY - Data Transfer 5-39
   EQLU - Finds Logical Unit Number 5-41
   IFTTY - Logical Unit Interactive? 5-42
   LOGLU - Logical Unit Return 5-43
   LUTRU - Returns True System Logical Unit 5-44
   REIO, XREIO - Reentrant I/O 5-45
   TRMLU - Finds Logical Unit Number 5-47
Session Related Routines 5-48
   GTERR - Returns SCB Error 5-48
   GTSCB - Returns contents of current SCB 5-50
   ICAPS - Returns Current Capability Level 5-51
   LUSES - Scans Session Control Blocks 5-52
   PTERR - Updates Error in Session Control Block 5-53
   SESSN - Calling Program in Session? 5-54
System Calls and Entry Points 5-55
   OPSYS - Returns $OPSY Value 5-55
   SEGLD Overlay Load with Link 5-56
   SEGLD with Loader 5-57
   SEGRT - Returns Instructions After SEGLD Request 5-59
   SETTM - Set System Time 5-61
   TATMP - Maps Track Assignment Table 5-63
Program Status Routines 5-64
   COR.A, COR.B - Returns Address 5-64
   IDGET - Retrieve ID Segment 5-66
   IFBRK - Test Break Flag 5-67
   LIMEM - Manages Memory 5-68
Conversion Routines 5-70
   $CVT3(CNUMD,CNUMO), $CVT1(KCVT) - Convert Binary to ASCII. 5-70

Miscellaneous 5-72
.DRCT - Resolves Indirect Address 5-72
FTIME - Returns Date and Time 5-73
INPRS - Inverse Parse 5-74
MESSS - Programmatic Access 5-75
PARSE ($PARS) - Parse String 5-77
SYCON - Writes Message to System Console 5-79
TMVAL - Reformats System Time 5-80
XQPRG - Load and Execute a Program 5-82

Appendix A Error Messages

Executive Error Messages A-1
Memory Protect Violations A-2
Dynamic Mapping Violations A-2
Dispatching Errors A-3
EX Errors A-3
Unexpected DM and MP Errors A-4
TI, RE and RQ Errors A-5
Disc Parity Error (Track Error) A-5
Parity Errors A-6
Other EXEC Errors A-8
Disc Allocation Error Messages A-8
Schedule Call Error Codes A-8
I/O Call Error Codes A-10
I/O Error Message Format A-12
Program Management Error Codes A-13
Logical Unit Lock Error Codes A-13
CLRQ A-13
Executive Halt Errors A-14
Error Routing A-16
FMP Error Codes A-24
VMA/EMA Errors A-33

Appendix B VMA/EMA Mapping Management Subroutines

.IMAP Subroutine B-4
.IRES Subroutine B-6
.JMAP Subroutine B-7
.JRES Subroutine B-8
MMAP Subroutine B-9
.ESEG Subroutine B-11
.LBP, .LBPR Subroutine B-12
.LPX, .LPXR Subroutine B-13
.EMIO Subroutine B-14

Appendix C System Communication Area and System Tables

System Communication Area C-1
Program ID Segment C-5
ID Segment Extensions C-10
Short ID Segments C-11
RTE-6/VM System Disc Layout C-11

Appendix D Program Related Tables

Program States D-4

Appendix E I/O Tables and Processing

Equipment Table (EQT) E-1
Device Reference Table (DRT) E-6
Driver Mapping Table (DMT) E-9
Interrupt Table and Trap Cells E-11
   Interrupt Handling Without Microcode E-11
   Interrupt Handling with Microcode E-13
Power Fail/Auto Restart E-13
Standard I/O Request Flow. E-16

Appendix F Memory Management and Related Tables

Address Translation F-1
Logical Memory and Base Page F-3
Memory Allocation Table Entry F-7

Appendix G DCB and Directory Formats

Data Control Block (DCB) Format G-2
Cartridge Directory Format G-4
File Directory G-5
   Disc File Directory G-6
   Type 0 File Directory Entry G-7

Appendix H Session Monitor Tables

Session Control block (SCB) H-1
Session Switch Table (SST) and Configuration Table H-3
Session Table Relationship H-4

Appendix I Record Formats

Source Record Formats I-1
Relocatable and Absolute Record Formats I-3
Absolute Tape Format I-9
Disc File Record Formats I-10
SIO Record Format I-11
Memory-Image Program File Formats (Type 6) I-12

Appendix J Scheduling FMGR Programmatically

List of Illustrations

Figure 1-1. Scheduling with Time-Slicing 1-5
Figure 1-2. Physical Memory Allocations. 1-9
Figure 1-3. Memory Protect Fence Locations 1-14
Figure 2-1. EXEC 26 Parameter Relationships. 2-82
Figure 3-1. Disc Cartridge Organization. 3-8
Figure 3-2. Sequential Transfer Between Disc Files/Buffers . 3-17
Figure 3-3. Data Transfer with Type 1 Files. 3-18
Figure 3-4. Read Type 1 File when IL Greater than 128. . . . 3-52
Figure 3-5. Sample WRITE to Type 1 File. 3-58
Figure 4-1. Working Set and Backing Store File 4-5
Figure 4-2. EMA in Physical Memory 4-6
Figure 5-1. Class I/O Multiple Terminal Input Example. . . . 5-8
Figure 5-2. Dispatching Input to Subtasks for Processing . . 5-12
Figure 5-3. Control Word Format (ICON) 5-17
Figure 5-4. Resource Number Example 1. 5-24
Figure 5-5. Resource Number Example 2. 5-25
Figure 5-6. "Deadly Embrace" Example 1 5-27
Figure 5-7. "Deadly Embrace" Example 2 5-27
Figure 5-8. "Deadly Embrace" Example 3 5-28
Figure B-1. VMA/EMA and Memory Structure B-2
Figure C-1. ID Segment Format. C-6
Figure C-2. ID Segment Extension C-10
Figure C-3. RTE-6/VM System Disc Layout. C-12
Figure D-1. User Program State Diagram D-6
Figure E-1. Equipment Table Entry Format E-3
Figure E-2. CONWD Word (EQT Entry Word 6) Expanded E-6
Figure E-3. Device Reference Table (DRT) E-7
Figure E-4. Device Reference Table Entry Format. E-8
Figure E-5. Driver Mapping Table E-10
Figure E-6. Unbuffered EXEC Read Request Flow. E-18
Figure F-1. Address Translation via Memory Mapping F-2
Figure F-2. RTE-6/VM 32K Word Logical Memory Configurations. F-5
Figure F-3. Base Page Structure. F-6
Figure F-4. Memory Allocation Table Entry Format F-7
Figure H-1. SCB H-2
Figure H-2. Session Switch Table (SST) Format. H-4
Figure H-3. Configuration Table. H-5
Figure H-4. Session Table Relationships. H-6
Figure I-1. Source Record Formats. I-2
Figure I-2. Record Formats I-4

List of Tables

Table 2-1. Summary of EXEC Calls 2-1
Table 2-2. Index of EXEC Calls 2-3
Table 2-3. Action Taken When the No-Suspend Bit is Set . . . 2-15
Table 2-4. Function Code 2-25
Table 2-5. Class I/O Terms 2-28
Table 2-6. Summary of EXEC Calls 9,10,23,24. 2-61
Table 2-7. I/O Status Word (ISTA1/ISTA2) Format. 2-76
Table 2-8. Device Status Table (EQT Word 5). 2-77
Table 3-1. Categories of File Types. 3-2
Table 3-2. FMP Call Summary. 3-16
Table 3-3. Relation of Actual to Req. Packing Buffer Size. . 3-22
Table 3-4. OPENF Defaults. 3-40
Table 3-5. Effect of IL Parameter in READF 3-51
Table 3-6. Effect of IL Parameter in WRITF 3-57
Table 3-7. Relationship Between NUR and IR 3-68
Table 3-8. FCONT Function Codes. 3-74
Table 3-9. ISTAT Format I. 3-78
Table 3-10. ISTAT Format II 3-79
Table 4-1. VMA and EMA Terms 4-2
Table 4-2. Common VMA/EMA Subroutines. 4-15
Table A-1. EXEC Call Error Summary A-19
Table B-1. VMA/EMA Mapping Management Subroutines. B-3
Table C-1. System Communications Area Locations. C-2
Table D-1. Summary of RTE-6/VM Program Types D-2
Table E-1. Interrupt Table Example E-12

Chapter 1
General Description

Introduction

RTE-6/VM is a disc-based operating system that provides the supervisory functions necessary to coordinate requests for, and allocation of, system services and resources. RTE-6/VM, a real-time system, processes all decision and scheduling tasks internally unless overridden by user intervention. User requests for system action can be made by a "call" from within a program or interactively via an operator command.

As the major control element within the operating environment, RTE-6/VM provides the user with various services and automatically handles the machine-related functions associated with each service. The major services provided by RTE-6/VM are briefly summarized below:

* Executive Communication scheme that provides a communication link between user-written programs and system services.

* Segmentation techniques that allows a large program to be separated into a main section of code and related segments, thereby allowing it to execute in a memory partition smaller than its total size. Segmentation can be implemented by the MLS-LOC Loader or programmatically via an EXEC call.

* Resource Management capabilities that allow cooperating user-written programs to share system resources (files, I/O devices, etc.).

* I/O scheme which allows a program to continue executing while its own I/O requests are being processed.

* Program execution control that features multiprogramming (allows several programs to be active concurrently) and time-slicing (prevents compute intensive programs from dominating the CPU).

* Partitioned memory technique that takes advantage of the hardware Dynamic Mapping System (DMS) to provide access to 2048k bytes of physical memory.

* Extended Memory Area (EMA) that allows user-written programs to access large data arrays; the size of the arrays being limited only by the size of physical memory. Large areas of EMA can be shared among programs for program to program communication. The EMA data to be shared resides in a shareable EMA partition.

* Demand-Paged Virtual Memory that provides RTE-6/VM programs with the capability to access very large data areas up to 128 megabytes.

* Operator interface that provides the user with the ability to control system action via operator commands.

In addition to the features listed above that are inherent to RTE-6/VM, software modules are available with the operating system that provide the user with additional capabilities. These features are as follows:

* File Management System
* Spooling System
* Session Monitor
* FORTRAN Compiler
* Macroassembler
* Pascal/1000 Compiler
* Compile Utility
* Compile and Load Utility
* MLS-LOC Loader
* MLS-LOC Loader Command File Utilities
* Interactive Relocating Loader
* Indexed Relocatable Library Utility
* Interactive Editor
* Debug Utilities
* On-Line Generator
* System Status Utilities
* File Merge Utility
* Disc Cartridge Save/Restore Utilities
* File Backup Utility
* Disc Backup Utilities
* Terminal Soft Keys Utility
* Track Assignment Table Log Program
* Source File Comparison Utility
* On-Line Driver Replacement Utility
* Help Utilities

The features described above that relate to the programmatic control of system action are described in later sections of this manual along with background information on the RTE-6/VM operating system. The features of RTE-6/VM that relate to the interactive control of system action are described in the RTE-6/VM Terminal User's Reference Manual and the appropriate subsystem manuals (see documentation map in the RTE-6/VM Index to Operating Systems Manuals). For information concerning the generation and configuration of the RTE-6/VM operating system, refer to the RTE-6/VM On-Line Generator Manual or obtain the RTE-6/VM System Manager's Manual.

Multiprogramming and Timeslicing

RTE-6/VM is a multiprogramming system that allows several programs to be active concurrently. Each program executes during the unused central processor time of the others. Scheduling/dispatching modules in the RTE-6/VM operating system decide when to execute programs that are simultaneously requesting system services and/or resources. The scheduling module places programs into a scheduled list in order of their priority (the highest priority program at the head of the list) and the dispatching module initiates the execution of the highest priority program. Programs with the same priority are scheduled on a first-come, first-serve basis. When the executing program completes, is terminated, or is suspended, it is removed from the scheduled list, and the dispatching module transfers control to the next program with the highest priority. Note that the next program to be executed could have the same priority as the program that was just removed from the list.

The scheduled list can be logically divided into two areas by placing a time-slicing boundary at a priority level. Programs with priorities that place them above the boundary (higher priority, lower numerically) are executed in the linear fashion described above.

Programs with priorities that place them below the boundary (lower priority, higher numerically) are executed in a similiar fashion with one exception; programs are assigned an execution interval when they are scheduled. When a program exceeds its interval, it is moved within its priority level in the scheduled list.

Each priority level below the time-slicing boundary can be considered a queue. The program at the head of each priority queue represents the next program of that priority to be executed. When the execution of the program at the head of the queue is initiated, a maximum time interval for execution (time quantum) is calculated by the operating system. The program is allowed to execute until one of the following occurs:

1. The program leaves the scheduled list (I/O suspended, memory suspended, etc.)

2. A higher priority program is ready to execute.

3. The program exceeds its time quantum.

If a program leaves the scheduled list, its time quantum is assumed exhausted. When the program is again ready to execute, it is placed at the end of the queue within its priority in the scheduled list and a new time quantum is established.

If a higher priority program causes the suspension of a time-slicing program, the remaining portion of the suspended program's time quantum is saved in its ID segment. When the suspended program is scheduled to continue executing, the saved quantum value is restored.

When a time-slicing program exceeds its time quantum, it is placed at the end of the queue within its priority in the scheduled list and control is transferred to the new head of the queue.

The time value used to calculate the quantum and the time-slicing boundary are manipulated by the QU command described in the RTE-6/VM Terminal User's Reference Manual; the considerations for manipulating them are discussed in the RTE-6/VM System Manager's Manual. Figure 1-1 shows a diagram of scheduling with time-slicing.

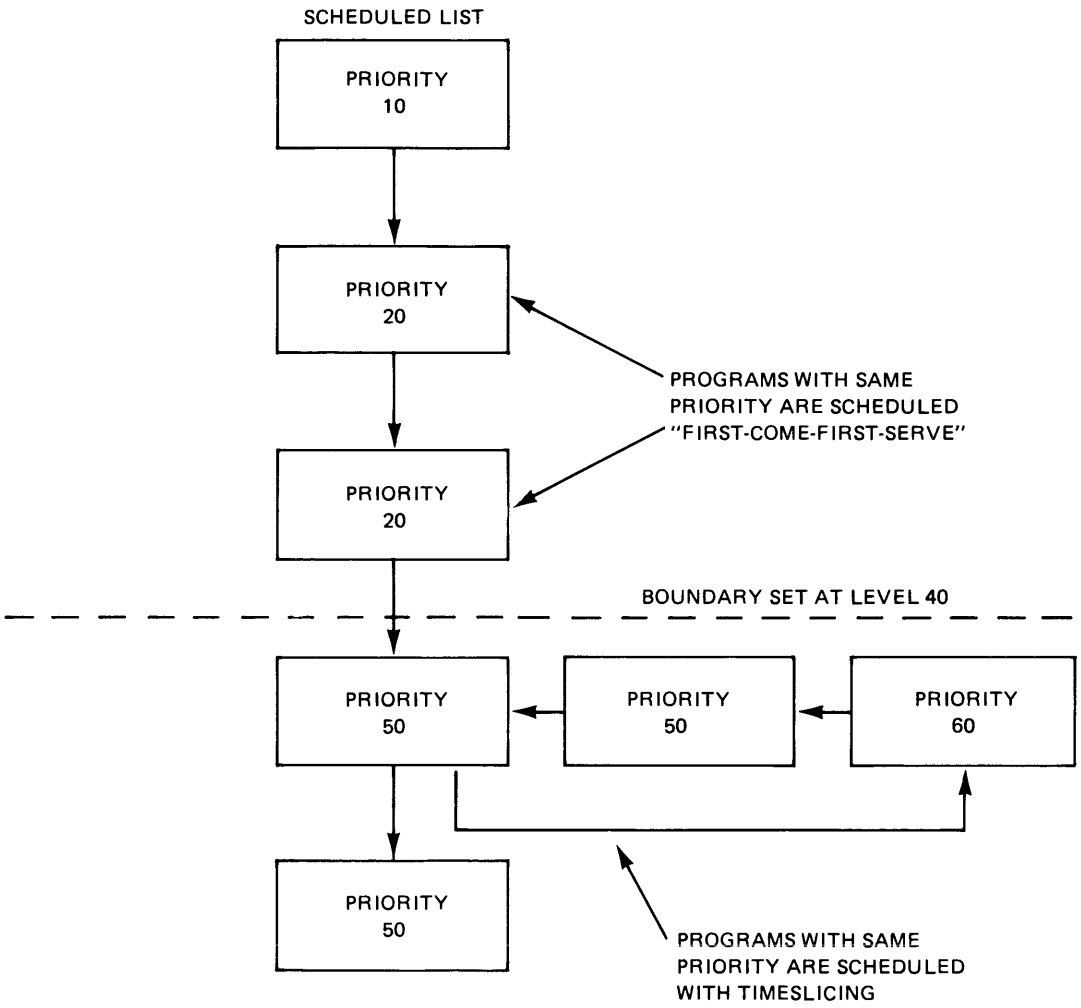


Figure 1-1. Scheduling with Time-Slicing

Program Types

Programs within the RTE-6/VM operating system are categorized according to where they reside when not executing (memory-resident or disc-resident), what type of memory partition they execute in, by the COMMON areas they have access to, and whether or not they may be duplicated. A program can be defined as a real-time or a background program. A background program can be further defined as a large background or extended background program in order to handle large user programs. A program's type can also indicate whether it is a main program, a program segment, a multi-level segment program, or a subprogram (refer to Appendix D for a summary of program types).

The user specifies the program either in the program definition statement (PROGRAM statement in FORTRAN, NAM statement in Macro/1000) or as a compiler option in Pascal. Otherwise, the user can assign the program type when the program is loaded or allow the program to default to it by leaving the program undefined. LOADR and MLLDR assign this default type; LINK and the generator use the program type parameter predefined in the NAM record.

A program's type, along with other necessary information, is maintained in memory by the system in the program's ID segment (refer to Appendix C for the ID Segment format).

Memory Management

The RTE-6/VM operating system is written to take advantage of the hardware Dynamic Mapping System (DMS) available with the HP/1000 computers. The cooperation between the software operating system and the hardware mapping system allows direct access to 1024K words (2 megabytes) of physical memory. More physical memory can be accessed with virtual memory.

The basic addressing space of the HP/1000 computer is 32,768 words (32K) as defined by the 15-bit address length used by the CPU. This is referred to as logical memory. The amount of memory actually installed in the computer system is referred to as physical memory. The DMS maps 32K words of physical memory into logical memory by translating the 20-bit physical addresses into 15-bit logical addresses through "memory maps".

Memory Maps

The page pointers contained in the map registers that comprise the memory maps are loaded by software modules within the operating system. Each map is configured to represent the 32 pages of physical memory (not necessarily contiguous) that contain the tables, buffers, data, program code, etc., necessary to perform specific processing tasks. Since there are four maps, four different 32 page sections of physical memory can be described simultaneously. The maps are altered by the operating system to reflect dynamic changes in the operating environment, i.e., when a program is scheduled to execute, a map has to be configured to describe the physical memory pages it requires (known as the program's logical address space).

The four memory maps are classified by the type of processing tasks they are associated with. The maps are comprised of the following:

* SYSTEM MAP---The System Map describes the logical address space associated with the RTE-6/VM operating system, including its base page, COMMON, Subsystem Global Area, Table Area I and II, driver partition, operating system code partition, System Driver Area (SDA), and System Available Memory (SAM). The system map is loaded during system initialization and is changed to map in different driver partitions or operating system code partitions on demand. Since the RTE-6/VM operating system handles all interrupt processing, the system map is automatically enabled by the hardware whenever an interrupt occurs.

* USER MAP---The User Map is associated with each disc-resident program. It is a unique set of pages that describe the logical address space containing the program's code, the program's base page, and optionally, Table Area I, driver partition, Table Area II, System Driver Area, and COMMON.

  All memory-resident programs use a common set of pages that define the memory occupied by the memory-resident program and its base page, the Memory-Resident Library, Table Area I, driver partition, COMMON, and optionally, Table Area II and System Driver Area. Memory-resident programs are brought into memory at boot-up and remain in memory. Therefore, these programs may be scheduled and executed quickly.

Each time a new memory-resident or disc-resident program is dispatched, the system reloads the User Map with the appropriate set of pages.

* PORT MAP A and PORT MAP B. The Port maps are associated with DCPC transfers. DCPC transfers are software assignable direct data paths between memory and a high speed peripheral device.

  This function is provided by the Dual Channel Port Controller (DCPC). There are two DCPC channels, each of which may be assigned to operate with an I/O device. Port A Map is automatically enabled when a transfer occurs on DCPC channel 1, and Port B Map is enabled when a DCPC channel 2 transfer occurs.

  DCPC transfers are accomplished by "stealing" CPU cycles instead of interrupting the CPU and transferring to an I/O service routine. Having separate maps associated with DCPC transfers, and having the transfer implemented by "cycle-stealing", facilitates multiprogramming since one program can be executing via the User Map while a DCPC transfer is in progress on another program's data buffer.

  The Port Maps are reloaded by the system each time a DCPC channel is assigned for an I/O request. The Port Maps will be the same as the System Map or the User Map associated with the program being serviced, depending on the type of request. Once initiated, the DCPC transfer is transparent to the user since the currently enabled map (System or User) shares the CPU with the Port Maps, i.e., during a given instruction cycle (comprised of several CPU cycles) the System and User Map is enabled alternately with the Port Map. Therefore, a maximum of three memory maps may be enabled concurrently, one execute map (System or User) and both Port Maps.

Physical Memory

At generation time, the user plans physical memory allocations and loads the system components and drivers for the most efficient configuration. The user determines the size of System Available Memory (SAM), the number and size of each partition, the size of COMMON, and the size and composition of the resident library and memory-resident program area. Refer to the RTE-6/VM On-Line Generator Manual and the RTE-6/VM System Manager's Manual for a description of the procedures used to configure physical memory.

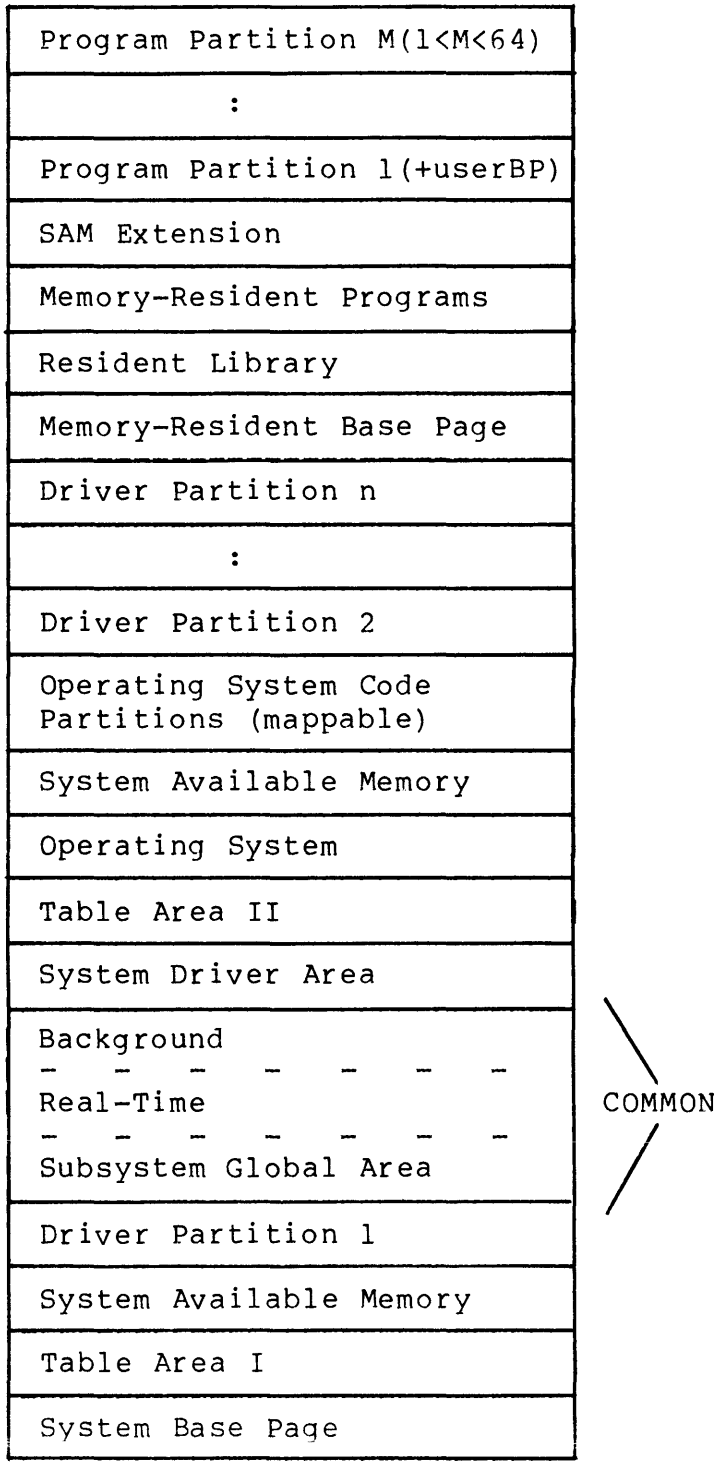


Figure 1-2. Physical Memory Allocations

The following is a brief description of the Physical Memory Configuration shown in Figure 1-2:

* System Base Page - Contains the operating system communication area and is used by the system to define request parameters, I/O tables, scheduling lists, pointers, operating parameters, memory bounds, etc. System links and trap cells are also located on the system base page.

  Base page links for the memory-resident library and memory-resident programs are only in the memory-resident base page and are not accessible by disc-resident programs. The Table Areas, SSGA, driver links, and the system communication area are accessible to all programs. Partition base pages, used for disc-resident program links, are described below with partitions. For all practical purposes, the memory-resident programs are in a single partition separate (protected) from all other partitions.

* Table Area I - Contains system tables, Equipment Table entries, Driver Mapping Table, Device Reference Table, Interrupt Table, the Disc Track Map Table, some system and HP subsystem entry points, and all Type 15 modules.

* Driver Partition - An area, established at generation time containing one or more drivers. All driver partitions are the same length, and only one driver partition is included in a 32K-word address space at any one point in time. The minimum driver partition size is two pages but may be increased when the system is generated.

* COMMON - This area is divided into three subareas: the Subsystem Global Area (SSGA), the Real-time COMMON area, and the Background COMMON area. SSGA is used by Hewlett-Packard software subsystems for buffering and communications. The Real-time and Background sub-areas (system COMMON) are reserved for user-written programs that declare COMMON. All programs relocated during generation time that declare COMMON will reference this system COMMON. Programs relocated on-line with LOADR or MLLDR may choose to reference system COMMON or use local COMMON.

* System Driver Area - An area for privileged drivers, large drivers, or drivers that do their own mapping. The drivers that go into this area are specified during the EQT definition phase of system generation. The System Driver Area (SDA) is included in the logical address space of the system, Type 2 and 3 programs, and optionally Type 1 programs.

* Table Area II - Contains the Keyword Table, ID segments, ID Segment Extensions, Class Table, Batch LU Switch Table, Memory-Resident Map, and a number of entry points for system pointers. This area has entry points that are created by the generator and others that are defined by Type 13 modules.

* System - Contains the absolute code of the Type 0 system modules (i.e., RTIOQ, SCHED, EXEC, etc.).

* Operating System Code Partitions - Six two page areas (partitions) of memory containing portions of the RTE-6/VM operating system. RTE-6/VM will map this physical area of memory into the logical driver partition area, as it needs the code in that area to satisfy a user's request. This area includes certain I/O modules.

* Memory-Resident Library - Contains the re-entrant or privileged library routines (Type 6) that are used by the memory-resident programs, or which are force loaded at generation time (Type 14). It is accessible only by memory-resident programs. All routines loaded into the resident library also go into the relocatable library for appending to disc-resident programs that require them.

* Memory-Resident Programs - This area contains all Type 1 programs that were relocated during generation.

* System Available Memory (SAM) - This is a temporary storage area used by the system for buffered I/O, Class I/O, re-entrant I/O, and parameter string passing.

* Program Partitions - This is an area established by the user for a disc-resident program to execute in. Each partition has its own base page that describes the linkages for the program running in the partition. Up to 64 partitions are allowed, within the constraints of available physical memory.

Common Areas

The real-time and background COMMON, along with Subsystem Global Area, occupy a contiguous area in memory and are treated as a single group for mapping purposes (refer to Figure 1-2). The use of COMMON is optional on a program basis, that is, any program may use real-time COMMON, background COMMON or no COMMON. If the program declares COMMON and the user chooses not to use local COMMON, both COMMON areas and the Subsystem Global Area will be included in the User Map. If a large background program does not use COMMON, it will not be included in the User Map, providing the user with a larger program area in the 32K-words of logical address space. Refer to Chapter 4 for a description of using shareable EMA as COMMON.

REAL-TIME AND BACKGROUND COMMON. If a program declares at least one word of COMMON, the use of real-time or background COMMON is selected by program type at generation or parameters with the on-line loader. Program types are summarized in Appendix D.

These system COMMON areas are not to be confused with the local COMMON area that may be specified for programs loaded on-line. The system COMMON areas are shareable by programs operating in different partitions, whereas the local COMMON area will be in its partition and is accessible only to that program, its subroutines and segments.

SUBSYSTEM GLOBAL AREA. The Subsystem Global Area (SSGA) consists of all Type 30 modules input to the generator. Accessed by entry point (using EXT statements) rather than COMMON declarations, SSGA provides multiple communication and buffer areas for Hewlett-Packard subsystems. SSGA access is enabled by program type at generation or through special parameters during on-line loading. Programs authorized for SSGA access have the COMMON area included in their maps and have the memory protect fence set below SSGA.

Memory Protection

Memory protection is provided by a combination of the Dynamic Mapping System and the Memory Protect Fence. DMS provides protection between program partitions by not allowing a program to access memory locations that are not defined by its memory map. The Memory Protect Fence prevents a program from addressing memory locations below a given address within its memory map.

A combination of DMS and the Memory Protect Fence provides protection for the driver partition, Table Area I, System Driver Area, Table Area II, and COMMON by preventing stores and jumps to locations below a specified address.

The Memory Protect Fence indicates the logical address space where addresses are compared to the fence before translation. If a disc-resident program does not use any of the COMMON areas, the Memory Protect Fence is set at the bottom of the program area. Similarly, for a memory-resident program not using COMMON, the Memory Protect Fence is set at the base of the memory-resident area.

For programs using system COMMON, the memory mapped includes all COMMON areas and the Memory Protect Fence is set at one of three possible locations, depending on the portion of COMMON being used. A hierarchy of protection is established within COMMON due to their physical locations. Background COMMON is the least protected (program's using any system COMMON can modify it) and SSGA is the most protected (only programs authorized for SSGA access can modify it). Figure 1-3 expands the COMMON area and shows these three fence settings as A, B, and C, respectively.

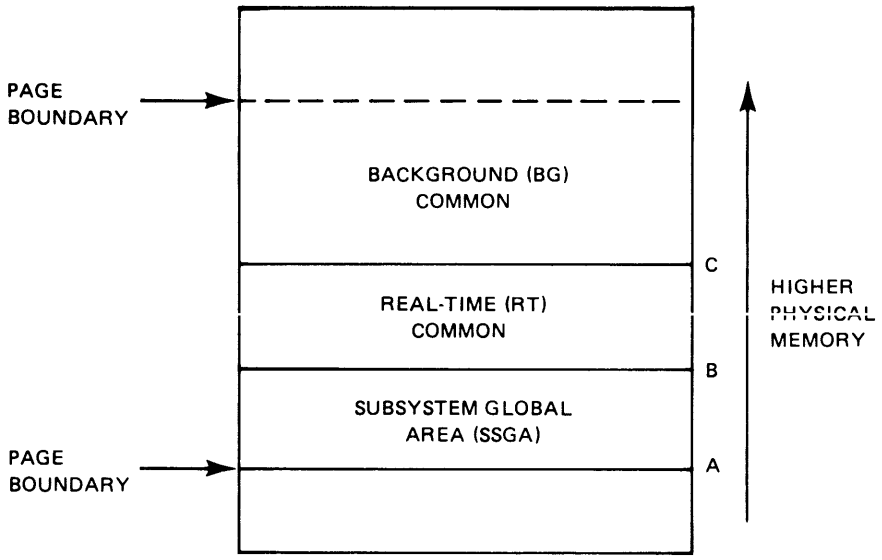


Figure 1-3. Memory Protect Fence Locations

Program Partitions

Program partitions are blocks of contiguous physical memory reserved for disc-resident programs. Program partitions are defined during system generation and may be redefined during the reconfiguration process at system boot-up (refer to the RTE-6/VM System Manager's Manual).

The number of partitions depends on the amount of available physical memory and the size of the defined partitions. Partition types can be specified as a mixture of real-time and background, all real-time, or all background. In addition, partitions may also be specified as mother partitions or shareable EMA partitions (refer to the discussion below). The user may optionally assign a program to run in any partition large enough to accommodate it. However, a program cannot be assigned to a shareable EMA partition, a mother partition with a shareable subpartition, or a subpartition of a shareable mother partition. Several programs can be assigned to the same partition, but only one program can run in that partition at a time. If a program is not assigned to a partition, then by default, real-time programs will run in real-time partitions, background programs in background partitions. In this case, the operating system decides which partition the program will execute in. If only one type of partition is defined, all programs will run in that type partition.

A mother partition is a large partition which is a collection of smaller real-time or background partitions called subpartitions. A mother partition is generally used to execute large programs or large VMA or EMA programs (that declare EMA data as local). When a mother partition is not in use, its subpartitions may be used by other programs. A mother partition or its subpartitions can also be defined as a shareable EMA partition.

Shareable EMA partitions are used for running EMA programs that declare EMA data as shareable data. If EMA data is to be shared with other programs, the declaring program will execute in one partition and the EMA data will reside in a shareable EMA partition. When a shareable EMA partition is not being used for data, other programs can run in the partition. Refer to Chapter 4, "VMA and EMA Programming" for more details on shareable EMA partitions.

Program Segmentation

Program segmentation allows a program's code to exceed the amount of logical or physical address space available to the program. Segmentation techniques allow a large program to be separated into a main section of code and related segments. This allows the large program to execute in a memory partition smaller than the total size of the program. Segmentation can be implemented automatically by the MLS-LOC Loader or programmatically via an EXEC call.

The preferred method of program segmentation is with the MLS-LOC Loader (MLLDR). MLLDR performs multilevel segmentation at load time. This allows program transportability since no changes are made to the source code in order to segment the program. Multilevel segmentation (MLS) is a tree-structured segmentation scheme which allows as much code as required to be memory-resident. A portion of code resides in logical memory. The remainder resides in physical memory and optionally on disc. The load-on-call feature is set up by MLLDR and is performed at execution time. When a subroutine is called, whether it is in logical memory, physical memory, or on disc, it is automatically made available for execution. The operating system performs any required memory mapping and disc access operations.

Two utilities are provided to aid development of MLS programs. The segmenter utility, SGMTR, reads a program's relocatable code and produces a segmentation structure for the program in the form of an MLLDR command file. The utility, SXREF, generates a program cross reference listing and checks the validity of MLS command files. Multilevel segmentation and the utilities, SGMTR and SXREF, are described in detail in the RTE-6/VM Loader Reference Manual.

Program segmentation can also be implemented programmatically via an EXEC call. These programs can only be loaded by LOADR. The large program is structured by the programmer during the coding process into a main program and several segments. When the code in one of the segments is required for execution, the currently executing program uses an EXEC 8 call to request the operating system to make a segment overlay. RTE loads the segment from the disc into a memory block following the end of the main program, overlaying whatever was previously there. When another segment is required, either the main program or the segment can make the EXEC call to request a segment overlay. Refer to Chapter 2 for an explanation of the EXEC 8 call.

Input/Output Processing

In the RTE-6/VM operating system, centralized control and logical referencing of I/O operations effect simple, device-independent programming. All I/O and interrupt processing is controlled by the operating system with the single exception of privileged interrupts (privileged interrupts circumvent the system for faster response time).

Programmatic requests for I/O services are made by EXEC routine calls coded into the calling program. The EXEC calls specify the type of transfer (Read, Write, Control) and the desired device. I/O requests from a program with a priority greater than 40 are queued to the controller's I/O list according to the calling program's priority. All other I/O requests are first-in, first-out. Automatic buffering for write operations is provided if specified at generation or with the system EQ command (refer to the RTE-6/VM Terminal User's Reference Manual).

In addition to the standard EXEC I/O scheduling processes described above, there are a number of other I/O functions available that can improve system performance in a multiprogramming environment:

* Device Time-Out -- sets a time-out value for a device to prevent indefinite program suspension because of a malfunctioning device.

* I/O Buffering -- automatic buffering on slower devices allows a calling program to initiate an output operation (only) without waiting for completion before resuming execution. An input without wait operation is a function of Class I/O (see below).

* Re-entrant I/O -- allows a disc-resident program to be swapped out of a memory partition and into disc storage when it is suspended for I/O. The status of the swapped program is maintained so that when the re-entrant I/O request has completed, and it once again achieves highest priority on the scheduled list, it can resume execution and I/O processing at the point of interruption.

* Logical Unit Lock -- assigns a logical unit (LU) exclusively to a specific program, thus preventing any other program from accessing it until it is unlocked.

* Class I/O -- a special set of I/O calls that provides a method for buffering data between two or more programs (mailbox I/O) or between programs and I/O devices. Class I/O permits a program to continue execution concurrently with its own I/O (I/O without wait).

Hardware Considerations

For a full understanding of the software I/O characteristics of the RTE-6/VM operating system described in this manual, the user should be familiar with two hardware-related terms:

1. I/O Controller - a combination of I/O card, cable and, for some devices, a controller box used to control one or more I/O devices associated with a computer I/O select code. (Select code refers to a physical card slot in the backplane of the computer.)

2. I/O Device - a physical unit (or portion of a unit) identified in the operating system by means of an Equipment Table (EQT) entry and a subchannel assignment.

Each I/O device is interfaced to the computer through an I/O controller that is associated with one of the computer I/O select codes. Controller Interrupts are directed to specific computer memory locations based on their select codes.

For details on the hardware I/O organization, consult the appropriate computer hardware reference manual.

Logical Unit Numbers

Logical Unit numbers (LUs) provide RTE users with the capability of logically addressing the physical devices defined by the Equipment Table. Logical Unit numbers (LUs) are used by executing programs to specify which I/O device requests are to be directed to. In an I/O EXEC call, the program simply specifies an LU and does not need to know which physical device or which I/O controller handles the transfer. This provides the user with system device independence.

An LU is associated with an EQT entry and a subchannel. Some I/O devices have EQT entries with one subchannel designation (i.e., line printers) and are referenced by a single LU number. Other devices (disc drives and CRT terminals) have EQT entries with several subchannel designations, with an LU assignment for each subchannel. When a user makes an I/O request specifying an LU, he can be addressing a total device (line printer) or a subsection of a device (left CTU of a terminal).

Logical Unit numbers are decimal integers that range from 1 to 254, LUs greater than 63 may only be accessed when operating under Session Monitor control. The functions of LUs 0 through 6 are predefined in the RTE-6/VM system as follows (could be system or session LUs):

0 - bit bucket (null device; no entry in Device Reference Table)

1 - system console.

2 - reserved for system (system disc subchannel).

3 - reserved for system (auxiliary disc subchannel).

4 - standard output device (left CTU of system console).

5 - standard input device (right CTU of system console).

6 - standard list device (line printer).

LU 8 is recommended for the magnetic tape device if one is present on the system. Peripheral disc subchannels must be assigned LUs greater than 6 and less than 64. If the Session Monitor is used, terminal LUs must be defined between 7 and 99. Additional logical units may be assigned for any function desired. On-line changes to existing LU assignments can be made by using the LU operator command described in the RTE-6/VM Terminal User's Reference Manual.

Power Fail

Power Fail is an optional hardware/software feature that saves all system status and context up to the point at which the computer signals a power failure. If generated into the system, the Power Fail routine performs the following steps:

1. When power fails, it saves all registers, stops DCPC transfers and saves maps. If not enough time was available, Power Fail issues a HLT 4.

2. When power comes on, it restarts the real-time clock, restores registers and maps, sets up a time-out entry (TO) back to its EQT entry, and then returns to the Power Fail interrupt location so that it can do more recovery work after the power fail system and operating system are re-enabled.

I/O Controller Time-Out

Each I/O controller may have a time-out clock to prevent indefinite I/O suspension. Indefinite I/O suspension can occur when a program initiates I/O and the device's controller fails to return a flag (possible hardware malfunction or improper program encoding). Without the controller time-out, the program that made the I/O call would remain in I/O suspension indefinitely, waiting for the "operation done" indication from the device's controller.

Privileged Interrupt Processing

RTE-6/VM allows interrupts from specified controllers to by-pass the standard system I/O processing modules and be processed by special routines. These I/O operations are therefore "privileged". Privileged interrupt processing is established for time-critical tasks such as power-fail processing or processing communication over a modem link.

I/O controller interrupts that are allowed to be processed as privileged are established at generation time. A special I/O card is placed in the backplane of the computer to physically separate the privileged interrupt controllers from the standard system processed controllers. The location of the "privileged-fence" card (if present) is stored in the System Base Page. Privileged controllers reside below the fence (greater priority) and non-privileged controllers reside above the fence (less priority).

When a privileged interrupt occurs, the Privileged Fence card holds off non-privileged interrupts. The system operates in the "hold-off-interrupt" (not interrupt disabled) state until the privileged interrupt has been processed.

The hold-off-interrupt state does not disable the interrupt system. It allows a higher priority privileged interrupt to interrupt a lower priority privileged interrupt. A non-privileged interrupt is not allowed to interrupt a privileged interrupt. For more information on privileged driver characteristics, see the RTE-6/VM Driver Writing Reference Manual.

Resource Management

The RTE-6/VM operating system allows cooperating programs to manage common system resources. A resource is defined to be any element within the RTE-6/VM environment that can be accessed by a user's program, e.g., an I/O device, a file, a program, an area of memory, or a subroutine. Cooperation between programs is established by coding them to take advantage of a utility subroutine (RNRQ) which allocates, deallocates, locks, and unlocks an arbitrary identification number known as a Resource Number (RN).

Within the cooperating programs, the RN is logically related to a particular resource by the statement stucture that comprises each program. When a program seeks exclusive access to a resource, it requests the system to lock the related RN. (The request is granted only if no other program has already locked the RN; otherwise the program is suspended until the RN is unlocked.) When it is finished with the resource, the program requests the system to unlock the RN so that other programs can lock it.

A RN is not physically assigned to any one resource. The logical association between the RN and a resource is accomplished only by the context of the statements within the program using the RN. The RN is known to the system but the resource with which it is associated is not, therefore all cooperating programs must agree on what RN is associated with what resource. The use of resource numbers is described in Chapter 5 of this manual.

Session Monitor

If the appropriate software modules (refer to the RTE-6/VM System Manager's Manual) are included at generation time, the RTE-6/VM operating system can be configured to provide controlled access to system services and resources by multiple users.

With Session Monitor configured into the system, the user is required to "log on" to a station (terminal) using an account ID assigned to him by the System Manager. At system initialization, the System Manager sets up an account file on disc which describes the I/O devices and the command capabilities assigned to each account ID. When a user has successfully logged on, a Session Control Block (SCB) is established for his "session" using information taken from the account file. The Log-on Processor provides the session user with a copy of FMGR, and each command entered is checked to verify that the user has the capability to use the command as specified in his SCB.

The I/O devices that the session user has access to must be defined in a section of his SCB known as the Session Switch Table (SST). The SST entries are taken from the session user's specific account file entry (LU's associated with the user's ID) and from a table in the account file common to all users known as the Configuration Table (LU's associated with each session station). The function of the SST is to link the session LUs on which a user makes an I/O request, to the system LUs that the I/O request will actually be directed to. When the user makes an I/O request, his SST is searched for the LU specified in the request. If the LU is found, it is switched to the associated system LU and the request is processed. If the requested LU is not found, an error message is returned, indicating that the LU is not defined for the user's session. The SST therefore defines the system I/O devices that the session user can access.

When operating in the session environment, access to disc cartridges is controlled by identifying them as belonging to a particular user or group of users. Disc cartridges can be mounted as:

* Private cartridges - allows Read/Write access only by the session user who mounted it and the System Manager.

* Group cartridges - allows Read/Write access only by members of the group that the cartridge is mounted to and the System Manager.

* System cartridges (LU 2 and LU 3) - allows Read/Write access by the System Manager and non-session programs, read-only access by session users. Can only be mounted or dismounted by System Manager.

* System cartridges (global) - allows Read/Write access by any system user. Can only be mounted or dismounted by System Manager.

Within the account file is a table (set up by System Manager) that indicates disc LUs that are available to session users (Spare Disc Pool). If a session user wishes to mount a spare cartridge, and has the capability to do so, a disc can be allocated from the pool (a "working" copy of the Free Disc Pool maintained in memory) and an entry is made in his SST indicating that he has access to that cartridge. The entry in the disc pool is also flagged indicating the disc is allocated.

Disc cartridges are mounted and dismounted via FMGR commands discussed in the RTE-6/VM Terminal User's Reference Manual. The formats of the session-related tables are shown in Appendix H. Refer to Chapter 3 for further information on disc cartridges.

Language Support

The languages available for user program development in the RTE-6/VM operating environment are briefly described below. For further information on these languages, refer to the appropriate reference manual.

* FORTRAN---a problem oriented programming language that is translated by a compiler. The FORTRAN compiler executes in RTE and accepts source programs from either an input device or a FMGR file. The resultant relocatable object programs and listed output files are stored in FMP files or output to specified devices. For further information, refer to the appropriate FORTRAN Programmer's Reference Manual.

* Pascal/1000---A top-down structured programming language that is translated by a compiler. The Pascal/1000 Compiler operates in a similiar manner as the FORTRAN compiler. For further information, refer to the Pascal/1000 Programmer's Reference Manual.

* Basic/1000C---a language subsystem for multi-user conversational program development, testing, debugging, execution, and compilation. Basic/1000C includes an interpreter with an editor and a multifunction debugger for fast, friendly conversational program development and a compiler for fast run-time performance and source security.

* REAL-TIME BASIC/1000D---an optional, conversational programming language that is easily learned, even by users without previous programming experience. Each statement entered by the user is immediately checked for correct syntax by the Real-Time BASIC Interpreter. No separate compilations or assembly operations are involved. A partly completed program can be run at any time to confirm that it executes as the user intended. For further information, refer to the Multi-User Real-Time BASIC Reference Manual.

* Macro/1000---a machine-oriented programming language. Source programs written in this language are accepted by the Macroassembler from either input devices or disc files and translated into absolute or relocatable object programs. Absolute code is output in binary records suitable for execution on HP CPUs. For further information, refer to the Macro/1000 Reference Manual.

* RTE Micro-Assembler---part of an optional support package for on-line users of special microprogrammed instructions. The Micro-Assembler translates source code into object microprograms. For further information, refer to the Micro-Assembler Reference Manual.

Executive Communication

EXEC calls are the line of communication between an executing program and system services. The required calls are coded into a program during its development phase. The calls have a structured format plus a number of parameter options that further define the specific operation to be performed.

The following is a partial list of system services available to an executing program via calls to the EXEC processor:

* Perform input and output operations
* Allocate and release disc space
* Terminate or suspend itself
* Load its segment
* Schedule other programs
* Recover scheduling strings
* Obtain the time of day
* Time-schedule program execution
* Obtain status information on partitions

Refer to Chapter 2 of this manual for complete descriptions and format considerations associated with EXEC calls.

File Management System

The File Management Package (FMP) allows the user to manipulate I/O devices and files. FMP can be used either interactively or programmatically. Interactively, using FMGR commands described in the RTE-6/VM Terminal User's Reference Manual or CI commands described in the RTE-6/VM CI Users Manual. Programmatically, using FMP calls described in Chapter 3 of this manual or for CI files, Chapter 6 in the CI Users Manual.

The FMP library contains routines that are called from user programs and used to manipulate disc and non-disc files (files which reference non-disc devices). Using calls to these routines, the user can create, access, purge, and obtain the status of files.

Files are classified according to the record format within the file and the type of data the system expects to find in each record. A file's type is defined when it is created and this information is placed in its file directory entry. When a file is accessed, this information is used by FMP to determine the files characteristics and initiate the appropriate action as specified by the type of file it is manipulating.

The user must also be aware of file types. Certain files are formatted to facilitate random access (fixed-length records), and others are formatted for sequential access (variable-length records). User-written programs should be coded to recognize and take advantage of a file's characteristics if efficient file manipulation is to be accomplished.

A file can contain up to (2**31)-1 records and can have a total size up to 32767 X 128 blocks (1 block = 256 bytes). For files with fixed-length records or variable-length records, the file size is defined at creation and is extended as needed.

The following is a summary list of the services available to user programs via FMP calls:

* Create Files (disc file only).
* Open files for specific modes of access.
* Read and write to files.
* Position to records within a file.
* Close files to access.
* Purge files from the system.
* Obtain position and status information on files.
* Rename files.
* Obtain disc cartridge list.

Additional features for CI files include:

* Longer file names and directories.
* Hierarchical directories.
* File type extensions.
* File ownership
* Error reporting and utility calls.

Refer to Chapter 3 of this manual and Chapter 6 of the CI Users Manual for complete description and format considerations associated with FMP calls.

System Library

The System Library, included with the RTE-6/VM Operating System, is a collection of relocatable routines that can be used to interface user programs with system services. Some of the important services available with RTE-6/VM are implemented by calling these routines. These System Library routines are described in Chapter 5. For example:

* VMA and EMA programming

* Resource management

* Parameter passage

In addition, the System Library contains various general purpose routines that perform services for the programmer. These services include:

* Re-entrant I/O processing

* Data conversion and string manipulation

* System status query

* Session environment related services

For more details on additional System Library routines, refer to the Relocatable Libraries Reference Manual RTE-A * RTE-6/VM (Part No. 92077-90037).

Spooling System

The Spooling System operates in conjunction with the File Management System to provide session input and output spooling, batch job processing with spooling or, through user program calls, to provide programmatic spooling without batch processing. Spooling means that jobs or data are placed on disc files for input, and data is sent to disc for output. This allows input and output to be performed independently of each other and of job processing. Spooling allows programs to be processed without having to wait for completion of input from or output to slow devices. The entire spool process can proceed automatically with virtually no user intervention, or it may be directly controlled during its various phases.

Spooling can be used to increase the throughput of a job stream or program that is limited by the idle time of slow peripheral devices. It does this by allowing programs to perform I/O to disc files rather than to the slower peripheral devices. The system then manages the I/O between the disc files and the peripheral devices to assure that all I/O reaches its proper destination.

The Spooling System provides the following capabilities:

* Opens and closes the disc files known as spool files; after close, optionally writes the file contents to a user-selected non-disc device for output.
* Keeps a record of the current status of all jobs and spool files in the system.
* Translates non-disc device references in program I/O calls to references to spool files.

For details on the interactive and programmatic services provided by the Spooling System, refer to the RTE-6/VM Batch and Spooling Reference Manual.

System Utility Programs

Standard system utilities are on-line programs that run under the RTE-6/VM Operating System and are called by the user to perform various program preparation, system status and housekeeping processes. The presence of any utility program in the system is optional, depending upon site-specific requirements.

The following brief descriptions of the utilities available with RTE-6/VM are intended to serve as an introduction. For more information, refer to the RTE-6/VM Utility Programs Reference Manual, the RTE-6/VM Loader Reference Manual, or the RTE-6/VM System Manager's Manual.

* Compile Utility (COMPL)---The compile program enables the user to invoke any of the HP-supported compilers or the Macroassembler. The utility will select the appropriate compiler or the Macroassembler by checking the control statement (first statement required in the program), the specified compiler option, or the compiler parameter specified in the COMPL run string. In addition, COMPL will automatically outspool the listing to the LU requested if spooling is enabled in the system.

* Compile and Load Utility (CLOAD)---CLOAD performs a composite function. It performs all the functions of COMPL and inputs the relocatable results to the to the loader (LOADR) for the creation of an executable memory-image program.

* MLS-LOC Loader (MLLDR)---The MLS-LOC Loader, MLLDR, allows the loading of very large programs on RTE-6/VM. MLLDR provides multilevel segmentation for large programs. Multilevel segmentation (MLS) allows as much code as required to be memory-resident. The MLS-LOC Loader accepts relocatable code for a large program from a command file, handles all program segment loading for the user, and produces an absolute load module that is ready for execution.

* MLLDR Command File Utilities (SGMTR, SXREF)---The SGMTR utility is used to assist in generating MLLDR Command Files necessary to load large programs using the MLS-LOC relocating loader (MLLDR). Another utility, SXREF, may be used to provide a cross reference information on program modules.

* Relocating Loader (LOADR)---The Relocating Loader program accepts relocatable programs and outputs absolute load modules in conformance with loader control parameter options specified by the user. Other command parameters cause the loader to purge unwanted permanently loaded programs from the system.

* LINDX is a utility for indexing a library file for LINK. It allows LINK to search a library or link a program faster than if it was not indexed.

* Relocating Linker (LINK)---LINK accepts relocatable programs and libraries, and outputs absolute type 6 program files.

* Indexed Relocatable Library Utility (INDXR)---The INDXR utility program is used to build an indexed relocatable library from user files (type 5), increasing the speed at which the MLS-LOC loader operates. When the MLS-LOC loader searches an indexed library for a particular module, it only needs to search a directory instead of the whole file.

* Interactive Editor (EDIT/1000)---The Interactive Editor (EDIT) is used to create and edit ASCII files. EDIT can operate in interactive mode (accepts commands from a keyboard device) batch mode (accepts commands from a job command file), or from a user command file spooled via the SL command.

* Interactive Debugger (DBUGR,MLSDB)---The DBUGR Subroutine can be appended to a user program by LOADR. The MLSDB Subroutine can be appended to a user program by the MLS-LOC Loader. These debugger subroutines can then aid the user in checking for logical errors in a program through interactive control commands. Debugging is performed at the Macro/1000 level. Refer to the Debug Subroutine Reference Manual for a complete description of all DBUGR and MLSDB functions.

* On-Line Generator (RT6GN)---The On-Line Generator permits use of an existing RTE-6/VM system to configure a new RTE-6/VM system according to user specifications. Generation can be directed from an answer file, logical unit or operator console.

* File Merge Utility (MERGE)---The Merge utility provides a simple way for files to be merged. Taking its input from a command file or interactively from a terminal, the Merge utility concatenates files to a destination file, creating a new file if none exists.

* File Backup Utility (FC)---The FC Utility program can be used to easily copy selected files between various media that includes discs and certain tape devices such as magnetic tapes, cartridge tape drives, integrated into CS80 discs and minicartridge tapes.

* The TF utility provides a means for copying files between disc and tape media, either disc-to-tape or tape-to-disc. Files can be copied to/from a 1600 BPI magnetic tape or a CS/80 cartridge tape (CTD). Refer to Chapter 4 of the RTE-6/VM Utilities Manual (Part No. 92084-90007) for more details.

* Disc Cartridge Save/Restore Utilities (WRITT/READT)---The disc cartridge Save/Restore utilities allow the user to save and restore peripheral disc cartridges through the use of magnetic tape. Saving disc cartridges on magnetic tape is accomplished through the WRITT program and restoring them to the system is accomplished through the READT program.

* Physical Disc Backup Utilities (PSAVE, PRSTR, PCOPY)---The Physical Disc Backup programs can be used either on-line or off-line to transfer data from disc to magnetic tape or vice-versa, copy data from disc to disc, verify successful transfer or copy operation, and to initialize a disc cartridge.

* System Status Program (WHZAT)---The WHZAT program provides status information regarding the current system environment. It will list active programs associated with the user's session or list all active programs and the current status of each program in the system. A list of programs having "father-son" relationships and a list of the status of every program partition in the system (occupied, non-occupied, or shared) can also be displayed. WHZAT can also provide a list of every program known to the system.

* Soft Key Programs (KEYS, KYDMP)---The KEYS and KYDMP programs are used to create user-defined command sets for programming the soft keys on the HP 2645A/48A Display Station. Soft keys provide the capability to enter entire sequences of commands with a single key stroke. The advantages are speed of entry and a significant reduction in operator errors during terminal entry sessions.

* Track Assignment Table Log Program (LGTAT)---The LGTAT program logs and displays the status of the system and auxiliary (only) disc tracks.

* Source File Comparison Utility (SCOM)---The SCOM Utility is used to compare two text files and print the similarities and differences between them. The SCOM utility can detect which words or lines have been inserted, deleted, or modified.

* On-Line Driver Replacement Utility (DRREL, DRRPL)---The purpose of the On-Line Replacement Utility is to make it possible for the user to replace a driver on-line instead of executing a new system generation. This utility is a useful tool for developing and debugging drivers. Refer to the RTE-6/VM Driver Writing Manual for a complete description of this utility.

* Help Utilities (HELP,GENIX,CMD)---The RTE-6/VM Help Utilities provide a means of generating help functions for any interactive program on the RTE operating system. The HELP and CMD programs allow the user to interactively request more information concerning any system error or interactive command in the system. The GENIX utility creates an index file used by the HELP and CMD programs. This allows users to add their own application errors or commands to an indexed file, which can then be searched by HELP or CMD programs.

Chapter 2
Executive Communication

Introduction

An executing program may request various system services via a call to the EXEC processor, specifying the request in a parameter string. Initiation of the call causes a memory protect violation interrupt (enables the System Map) and transfers control to the EXEC module. The EXEC module determines the type of request from the parameter string and initiates processing if the request was legally specified. A summary of the services available to the user via EXEC calls is shown in Table 2-1 in the order of their presentation. Table 2-2 lists the EXEC calls in numerical order.

Table 2-1. Summary of EXEC Calls

STANDARD I/O EXEC CALLS	
EXEC Call#	Description
1	Read from I/O device.
2	Write to I/O device.
3	Control Operation on I/O device.
CLASS I/O EXEC CALLS	
17	Class I/O Read from I/O device.
18	Class I/O Write to I/O device.
20	Class I/O Write/Read to/from I/O device.
19	Class I/O Control on I/O device.
21	Class I/O Get (completion of a Class I/O transfer)

Table 2-1. Summary of EXEC Calls (Cont.)

PROGRAM MANAGEMENT EXEC CALLS	
EXEC Call#	Description
6	Terminate the calling program.
7	Suspend execution of the calling program.
8	Load the calling program segment.
9	Immediate schedule of program with wait.
10	Immediate schedule of program without wait.
23	Queue schedule of program with wait.
24	Queue schedule of program without wait.
12	Time schedule a program.
14	Pass string between programs.
22	Control program swapping.
SYSTEM STATUS EXEC CALLS	
11	Request the current time.
13	Request status and type of an I/O device.
25	Request status about a specified memory partition.
26	Give the memory limits of the calling programs partition.
DISC TRACK MANAGEMENT EXEC CALLS	
4	Allocate disc tracks as local to a program.
15	Allocate disc tracks as global.
5	Release disc tracks which were locally allocated.
16	Release disc tracks which were globally allocated.

Table 2-2. Index of EXEC Calls

EXEC Call#	Description
1	I/O read
2	I/O write
3	I/O control
4	Track allocation (local)
5	Track release (local)
6	Program completion
7	Program suspend
8	Program segment load
9	Program schedule-Immediate with wait
10	Program schedule-Immediate without wait
11	Time request
12	Timed program execution
13	I/O status
14	String passage
15	Track allocation (global)
16	Track release (global)
17	Class I/O Read
18	Class I/O Write
19	Class I/O Control
20	Class I/O Write/Read
21	Class I/O Get
22	Program swap control
23	Program schedule-queue with wait
24	Program schedule-queue without wait
25	Memory partition status
26	Memory size request

EXEC Call Formats

EXEC calls can be made from FORTRAN, Pascal/1000, or Macro/1000 program. In FORTRAN, EXEC calls are coded as either function or subroutine call statements. In Pascal/1000, EXEC calls are coded as either a procedure or a function. In Macro/1000, EXEC calls are coded as JSB EXEC, followed by a series of parameter definitions. For any FORTRAN or Pascal/1000 EXEC call statement, the object code generated is equivalent to the corresponding Macro/1000 object code.

The discussion that follows shows the general formats used to code EXEC calls into FORTRAN, Pascal/1000, and Macro/1000 programs. After each general format is an example of its use. Each example shows the same action being initiated, e.g., reading 10 words from LU 5 and placing them into the first 10 words of a 100 word buffer.

FORTRAN Subroutine Call Format:

```
 :
Pl          Array parameters are defined in DIMENSION
 :          or COMMON statements.
Pn          One-word parameters are defined as integer
 :          variables.
CALL EXEC(Pl,...,Pn)
 :
```

Example:

```
 :
DIMENSION IBUFF(100)   100 word buffer.
 :
ICODE=1                Request Code Word.  1=Read.
ICNWD=5                Control Word.  LU=5.
ILEN=10                10 words are to be read.
 :
CALL EXEC(ICODE,ICNWD,IBUFF,ILEN)
    or
CALL EXEC(1,5,IBUFF,10)
 :
```

In FORTRAN EXEC calls, one-word integer parameters can be defined as integer variables or actual integer values. Array parameters must be defined in a DIMENSION statement. Parameters are function and position dependent.

FORTRAN Function Call Format:

```
 :
P1        Array parameters are defined in DIMENSION or
 :        COMMON statements.
Pn        One-word parameters are defined as integer
 :        variables.
REG       Defined as real variable comprised of two
 :        integer variables.
 :
REG=EXEC(P1,...,Pn)
 :
```

Example:

```
 :
DIMENSION IBUFF(100),IAB(2)
EQUIVALENCE (REG,IAB(1))
 :
ICODE=1
ICNWD=5
ILEN=10
 :
REG=EXEC(ICODE,ICNWD,IBUFF,ILEN)
  or
REG=EXEC(1,5,IBUFF,10)
 :
```

The purpose for using the function call format is to obtain the contents of the A- and B-Registers after the EXEC call has been executed. The A- and B-Register contents must be returned as a real variable. In the example, the real variable REG has been equated to the integer variables IAB(1) and IAB(2), making the A-Register contents available in IAB(1) and the B-Register contents available in IAB(2). These two values can then be examined for error indications (refer to the section on EXEC Call Error Returns).

Pascal/1000 Procedure Call Format:

```
      :
     P1
      :
     PN
      :
PROCEDURE exec_name $ALIAS 'EXEC'$
   (    icode: typex;            All the parameters  in the
    VAR parm2: typex;            EXEC procedure  must  be a
     :                           previously  defined  type.
    VAR parmN: typex);           The parameters in the call
   EXTERNAL;                     to the EXEC procedure must
     :                           be defined  as  a  variable
    BEGIN                        or a constant.
     :
  exec_name(P1...PN);
     :
```

EXAMPLE:

```
     :
  CONST
     icode=1;
     lu=5;
     length=-10;
  TYPE
     buffer=PACKED ARRAY [1..200] OF CHAR;
     int=-32768..32767;
  VAR
     word_buffer:buffer;
      :
  PROCEDURE read_into_buffer $ALIAS 'EXEC'$
     (    icode:int;
          source:int;
      VAR destination:buffer;
          number_to_read:int);
    EXTERNAL;
      :
     BEGIN
       :
     read_into_buffer(icode,lu,word_buffer,length)
       :
```

Pascal/1000 EXEC calls usually use aliases for each EXEC service used in a program. This method is used to make the program more readable and allows for a different set of parameters for each EXEC call in the program.

Pascal/1000 Function Call Format:

```
    P1
     :
    PN
     :
 FUNCTION exec_name $ALIAS 'EXEC'$
     (    icode: typex;          All the parameters in the
      VAR parm2: typex;          EXEC function must be a
       :                         previously defined type.
      VAR parmN: typex):real;    The parameters in the call
    EXTERNAL;                    to the EXEC function must
     :                           be defined as a variable
   BEGIN                         or a constant.The function
     :                           is defined as an integer,
   value:=exec_name;             a real, or a previously
     :                           defined type.

EXAMPLE:

     :
  CONST
    icode=1;
    lu=5;
    length=-10;
  TYPE
    buffer=PACKED ARRAY [1..200] OF CHAR;
    int=-32768..32767;
    error:PACKED ARRAY [1..4] OF CHAR;
  VAR
    error_return:error;
    word_buffer:buffer;
     :
  FUNCTION read_into_buffer $ALIAS 'EXEC'$
    (   icode: int;
        source:int;
    VAR destination:buffer;
    number_to_read:int):error;
   EXTERNAL;
     :
   BEGIN
     :
error_return:=read_into_buffer(icode,lu,word_buffer,length);
     :
```

The purpose for using the function call format is to obtain the contents of the A and B-Registers after the EXEC call has been executed.

Macro/1000 Subroutine call:

```
        :
        EXT EXEC    Link calling program to EXEC module.
        :
        JSB EXEC    Transfer control to EXEC module.
        DEF RTN     Define return point from EXEC.
        DEF P1      Define address of first parameter.
        :
        DEF Pn      Define address of n-th parameter.
RTN     :           Continue execution of program.
        :
P1      define parameter value.
        :
Pn      define parameter value.
        :
```

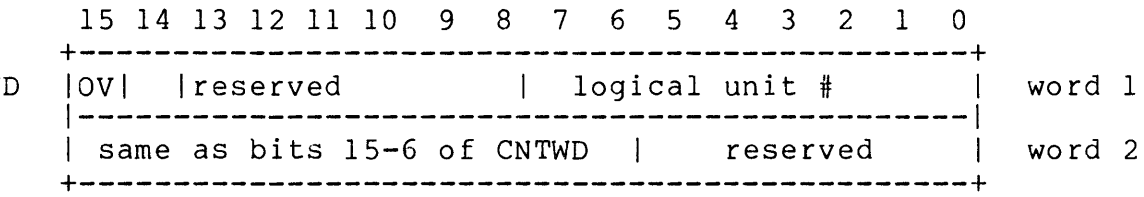
Example:

```
        :
        EXT EXEC
        :
        JSB EXEC
        DEF NSI
        DEF ICODE
        DEF ICNWD
        DEF IBUFF
        DEF ILEN
NSI     :
        :
ICODE   DEC 1       Request Code Word.  1 = Read.
ICNWD   DEC 5       Control Word.  LU = 5.
IBUFF   BSS 100     100 word buffer where data is placed.
ILEN    DEC 10      10 words are to be read.
        :
```

The parameters (P1-Pn) defined in a Macro/1000 EXEC call are position dependent and the number of parameters (n) is dependent on the function of the EXEC call. For example, the Read example shown above requires four parameter definitions (n = 4) and they must be defined in the order shown.

XLUEX I/O Extended Logical Unit EXEC

Standard EXEC calls provide access to logical unit numbers of 63 or less. RTE supports up to 255 logical units. The XLUEX (extended logical unit EXEC) subroutine allows the programmer access to all LUs within the session switch table by splitting the control word in an I/O EXEC request into a double word quantity, as follows:



If the OV bit is set, then LU mapping is overridden; thus, a write to LU 1 will go to system LU 1 (typically the system console), and not to the scheduling terminal.

All parameters in an XLUEX call are identical to those in a standard EXEC call, except for the CNTWD parameter. XLUEX calls are functionally equivalent to EXEC calls, unless the OV bit is set.

When programming in FORTRAN or PASCAL, it is suggested that the XLUEX CNTWD be defined as an array of two single-word integers, with the LU number in the first word.

XLUEX should be called with ICODEs of 1,2,3,13,17,18,19, or 20.

This example accepts input from a terminal at LU 68 using an XLUEX call with the echo bit set.

```
      PROGRAM XLTST
      IMPLICIT INTEGER (A-Z)
      DIMENSION BUFR (40), CNTWD (2)
       :
      BUFLEN=40
      ECODE=1 + 100000B
      CNTWD (1) = 68
      CNTWD (2) = 400B
      CALL XLUEX (ECODE,CNTWD,BUFR,BUFLEN,*100)
    C SUCCESSFUL READ
    C ERROR OCCURRED IN READ CALL
  100 CALL ABREG (A,B)
      WRITE (1,)("ERROR ON READ = ",2A2)A,B
      END
```

EXEC Call Error Returns

EXEC calls that contain errors will cause the offending program to be aborted if the error was severe. If an error was not severe, the program is aborted or, at the user's option, reports the error to the program and allows it to continue executing. Shown below is a partial summary of the EXEC errors. The errors marked with an "*" are considered severe and will always cause the program to be aborted. The error code is placed in the user's Session Control Block (SCB), words 5 through 8.

Error Code	Error Type
* MP	Memory Protect
* DM	Dynamic Mapping
* RQ	Request Code
* DP	Dispatching
* RE	Reentrancy
* PE	Parity
SC	Scheduling
LU	LU Lock
IO	Input/Output Error
DR	Disc Allocation
RN	Resource Number

A detailed explanation of EXEC call error messages is given in Appendix A.

No-Abort Option

If the user wants non-severe errors to be reported to the calling program and the program to continue executing, the return point from the EXEC call can be altered by setting the "no-abort" bit (bit 15) in the request code word (ICODE=ICODE+100000B). This causes the system to execute the first line of code (it must be a one-word instruction) following the EXEC call if an error occurs. When the no-abort bit is set and an error occurs, the error code is not placed in the user's SCB. If there is no error, the second line of code following the EXEC call is executed.

The following section from a sample FORTRAN program demonstrates the use of the altered error return points:

```
                        :
                    parameter definitions
                        :
                    ICODE=ICODE+100000B <--set no-abort bit.
                        :
                    CALL EXEC(ICODE,ICNWD,IBUFF,ILEN)
   error return-->      GO TO 100
no error return--> 10   :
                        :
                100     error processing
                        :
```

NOTE

The statement to be executed on a "no error" return must have a statement number or the FORTRAN compiler issues a warning message.

In FORTRAN, only the GO TO statement should be placed after an EXEC call with the no-abort bit set; any other command would cause error information to be lost. The GO TO statement must not refer to the next statement, i.e.,

```
                :
            CALL EXEC(ICODE+100000B,ICNWD,IBUFF,ILEN)
            GO TO 100
        100  :
             :
```

This is illegal because the FORTRAN compiler tries to optimize the two statements and will not produce a jump if the jump destination is the next executable statement; the GO TO would be ignored.

In FORTRAN, an alternate return label may be specified in the EXEC call by an asterisk and a statement label. When the no-abort bit is set and an error occurs, the alternate return statement is executed. When using an alternate return label, the first line of code following the EXEC call is executed when no error occurs. In this case, the GOTO statement does not need to be the statement following the EXEC call.

```
                        :
                        :
                    parameter definitions
                        :
                    ICODE=ICODE + 100000B <--set no-abort bit
                        :
                    CALL EXEC (ICODE,ICNWD,IBUFF,ILEN,*100)
no error return-->      :
                        :
error return--> 100 error processing
```

As discussed previously, if a non-severe error return is made to a program, the A- and B-Registers contain the ASCII error codes. The A-Register contains the error type (SC, LU, IO, DR, RN), and the B-Register contains the error number (ASCII 01, 02, 03, etc.). Note that the no-abort error return will return control to the calling program when a parity error (PE) occurs, if the error is on the system or auxiliary disc (LU 2 or LU 3). In this case, the B-Register will be set to -1. If a parity error occurs on a disc other than LU 2 or 3, the error is considered "severe" and the calling program will be aborted.

In a Pascal program, the first statement after an EXEC procedure call with the no-abort bit set must be a one-word instruction. There are two statements in Pascal in this category:

- GOTO statement.
- Procedure statement of a parameterless procedure declared with the $DIRECT compiler option.

The following example demonstrates the use of the no-abort bit in a Pascal program. Refer to the Pascal/1000 Reference Manual for future details.

```
                    :
                parameter definitions
                    :
                EXEC7:= 7 + no-abort bit;
                    :
                PROCEDURE exec_error;
                  $RECURSIVE OFF,DIRECT$
                    :
                  process error
                    :
                BEGIN
                    :
                  suspend (EXEC7);
                  exec_error;
                    :
```

In Macro/1000, the A- and B-Registers can be manipulated directly to obtain error information. In FORTRAN or Pascal, an HP-supplied subroutine (ABREG) can be used to obtain the contents of the A- and B-Registers. The ABREG subroutine is used with an EXEC call; the use of the EXEC function call makes ABREG unnecessary. The use of ABREG is shown in the following example:

```
                    :
                parameter definitions
                    :
                CALL EXEC(ICODE+100000B,...) <--set no-abort
   error return->    GO TO 100                    bit.
no error return-> 10     :
                         :
                100   CALL ABREG(IA,IB)
                      error processing
                         :
```

After the return from ABREG, IA contains the contents of the A-Register and IB contains the contents of the B-Register. The calling program can then implement error processing accordingly.

Note that if an EXEC call is successful, the A- and B-Registers can contain pertinent information about the status of the call. ABREG can be used to retreive this information also. The contents of the A- and B-Registers after a successful call are discussed with the descriptions of the individual EXEC calls later in this section.

ABREG is described in the RTE-6/VM Relocatable Library Manual.

NOTE

DO NOT set the no-abort bit in an EXEC function call. For example, DO NOT USE the following technique:

```
      :
     REG=EXEC(ICODE+100000B,...)
     GO TO 100
      :
      :
100  error processing
      :
```

The reason for not using this method is that when the function call is compiled, a double-word store instruction is generated to save the contents of the A- and B-Registers. This instruction is placed directly after the EXEC function call. If the EXEC call is successful, control is returned to the second word after the call. This would be the second half of a double word instruction and would therefore be erroneous.

No-Suspend I/O Option

Certain programs in a real-time environment may be considered too important to be suspended due to an downed I/O device. This capability is provided by the "no-suspend" I/O option. Programs can check device status before executing I/O requests. However, if the device goes down due to (or during) the I/O request, the calling program will be suspended unless the no-suspend bit (bit 14) in the I/O request code word (ICODE=ICODE+40000B) is set. The no-suspend bit can be used for the following I/O EXEC requests:

Standard I/O EXEC calls: 1,2,3

Class I/O EXEC calls: 17,18,19,20

When the no-suspend bit is set, the calling program resumes execution at the first line of code (it must be a one-word instruction) following the EXEC call if an error occurs. If there is no error, the second line of code following the EXEC call is executed. The use of the altered error return points, and the A- and B-Registers returns are the same for the no-suspend I/O option and the no-abort option. The same restrictions on the use of the no-abort bit hold for the no-suspend bit. The alternate return label can also be used for the no-suspend option.

The following section from a sample FORTRAN program demonstrates the use of the no-suspend bit:

```
                    :
               parameter definitions
                    :
               ICODE=ICODE+40000B <--set no-suspend bit.
                    :
                    CALL EXEC (ICODE,ICNWD,IBUFF,ILEN)
error return------>    GO TO 100
no error return--> 10    :
                    :
                100 error processing
                    :
```

Table 2-3 outlines the various actions taken by the system depending on the condition of the system at the time of the I/O request when the no-suspend bit is set.

Table 2-3. Action Taken When the No-Suspend Bit is Set

BUFFERED I/O REQUESTS		
System Condition at request time.	Action Taken	A- + B-Registers
Requested EQT or LU is LOCKED.	Program resumes at first line after call.	IO13
Requested EQT or LU is DOWN.	Program resumes at first line after call.	IO14
No buffer memory (Standard I/O EXEC calls 1,2,3 only).	Request continues to be processed unbuffered.	meaningless
UNBUFFERED I/O REQUESTS		
External Condition	Action Taken	A- + B-Registers
Device goes down during I/O processing.	Program resumes at first line after call.	IO14
Operator downs the device using the "DN" command.	Program resumes at first line after call.	IO14

EXEC Description Conventions

In the subsections that follow, certain conventions are used to describe EXEC calls. These conventions are summarized below.

* Parameters that are underlined, such as:

```
CALL EXEC(IP1,IP2,IP3)
          -------
```

  have values returned by the system, i.e., the value is not supplied by the user.

* Parameters that are double underlined, such as:

```
CALL EXEC(IP1,IP2,IP3)
          ===
```

  have values that are returned by the system in some cases and user-supplied in other cases. The comments associated with the call description should be consulted for details concerning the use of these parameters.

* Parameters enclosed in square brackets, such as:

```
CALL EXEC(IP1,IP2[,IP3])
```

  are optional.

* Parameters enclosed in angle brackets, such as:

```
CALL EXEC(IP1,IP2<,IP3>)
```

  are optional in some cases and required in others. The comments associated with the call description should be consulted for details concerning their use.

* Parameters with no qualifiers, i.e., square brackets, angle brackets, or underlines, are required and their value is supplied by the user.

* If an optional parameter is to be used, but the preceding optional parameters are not, dummy variables must be used as place holders for the unused parameters.

* Note that a VMA/EMA variable cannot be used as a parameter in an EXEC call.

* All EXEC call descriptions in this section use the FORTRAN subroutine call format. If desired, the description of EXEC call general formats included at the beginning of this section can be consulted to convert the calls to the FORTRAN function call, Pascal/1000 procedure or function calls, or Macro/1000 subroutine calls.

* FORTRAN 77 character variables cannot be passed to an EXEC call. Refer to the FORTRAN 77 Reference Manual for more information on character data.

Standard I/O EXEC Calls

Standard I/O EXEC calls are used to transfer data to or from external I/O devices in addition to performing various I/O control operations. For output or control requests to unbuffered devices, the calling program is I/O suspended (state 2) until the operation completes. All input requests cause the calling program to be placed in state 2 until completion. Note that a device is specified as buffered or non-buffered at generation time (refer to RTE-6/VM On-Line Generator Manual) or by the EQ system command (refer to the RTE-6/VM Terminal User's Reference Manual).

The EXEC calls included in this section are listed below in the order of their presentation:

* EXEC 1 or 2 - Read or Write

* EXEC 3 - I/O Control

Note that the I/O operations performed by EXEC 1, 2 and 3 calls can also be accomplished on a no-wait basis by using the Class I/O calls described in this chapter or on a buffered basis by using the library subroutine REIO described in the RTE-6/VM Relocatable Library Manual.

The size of the buffer used in the I/O call is limited by the amount of memory that can be mapped to hold the buffer. The maximum buffer size can be calculated as follows:

size = 32767 - start address of SSGA

The system uses the area before the start of the SSGA to map a base page, the driver, and any I/O tables needed by the driver.

Note that if the buffer does not start on a page boundary, the buffer size can be reduced by up to 1 page because the start page must be mapped even though it is not completely used.

Read or Write Call — EXEC 1 or 2

Transfers information to (write) or from (read) a disc or non-disc I/O device.

```
CALL EXEC(ICODE,ICNWD,IBUFF,ILEN[,IOP1[,IOP2]])
                      =====
```

ICODE - Request code. 1=read, 2=write.

ICNWD - Control word. Specifies the LU of the device involved in the data transfer, driver dependent information, and optional parameter (IOP1 & IOP2) consideration.

IBUFF - Data buffer. For read operations (ICODE=1), the array into which the system returns data. For write operations (ICODE = 2), the array into which the program places data to be written.

ILEN - Data length. Positive number of words or negative number of characters to be read or written.

IOP1 - Track number for disc transfers; optional parameter for non-disc transfers.

IOP2 - Sector number for disc transfers; optional parameter for non-disc transfers.

COMMENTS:

REQUEST CODE (ICODE)---The no-suspend option and the no-abort option can be specified for an EXEC 1 or 2 call. Refer to the section on the EXEC call error returns for a detailed explanation.

CONTROL WORD (ICNWD)---The format of the control word is as follows:

```
   (UB)     (Z)     (X) (A) (K) (V) (M)
 ---------------------------------------------------------------
  XX|14|XX| 12 |XX| 10| 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |0|
 ---------------------------------------------------------------
                  |<--Function Code-->|<----Logical Unit--->|
```

The function of the control word fields are summarized below:

* Logical Unit (bit 0-5). Session logical unit number (LU) of the device to/from which data is to be transferred (less than 64). This can be the LU of a disc or non-disc device. If the LU is specified as zero, the call is executed, but no data is transferred. If LU specifies a disc device, IOP1 and IOP2 are required.

* Function Code (bit 6-10). Specifies control information for the driver module associated with the I/O device involved in the data transfer. This information is driver dependent and the user should refer to the appropriate driver reference manual for more information. The function code field can be defaulted to zero if no special driver action is desired. See Table 2-4 for a list of possible function codes. Some example values and corresponding meanings are:

  DVA05 Bit 8=1 EXEC1 (Read from 264X Terminal-ECHO on.)

  DVR12 Bit 7=0 EXEC2 (Write to 2607 Line Printer-first character of output is used for line control.)

* Z-bit (bit 12). If the Z-bit is set ("1"), a control buffer containing additional information is passed to the driver. The control buffer is defined by IOP1 and IOP2. This technique is only used for transfers to non-disc devices.

  The Z-bit is clear ("0"), if a control buffer is not being passed to the driver or the transfer is to a disc device. This is the default condition.

* UB-bit (bit 14). If the UB-bit is set ("1"), the I/O operation is forced to be unbuffered, even if the I/O device is a buffered device. This is useful if I/O status is desired after a transfer or for very large (greater than or equal to 1K words) I/O requests.

* Bit 11, 13 and 15. These bits are used by the system and should be set to zero by the user.

DATA LENGTH (ILEN)---This variable defines the length of the data record to be transferred. A positive number is used to specify the length of the data record in words. A negative number is used to specify the length of the data record in characters (bytes). If the data record contains REAL data, this must be allowed for in the ILEN parameter. For example, a data record containing 10 REAL values would require ILEN to be +20 or -40 since each REAL value requires two words. If the data record contains double-precision data, three or four words are required for each value, depending on the option taken for the FORTRAN Compiler at generation time.

OPTIONAL PARAMETERS (IOP1 and IOP2)---These two parameters are required or optional depending on the state of the Z-bit, as described above. If the Z-bit is set ("1"), IOP1 specifies the address of a control buffer and IOP2 specifies the length of the buffer (positive number of words or negative number of characters). The contents of the control buffer is driver dependent. Refer to the appropriate Driver Reference Manual for more detailed information.

If the Z-bit is clear ("0") and the LU specifies a disc device, IOP1 contains the track number and IOP2 contains the sector number to be used for the transfer. For CS80 discs, IOP1 contains the track number or high word of block and IOP2 contains the sector number or the low word of block. Refer to the DVM33 Disc Driver Reference Manual for further information.

If an I/O request to the disc is made with the specified track number greater than the size of the subchannel, or if the track number is equal to -1, the disc driver sets the B-Register to the number of tracks on the subchannel. If the request was a read, the disc driver also returns the number of 64-word sectors per track in the first word of the buffer.

If the specified track number is negative (not equal -1) an IO07 error is issued.

If the Z-bit is clear and the LU specifies a non-disc device, IOP1 and IOP2 are not used.

A- AND B-REGISTER RETURNS---End-of-operation information for successful read and unbuffered write operations is transmitted to the calling program via the A- and B-Registers. These register returns are summarized below:

* A-Register <-- word 5 (status word) of device's EQT entry (refer to the Table 2-8 for the EQT word 5 format).

* B-Register <-- positive number indicating number of words transferred (if ILEN in call was positive), or positive number of characters (bytes) transferred (if ILEN in call was negative).

* A- and B-Register returns are meaningless for write requests to buffered devices.

For unsuccessful operations, error information is returned in the A- and B-Registers. "EXEC Call Error Returns", included earlier in this section, describes the considerations associated with error returns.

An EXEC status request may be used after the call to ABREG to check the values of the A- and B-Registers.

I/O AND SWAPPING---Disc-resident programs performing I/O are swappable under any one of the following conditions:

A. The data buffer is not in the calling program's partition, i.e., it is in system COMMON.

B. The input or output buffer is completely contained in the Temporary Data Block (TDB) of a reentrant routine, and enough SAM was allocated to hold the TDB. This is the case if the I/O was done via the system library routine REIO (refer to the RTE-6/VM Relocatable Library Manual).

C. The device is buffered, the request is for output, and enough SAM was allocated for buffering the data record to be transferred.

D. The I/O request is a Class I/O request.

Only the first buffer of a double-buffer request (Z-bit set) is checked to determine swappability. It is the user's responsiblity to put the second buffer in an area that implies swappability (COMMON, etc.) if condition "A" or "B" is true. The system handles condition "C" and "D". Programs that request input from interactive devices (i.e., terminal), should use the subroutine REIO instead of an EXEC 1 call to assure program swappability and improve system performance.

EXAMPLE---Write a message to the terminal (LU 1) which prompts the operator to input his name. Read the response entered by the operator. Check for EXEC errors.

```
      PROGRAM OPNAM
      DIMENSION IBUF1(5),IBUF2(10)
      DATA IBUF1/10HENTER_NAME/
       .
       .
      ICODE=2+140000B              !WRITE REQUEST TO LU 1 WITH THE
      ICNWD=1                      !NO-ABORT  AND  THE  NO-SUSPEND
      ILEN=5                                          !BITS SET.
      CALL EXEC(ICODE,ICNWD,IBUF1,ILEN,*100)
   10 CONTINUE
       .
       .
      CALL EXEC(1+140000B,1,IBUF2,-20,*100)!READ REQUEST FROM LU 1
   20 CONTINUE                             !WITH THE NO-ABORT  AND
       .                                   !NO-SUSPEND BITS SET.
       .
  100 CALL ABREG(IA,IB)
      error processing
       .
       .
      END
```

I/O Control Call — EXEC 3

Initiates various I/O control operations (backspace, write end-of-file, rewind, etc.) on specified LU. Function is dependent on device to be controlled.

```
CALL EXEC(ICODE,ICNWD[,IOP1])
```

ICODE - Request code. 3 = control.

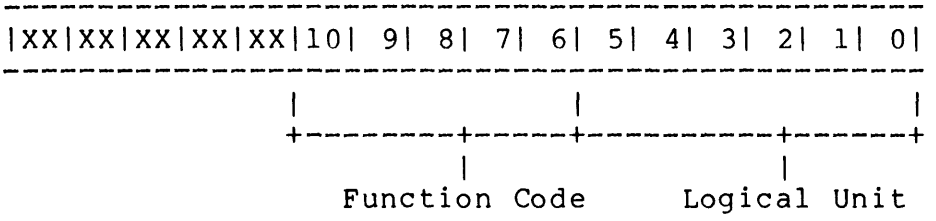
ICNWD - Control word. Specifies the LU and the control action to be initiated on that LU.

IOP1 - Optional parameter depending on the function code specified.

COMMENTS:

REQUEST CODE (ICODE)---The no-suspend option and the no-abort option can be specified for an EXEC 3. Refer to the section on the EXEC call error returns for a detailed explanation.

CONTROL WORD (ICNWD)---The format of the control word is as follows:



The function of the control word fields are summarized below:

* Logical Unit (bit 0-5). Session LU of the device that control action is to be initiated on.

* Function Code (bit 6-10). Specifies the type of control action to be initiated. Shown in Table 2-4 is a partial summary of the function codes associated with various devices and drivers. For more details, the user should refer to the driver reference manual associated with the device to be controlled.

Table 2-4. Function Codes

FUNCTION CODE (OCTAL)	ACTION
00	Clear device (U)
01	Write end-of-file (MT,CTU)
02	Backspace one record (MT,CTU)
03	Forward space one record (MT,CTU)
04	Rewind (MT,CTU)
05	Rewind standby (MT), Rewind (CTU)
06	Dynamic Status (MT,CTU)
11	List output line spacing, requires use of IOP1. (LP)
12	Write inter-record gap (MT)
13	Forward space one file (MT, CTU)
14	Backspace one file (MT, CTU)
15	Conditional form feed (LP)
20	Enable Terminal (CRT)
21	Disable Terminal (CRT)
22	Set time out, requires IOP1 (CRT)
26	Write end-of-data (CTU)
27	Locate file, requires IOP1 (CTU)
U = Universal MT = Magnetic Tape Drive CTU = Terminal Cartridge Tape Unit LP = Line Printer CRT = 26XX Terminal	
For CS80 Cartridge Tape Drives refer to the DVM33 Driver Manual for possible function codes.	

OPTIONAL PARAMETER (IOP1)---IOP1 is optional or required depending on the function code used. For example, if the function code is 11 (octal); indicating line spacing for the line printer, IOP1 would contain the number of lines to be spaced. If the function code is 27 (octal), IOP1 would contain the number of a file to be located on the CTU. For more details on the optional parameter, the user should refer to the driver reference manual associated with the device being controlled.

A- AND B-REGISTER RETURNS---For successful I/O control calls, the values returned in the A- and B-Registers are as follows:

* A-Register <-- If call was to an unbuffered device, word 5 (status word) of devices' EQT entry is returned (refer to Table 2-7 for the EQT format). If the call was to a buffered device, return is meaningless.

* B-Register <-- meaningless.

Register returns for unsuccessful call contain error information (refer to the section "EXEC Call Error Returns").

EXAMPLE---Cause line printer (LU 6) to space five lines.

```
      .
      .
      .
ICODE=3
ICNWD=6+1100B
IOP1=5
CALL EXEC(ICODE,ICNWD,IOP1)
      .
      .
      .
```

Class I/O EXEC Calls

The Class I/O feature of RTE-6/VM is implemented by a special set of EXEC I/O and System Library calls. These Class I/O calls provide user programs with I/O and communication capabilities not available with the standard I/O EXEC calls discussed earlier in this section. The features provided by Class I/O are summarized below:

* I/O without wait - allows a program to continue executing concurrently with its own input operation (Class Read) or output operation to an unbuffered device (Class Write).

* Mailbox I/O - allows cooperating programs to communicate via a controlled access data buffer.

* Data passage synchronization - prevents communicating programs from processing incomplete or non-updated data. A program can suspend itself until it receives a signal indicating that valid data is available from another program.

* I/O control without wait - allows a program to initiate a control operation on an I/O device and continue executing without waiting for the control operation to complete.

Class I/O is based on the use of a buffer in System Available Memory (SAM) with an associated "access key". The access key is known as a class number. Definitions of more terms unique to Class I/O are given in Table 2-5.

Note that the use of Class I/O is exclusive of system or local COMMON used in standard program-to-program communication.

Class I/O can be considered "double-call" I/O. One call is necessary to initiate the operation, and another call is necessary to complete the operation. The initiation call (Class Read, Write, Write/Read, or Control) places request parameters, plus data if required, in the class buffer in SAM. A buffer in SAM is allocated each time a Class I/O operation is initiated. The completion call (Class Get) retrieves the data and optionally releases the request. The class number must be used as a parameter in the Class Get call, thereby insuring that only authorized programs (programs "knowing" the correct class number) can access the buffer. If a program other than the program that initiated the I/O operation wishes to retrieve the results, the class number must be made available to the retrieving program (via COMMON, in a command string, or passed in an EXEC call itself). Note that once a Class I/O operation is initiated, the calling program has the option of either continuing with its execution or waiting for the operation to complete. When the operation is completed (using the Class Get call), the buffer can be released or retained at the users option.

Table 2-5. Class I/O Terms

Class Number	The "access key" used to access class buffers on this class number. It is originally allocated using a request to the CLRQ subroutine.
Class	The class number and the buffer are collectively known as a class.
Class Request	An access to a Logical Unit number or Mailbox I/O with a class number.
Pending Class Requests	The set of uncompleted Class I/O buffers referencing the class number and queued on I/O devices.
Completed Class Queue	The set of all completed Class I/O buffers queued on the specified class number. The structure is first-in, first-out.

A class number is allocated when a program issues a call to the CLRQ library subroutine or an EXEC (17,18,19, or 20) and sets the ICLAS parameter to zero. The preferred method to allocate a class number is a CLRQ request. (Refer to Chapter 5 for a description of this routine.) This routine allows the assignment of ownership to a class number so that in the event of a program terminating or aborting without cleaning up the classes assigned to it, the system will be able to deallocate those resources. However, when the user program issues an EXEC call to allocate the class number, the class remains allocated until a program deallocates it. The class should always be deallocated when it is no longer needed; freeing it for use by other programs. The maximum number of classes (1-255) is established at generation time. (Refer to the RTE-6/VM On-Line Generator Reference Manual.) Note that a program can have more than one class number allocated to itself.

When a Class I/O request is made (i.e., Read, Write, etc.) it is associated with the specified class number and queued on the I/O device. This is known as the "pending class request". The request remains pending until the driver has received the request and processed it accordingly.

When the driver has completed the specified operation, the request is linked to the "Completed Class Queue" associated with the class number. The results of the operation are then available to the calling program (or another program) via a Class Get call. Note that the queueing technique (pending and complete) allows more than one buffer to be associated with the same class number, i.e., a program can make multiple requests specifying the same class number.

For "I/O without wait" operations, data can be read from, or written to an I/O device by first transferring the data to the buffer in SAM (Class Read, Write or Write/Read). The user program can either continue execution of other tasks without waiting for the I/O transfer to complete, or can suspend or terminate itself (releasing system services to other waiting programs) until the data transfer is completed. A user program recovers the results of a Class I/O call by issuing a Class Get call. If the results are not present, the caller either can wait or return to execute more code before reissuing the Class Get call.

A simple example of I/O without wait would be a program that issues a Class Read call in its code, followed by a series of other coded operations. While these following operations were being executed, the system simultaneously would be reading the data into the allocated class buffer. The calling program (or another program) would issue a Class Get call to fetch the data from the buffer. Examples of each Class I/O EXEC call is given at the end of this section. A more detailed example of I/O without wait is given in Chapter 5.

Class I/O Operation

The system handles a Class I/O call in the following manner:

a. When a program issues a Class I/O request (and the call is received), the system allocates a buffer from SAM and puts the request parameters in the header (first eight words) of the buffer. The buffer is then queued (pending) on the EQT entry specified by the LU (the "I/O Request List"). The system then returns control to the calling program.

b. If this is the only request pending on the EQT, the driver is called immediately; otherwise, the system returns control to the calling program and queues the request on the EQT according to the calling program's priority.

c. If buffer space is not available in SAM, the calling program is memory suspended unless bit 15 (no-wait) is set in the ICLAS parameter. If the no-wait bit is set, control is returned to the calling program with the A-Register containing a -2, indicating no memory available. If the program is suspended, no memory will be granted to lower priority programs until this program's Class I/O request is satisfied.

d. If too much memory was asked for (more than all of SAM) the program is aborted with an IO04 error return, unless the no-abort bit is set in the ICODE parameter.

e. If a class number is not available, the calling program is placed in the general wait list (status = 3) until the condition changes. If the no-wait bit is set, the program is not suspended and the A-Register will contain a "-1".

f. If the device is down, the calling program may be suspended unless the no-suspend bit (bit 14) in the I/O request code word (ICODE=ICODE+40000B) is set. Refer to the section on the EXEC Call Error Returns for a detailed explanation of the A- and B-Registers.

g. If the call is successful, the A-Register will contain zero on return to the program.

The buffer area furnished by the system is filled with the caller's data if the request is either a Class Write, or a Write/Read call.

After the driver receives the Class I/O request (in the form of a standard I/O call) and completes, the system will:

a. Release the data buffer portion of the request if it is a Class Write and the save bit in the ICLAS parameter was not set. The header is retained for the Class Get call.

   If the request is a Class Write/Read the data buffer is not released at this time.

b. Queue the header portion of the buffer (plus data if required) in the Completed Class Queue.

c. If a Class Get call is pending on the class number, reschedule the calling program. (This means that the user issued a Class Get call before the driver completed the request.) Note that the program that issued the Class I/O request and the program that issued the Class Get call do not have to be the same program.

d. If there is no Class Get call outstanding, the system continues and the driver is free for other calls.

When the user issues the Class Get call, the Completed Class Queue is checked and only one of the following paths is taken:

a. If the Completed Class Queue is not empty, the data in the class buffer (if applicable) is returned. The calling program has the option of leaving the I/O request in the Completed Class Queue so as not to lose the data (a subsequent Class Get call will obtain the same data), or dequeuing the request and releasing the header, buffer, and the class number back to the system.

b. If the Completed Class Queue is empty, (i.e., a Class Get call is issued before the Class I/O operation is completed), the calling program is suspended in the general wait list (status = 3) and a marker so stating is entered in the Completed Class Queue header. If desired, the program can set the no-wait bit to avoid suspension. In any case, when a completed request is queued on the class number, any program waiting in the general wait list for this class is automatically rescheduled. Note that only one program can be waiting for any given class at any instant. If a second program attempts a Class Get call on the same class number before the first one has been satisfied it will be aborted (I/O error IO10). The programs involved can avoid being aborted by setting the no-abort bit in the ICODE parameter of the Class Get call.

The Class I/O EXEC calls are listed below in the order of their presentation:

* Class Read, Write and Write/Read - EXEC 17, 18, and 20.

* Class I/O Control - EXEC 19

* Class Get - EXEC 21

Class Read, Write, and Write/Read

Initiates the transfer of information to/from a non-disc I/O device, or to another program.

```
CALL EXEC(ICODE,ICNWD,IBUFF,ILEN,[IOP1][,IOP2],ICLAS)
                      -----
```

ICODE - Request code. 17=Read, 18=Write, 20=Write/Read.

ICNWD - Control word. Specifies the LU of device involved in data transfer, driver-dependent information and optional parameter (IOP1 and IOP2) considerations.

IBUFF - Data buffer. For Read operations, a dummy parameter. For Write and Write/Read operations, the array where the program places the data to be written.

ILEN - Data length. Positive number of words or negative number of characters to be read or written.

IOP1 - Optional buffer address (refer to Comments below).

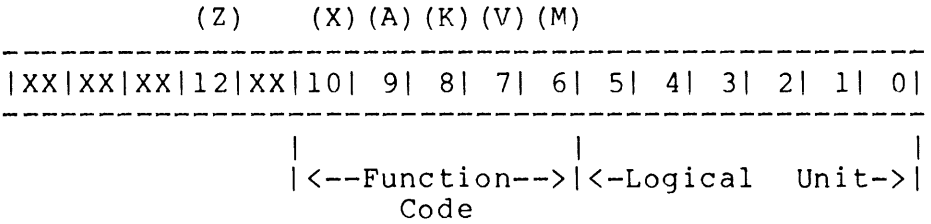
IOP2 - Optional buffer length.

ICLAS - Class word (refer to Comments below).

COMMENTS:

REQUEST CODE (ICODE)---The no-suspend option can be specified for an EXEC 17, 18, and 20 call to avoid being suspended when a device is downed due to the Class I/O request. For example, if a Class Write request with the no-suspend bit set is made to a line printer which was left off-line, the program will not be suspended. Refer to the section on the EXEC Call Error Returns for a detailed explanation.

CONTROL WORD (ICNWD)---The format of the control word is as follows:



The function of the control word fields are summarized below:

* Logical Unit (bit 0-5). Session LU of the device that data is to be transferred to or from. This must be an LU of a non-disc device. If the LU is set to zero, the call is executed but no data is transferred to/from an external device.

* Function Code (bit 6-10). Specifies control information for the driver module associated with the I/O device involved in the data transfer. This information is driver dependent and the appropriate driver reference manual should be consulted for more information. The function code can be defaulted to zero if no special action is required. Refer to Table 2-4 for possible function codes.

* Z-bit (bit 12). The Z-bit is set to "1" if an additional control buffer is to be passed to the driver module or passed to the program that performs the Class Get call. The control buffer is defined by the optional parameters IOP1 and IOP2.

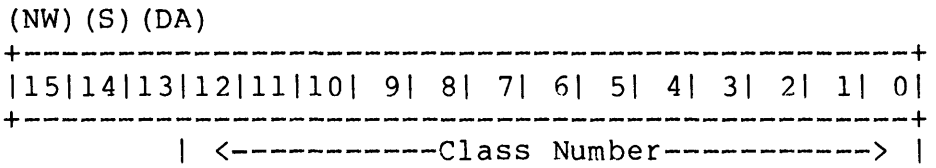
* Bit 11 and 13-15. These bits are used by the system and should be set to zero by the user.

ILEN---This variable defines the length of the data record to be transferred. A positive number is used to specify the length of the data record in words. A negative number is used to specify the length of the data record in characters (bytes). The data type must be allowed for in ILEN. For example, a data record containing 10 REAL values would require ILEN to be +20 or -40 since each REAL value requires two words. If the data record contains double-precision data, three or four words are required for each value, depending on the options taken at generation time for the FORTRAN compiler (refer to the appropriate FORTRAN Reference Manual).

OPTIONAL PARAMETERS (IOP1 AND IOP2)---These two parameters are required or optional, depending on the state of the "Z" bit. If the Z-bit is set, IOP1 defines the address of an optional control buffer and IOP2 defines the length of the buffer in positive words or negative characters. The control buffer can contain driver dependent information or control information to be used by the program that issues the Class Get call. (The user should refer to the appropriate Driver Reference Manual for more details.)

If the Z-bit is clear ("0"), IOP1, and IOP2 can be used to pass one-word parameters to the Class Get (EXEC 21) call described later in this section.

CLASS WORD (ICLAS)---ICLAS is the parameter used by a program(s) to coordinate the various Class I/O operations. The format of the class word is as follows:



The function of the class word fields are described below:

* CLASS NUMBER (bit 0-12) - A class number is assigned to the user program, generally when the first Class I/O or CLRQ request is issued. The preferred method of allocating a class number is with the CLRQ library routine described in Chapter 5. To obtain a class number from the system in a Class I/O EXEC Call, these bits are set to zero. This causes the system to allocate a class number (if one is available) to the calling program. The number is returned to the ICLAS parameter when the call completes and the user must specify this parameter (unaltered) when using it for later calls.

* "No-Wait" Bit (bit 15) - When set the calling program does not suspend if memory (SAM) or a class number is not available. The A-Register should be checked for the status of the call. Note that this is different than the no-suspend bit which prevents the calling program from suspending when a device is downed due to the Class I/O request.

* "Save" Bit (bit 14) - When set the data buffers allocated by a Class Write request will be saved for future processing by a Class Get. For Class Read, or Write/Read calls, this bit has no significance.

* "De-Allocate" Bit (bit 13) - For Class Write, Read, or Write/Read calls, this bit has no significance.

A- AND B-REGISTER RETURNS---When the user program issues a Class I/O request, the system allocates a buffer from SAM and puts the request in the SAM buffer. The request is queued on the EQT and the system returns control to the user program. If memory is not available, one of three conditions exists:

1. The program is requesting more memory space than will ever be available. In this case, the program is aborted with an I004 error, unless the no-abort bit was set in the ICODE parameter. If the no-abort option was set, then the A-Register will contain ASCII "I/O" and the B-Register will contain an ASCII "04".

2. The program is requesting a valid amount of memory but the system must wait until memory is returned before it can satisfy the request. The program is suspended unless the no-wait bit is set, in which case a return is made with the A-Register set to -2.

3. If the buffer limit is exceeded, the program will be suspended until this condition clears. If the no-wait bit is set, the program is not suspended and the A-Register is set to -2.

The A-Register will contain -1 if the no-wait bit was set and the program tried to allocate a class number with no class numbers available.

The A-Register will contain zero if the request was successful.

For a successful request, the returned content of the B-Register is meaningless. Error information is returned to the A- and B-Register for unsuccessful calls (refer to "EXEC Call Error Returns").

Class Write---The general flow of a Class Write operation is as follows:

1. User places data in IBUFF, specifies data length in ILEN, and issues an EXEC 18 call specifying a previously allocated class number in ICLAS (or if ICLAS = 0, a class number can be allocated, if available).

2. The system allocates a buffer in SAM (if available) large enough to contain the data in IBUFF plus the request parameters. It then moves the data from IBUFF plus the request parameters into the buffer in SAM. The calling program continues executing or suspends itself with an EXEC 21 call to the class number.

3. The request is queued (according to program priority) on the EQT associated with the LU specified in ICNWD.

4. When the driver completes the Write operation, the request portion of the buffer (the header) is linked into the Completed Class Queue. The data portion of the buffer is released back to the system (SAM), unless the "save buffer" bit in the ICLAS parameter is set. Any program suspended by a previous Class Get call (EXEC 21) on the class number is rescheduled. Refer to the "Class Get - EXEC 21", section for details associated with the Class Get call.

Class Read---The general flow of a Class Read operation is as follows:

1. User issues an EXEC 17 call specifying a previously allocated class number in ICLAS (or if ICLAS = 0, a class number can be allocated, if available). The amount of data to be transferred from the external I/O device to a buffer in SAM is specified in ILEN.

2. The system allocates a buffer in SAM large enough to contain the data plus the request parameters. It then places the request parameters in the buffer in SAM. The calling program continues executing or suspends itself with an EXEC 21 call to the class number.

3. The request is then queued (according to the calling program's priority) on the EQT associated with the LU specified in ICNWD.

4. When the driver completes the transfer of data from the external I/O device to the buffer in SAM, the entire class buffer is linked into the Completed Class Queue. Any program suspended by a previous Class Get call (EXEC 21) on the class will be rescheduled. Refer to "Class Get - EXEC 21", for details concerning the Class Get call.

Class Write/Read---The general flow of a Class Write/Read request has characteristics of both the Class Write and Class Read calls described above. A Class Write/Read call functions like a Class Write except that the driver receives a Class Read request code and the buffer is not released on I/O completion. Both the request portion (the header) plus the data portion of the class buffer is linked into the Completed Class Queue and the flow continues like a Class Read request. Hence the name Write/Read.

The Class Write/Read call with LU=0 is used for program-to-program communication. The data is transferred from the user program's buffer into the class buffer and written to LU 0 ("bit-bucket"). The data is retained in the Completed Class Queue to be recovered by an EXEC 21 call from another program.

Refer to the end of the Class I/O EXEC Call section for examples of each Class I/O Request Call.

Class I/O Control — EXEC 19

Performs various I/O control operations such as backspace, write end-of-file, rewind, etc. The calling program does not wait for the operation to complete.

```
CALL EXEC(ICODE,ICNWD[,IPRM],ICLAS[,IOP1][,IOP2])
```

ICODE - Request code. 19 = I/O control.

ICNWD - Control word. Specifies the Session LU and the control function to be carried out on that LU.

IPRM - Optional parameter. Required for some control functions. Must be used as a place holder if not required for the control operation.

ICLAS - Class word.

IOP1 - Optional buffer address.

IOP2 - Optional buffer length

COMMENTS:

PARAMETER CONSIDERATIONS---ICNWD and IPRM perform the same function for the Class I/O control operation (EXEC 19) as for the standard I/O control operation (EXEC 3) described earlier. The no-suspend option is also applicable to the ICODE parameter. Refer to the section on the EXEC Call Error Returns for a detailed explanation.

ICLAS is described earlier in this section with the discussion of the Class Read, Write, and Write/Read EXEC calls.

IOP1 and IOP2 can be used to pass an optional control buffer to the Class Get call described later in this section.

A- AND B-REGISTER RETURNS---For successful EXEC 19 calls, the A-Register contains zero; the contents returned to the B-Register are meaningless. Error information is returned to the A- and B-Register for unsuccessful EXEC 19 calls (refer to EXEC Call Error Returns earlier in this chapter).

GENERAL FLOW---The general flow of the EXEC 19 call is as follows:

1. User program makes EXEC 19 call specifying the LU and operation to be performed in ICNWD. A previously allocated class number is specified in ICLAS (or if ICLAS=0, a class number can be allocated, if available).

2. The system allocates a buffer in SAM and places the request parameters in the SAM buffer. The calling program can continue executing or suspend itself with an EXEC 21 call to the class number.

3. The request is queued (according to program priority) on the EQT associated with the LU specified in ICNWD.

4. When the driver completes the control operation, the request is linked into the Completed Class Queue. Any program suspended by a previous Class Get call (EXEC 21) to the class is rescheduled. Refer to "Class Get - EXEC 21" for details associated with the Class Get call.

Refer to the end of this section for an example of the EXEC 19 call.

Class Get — EXEC 21

Completes the operation previously initiated by a Class Read, Write, Write/Read or Control request.

```
CALL EXEC(ICODE,ICLAS,IBUFF,ILEN[,IOP1][,IOP2][,IOP3])
                      -----      ----   ----   ----
```

ICODE - Request code. 21 = Class Get.

ICLAS - Class word.

IBUFF - Data buffer. Array (in calling program) where the system places retrieved data.

ILEN - Data length. Positive number of words or negative number of characters (bytes) to be retrieved.

IOP1 IOP2 - Optional control buffer parameters obtained from the original class request.

IOP3 - Optional parameter indicating type of class request; Read, Write, Write/Read, or Control.

COMMENTS:

CLASS WORD (ICLAS)---The format of the class word is described earlier in this section (refer to "Class Read, Write, and Write/Read - EXEC 17, 18, and 20"). The function of bits 15, 14, and 13 of the class word are described below:

* "No-Wait" Bit (15)---When set, the calling program is not suspended if the results of the operation are not yet available, i.e., the request has not been linked into the Completed Class Queue.

* "Save" Bit (14)---When set, the buffer in the Completed Class Queue is not released; a subsequent Class Get call will return the same data.

* "De-Allocate" Bit (13)---When set, the class number is not released back to the system. If bit 13 is cleared to zero, and no requests for the class are pending on the driver for processing, the class number is released to the system.

Bits 14 and 13 work in conjunction with each other. If bit 14 is set, the buffer is not released and the class number cannot be released because there will still be an outstanding request against it.

If CLRQ was not used to assign the class number, then the class number should always be released when it is no longer needed. Only when the Class Get call retrieves the results of the last request on a class or examines an empty class queue can the class number be released by clearing bit 13 in the class word (ICLAS). The system does not automatically release the class number when the allocating program terminates, unless the class number is allocated using the CLRQ routine. (Refer to Chapter 5 for details on the CLRQ routine.)

DATA LENGTH (ILEN)---This variable defines the length of the data record to be retrieved (positive number of words, negative number of characters); the data type must be allowed for. If the data record contains Real values, two words per value are required. If the data record contains double-precision data, three or four words per value are required depending on the options taken at generation time for the FORTRAN compiler (refer to the appropriate FORTRAN Reference Manual).

Note that ILEN must allow for the length of the optional output buffer (Z-bit set in original call) if it is to be retrieved.

OPTIONAL PARAMETERS (IOP1, IOP2, AND IOP3)---Optional control buffer parameters from the Class Read, Write, Write/Read, or Control calls are returned in IOP1 and IOP2. These words are protected from modification by the driver. The original request code received by the driver is returned in IOP3 as follows:

ORIGINAL REQUEST CODE	IOP3 RETURN
17/20 (READ,WRITE/READ)	1
18 (WRITE)	2
19 (CONTROL)	3

GENERAL FLOW---When a program issues a Class Get call, the program is telling the system that it is ready to accept data provided by a previous Class Read or Write/Read call, or remove a completed Class Write or Control request from the Completed Class Queue. If the driver has not yet completed (Class Get call was issued before request was linked into Completed Class Queue), the calling program is suspended in the general wait list (state 3) and a marker so indicating is entered in the class queue header. When the driver completes, the program is automatically rescheduled. If desired, the program can set the no-wait bit to avoid suspension.

One of the features of the Class Get call is that a user program waiting for system resources can suspend itself without CPU overhead or program overhead such as polling. A program can issue a Class Get call on a class number associated with a device or another program and thereby suspend itself. The program will be rescheduled when there is data available to process; the requested data will be available. After the data is processed, the program can again suspend itself with another Class Get call.

BUFFER CONSIDERATIONS---There are several buffer considerations in using the Class Get call:

* The number of words returned to IBUFF is the lesser of:

  a. the number requested (ILEN specified in Class Get call).

  b. the number in the Completed Class Queue element being retrieved. (ILEN specified in original request.)

* If the original request was made with the Z-bit set in the control word (ICNWD), the returned value of IOP1 will be meaningless.

* The "Z-buffer" will be returned only if the original request was a Read or Write/Read call; for Class Write requests, no data is returned. Note that ILEN must allow for the length of the Z-buffer; ILEN = length of original request buffer + length of Z-buffer.

* The remaining words in IBUFF (if any) beyond the number indicated by the transmission log (B-Register), are undefined. If a Z-buffer is also returned, the words remaining beyond the end of the Z-buffer are undefined.

A- AND B-REGISTER RETURNS---The A- and B-Register contents after the return from a successful EXEC 21 call are as follows:

If a return is made with data, then:

A-Register (bit 15) = 0

A-Register(bit 14-0)=status (word 5 of devices EQT entry).

B-Register=transmission log (the positive number of words of characters transferred depending on the original request.

If a return is made without data (no-wait bit set in ICLAS), then:

A-Register = negative number of (requests +1) made to the class but not yet serviced by the driver (pending class requests).

B-Register = meaningless.

For unsuccessful calls, error information is returned in the A- and B-Registers (refer to "EXEC Call Error Returns").

CLASS I/O EXEC CALL EXAMPLES

Example 1. Initiate a Class I/O Read from a terminal into a buffer. Use the library routine CLRQ to allocate the class number.

```
      PROGRAM CREAD
      INTEGER BUFR(32)
         :
C     USE CLRQ TO ALLOCATE THE CLASS NUMBER
C
      ICLAS=0
      IFUNC=1
      CALL CLRQ(IFUNC,ICLAS)
C
C     REQUEST INPUT OF DATA FROM THE TERMINAL
      ICODE=17
      ICNWD=1
      CALL EXEC(ICODE,ICNWD,BUFR,32,0,0,ICLAS)
C
C     CONTINUE EXECUTION WHILE I/O IS BEING DONE
         :
C
C     COMPLETE THE OPERATION - CLASS GET
C
      CALL EXEC(21,ICLAS,BUFR,-64)
         :
      END
```

Example 2. Execute a Class I/O Write from a buffer to the line printer (LU 6). Check for errors using the no-abort and the no-suspend option.

```
      PROGRAM CWRIT
      INTEGER BUFR(32)
          :
C     FILL THE BUFFER
          :
C     ALLOCATE THE CLASS NUMBER
      ICLAS=0
      IFUNC=1
      CALL CLRQ (IFUNC,ICLAS)
C
C     OUTPUT DATA TO LINE PRINTER - CLASS WRITE
      ICODE=18
      ICNWD=1
      CALL EXEC(ICODE+140000B,ICNWD,BUFR,32,0,0,ICLAS,*500)
C
C     EXEC CALL WAS SUCCESSFUL - CONTINUE EXECUTION
C     WHILE THE I/O IS BEING COMPLETED
          :
C     COMPLETE THE OPERATION - CLASS GET
C     AND DEALLOCATE THE CLASS NUMBER
C
  200 CALL EXEC(21,ICLAS+20000B,BUFR,32)
          :
  500 CONTINUE
C     ERROR PROCESSING
          :
      END
```

Example 3. Class Write/Read using program to program communication. Use the routine CLRQ to assign the class number.

```
      PROGRAM PRG3A
      INTEGER BUFR(32),NAME(3)
      DATA NAME/'PR63B'/
C     FILL THE BUFFER
         :
C     ALLOCATE THE CLASS NUMBER
C
      ICLAS=0
      IFUNC=1
      CALL CLRQ(IFUNC,ICLAS,NAME)
         :
C     DO CLASS WRITE/READ TO LU ZERO FOR PROGRAM
C     TO PROGRAM COMMUNICATION
      ICODE=20
      CALL EXEC(ICODE+140000B,0,BUFR,-64,0,0,ICLAS,*500)
         :
C     SCHEDULE PRG3B TO GET THE BUFFER PASSING
C     IT THE CLASS NUMBER
      CALL EXEC (10,NAME,ICLAS)
         :
  500 error processing
         :
      END
```

```
      PROGRAM PRG3B
      INTEGER BUFR(32),PARM(5)
C
C     GET THE CLASS NUMBER USING RMPAR
      CALL RMPAR(PARM)
      ICLAS=PARM(1)
C
C     ACCEPT DATA FROM PRG3A USING THE CLASS GET
      CALL EXEC(21,ICLAS,BUFR,32)
      END
```

Example 4. Class I/O Control Operation to rewind the tape on the magnetic tape drive.

```
      PROGRAM CCONT
      INTEGER DUMBUF(4)
C
C     ALLOCATE A CLASS NUMBER
      ICLAS=0
      IFUNC=1
      CALL CLRQ(IFUNC,ICLAS)
        :
C
C     REWIND THE MAG TAPE - CLASS CONTROL
C     MAG TAPE = LU 8
C     REWIND FUNCTION CODE = 04 (OCTAL)
      ICODE=19
      ICNWD=8+400B
      CALL EXEC(19,ICNWD,0,ICLAS)
        :
C
C     COMPLETE THE OPERATION - CLASS GET
      CALL EXEC(21,ICLAS,DUMBUF,4)
      END
```

Program Management EXEC Calls

The program management EXEC calls allow user-written programs to control their own execution or the execution of other programs. This includes scheduling, suspending, or terminating programs, as well as, controlling program swapping and implementing communication between programs. A list of the program management EXEC calls are summarized below in the order of their presentation.

* EXEC 6 - Program Completion
* EXEC 7 - Program Suspension
* EXEC 8 - Program Segment Load
* EXEC 9,10,23,24 - Program Scheduling
* EXEC 12 - Timed Program Execution
* EXEC 14 - String Passage
* EXEC 22 - Program Swapping Control

Program Completion — EXEC 6

Notifies the operating system that the calling program wishes to terminate itself or a subordinate program.

```
CALL EXEC(ICODE[,INAME][,ICMCD][,IOP1][,IOP2][,IOP3][,IOP4]
     [,IOP5])
```

ICODE - Request code. 6 = program termination

INAME - Program name. 3-word array containing the ASCII name of the program to be terminated (must be subordinate program). Set to zero if calling program wishes to terminate itself.

ICMCD - Completion code.

IOP1 thru IOP5 - Optional parameters. Only used when INAME=0 (self-termination). Parameter values are saved in the terminating program's ID segment; can be recovered when program next executes.

COMMENTS:

COMPLETION CODE (ICMCD). The completion code specifies the manner in which the program indicated by INAME is to be terminated. The codes are summarized below:

* ICMCD = 0 Normal completion.

* ICMCD = -1 Serially-reusable completion. When rescheduled, the program is not reloaded into memory if it is still resident and its ID segment still exists. Note that a type 6 program, which the system "RP's" automatically using the FMGR RU command, can never be serially-reusable. The system automatically releases its ID segment upon completion. (Refer to the RTE-6/VM Terminal User's Reference Manual for details on the FMGR RU command.) ICMCD = -1 should only be used with disc-resident programs that can initialize their own buffers or storage locations. The program is reloaded from disc only if it has been overlaid by another program, therefore, the program must be able to maintain the integrity of its data in memory.

* ICMCD = 1 Saving-resources completion. Makes program dormant; saves suspension point and all resources allocated to the program. When program is rescheduled, it will continue executing from point of suspension and have access to previously allocated resources. If a program does not terminate itself, it can only be rescheduled by its father or by the RU or ON operator command. If a program does terminate itself, it can be rescheduled by any stimulus (EXEC, operator, time, or interrupt scheduling). Programs that terminate saving resources must not be cloned. The program should be either linked using the LINK DC command (refer to the RTE-6/VM LINK User's Manual) or run with the :IH option specified. An EXEC 6 call with ICMCD = 1, is similiar to the EXEC 7 call described later or the SS system command described in the RTE-6/VM Terminal User's Reference manual.

* ICMCD = 2 Terminate program and remove it from the time list. If program is I/O suspended, the system waits until the I/O operation completes before setting the program dormant. Disc tracks allocated to the program are not released. An EXEC 6 with ICMCD = 2 is equivalent to the "OF,program,0" system command (refer to RTE-6/VM Terminal User's Reference Manual).

* ICMCD = 3 Immediately terminate program, remove it from time list, and release all disc tracks allocated to it. If program is I/O suspended, a system generated clear request is sent to the driver. An abort message is displayed on the system console. An EXEC 6 with ICMCD = 3 is equivalent to the "OF,program,1" system command (refer to RTE-6/VM Terminal User's Reference Manual).

OPTIONAL PARAMETER PASSAGE---If INAME = 0 (self-termination) and the optional parameters are included in the EXEC 6 call, they are stored in the program's ID segment, word 1 through 5, (refer to Appendix C for the segment format) along with their address which is placed in the B-Register save word (word 10) of the ID segment. When the program is again scheduled to execute, it can obtain the parameters by calling the library routine RMPAR (refer to Chapter 5 for a description of the RMPAR routine). The RMPAR call should be the first statement executed when the program is rescheduled to prevent the restored B-Register from being modified before RMPAR has used it to access the parameters. The use of the optional parameters allows a program being scheduled from the time list to use the same parameters each time it executes.

In order to pick up the parameters stored in the ID segment using RMPAR, the program must be rescheduled using the RU command or program scheduling EXEC calls.

A- AND B-REGISTER RETURNS---For successful EXEC 6 calls, the contents of the A- and B-Registers are summarized below:

* A-Register. Unchanged.

* B-Register. Unchanged (no optional parameters specified) or address of optional parameters.

Note that the FORTRAN Compiler generates an EXEC 6 when it compiles an END statement.

Note also that a father can terminate its son normally (ICMCD=0) or with the son saving resources (ICMCD=1).

EXAMPLE---Before program terminates itself, it obtains the current time (refer to "Time Request - EXEC 11"). The time is placed into the optional parameters used in the EXEC 6 call. When the program is rescheduled it obtains the saved time values and prints them on the session console (LU 1).

```
      PROGRAM EXC6
      DIMENSION ITM(5),IPAR(5)
      CALL RMPAR(IPAR)     !GET THE VALUES OF THE PASSED PARAMETERS
      IHRS=IPAR(4)
      IMIN=IPAR(3)
      ISEC=IPAR(2)
        .
        .
      WRITE(1,100)IHRS,IMIN,ISEC
 100  FORMAT("PROGRAM LAST RUN AT",I2,":",I2,":",I2)
      CONTINUE
        .
        .
      INAME=0
      ICMCD=0
      CALL EXEC(11,ITM)          !OBTAIN CURRENT TIME AND TERMINATE
      CALL EXEC(6,INAME,ICMCD,0,ITM(2),ITM(3),ITM(4))     !NORMALLY
      END
```

Program Suspend — EXEC 7

Suspends execution of the calling program. Program can be rescheduled by using the GO system command described in the RTE-6/VM Terminal User's Reference Manual.

CALL EXEC(ICODE)
ICODE - Request code. 7 = program suspend

COMMENTS:

A-REGISTER, B-REGISTER, AND PARAMETER PASSAGE---The execution of the EXEC 7 call causes the system to place the calling program into the operator suspend list (state 6). The contents of all registers are saved. When the program is rescheduled via the GO command (without parameters), the registers are restored to their presuspension status and the program resumes execution. If the program is rescheduled with a GO command that includes parameters, the B-Register will contain the address of the parameters and they can be recovered by calling the library routine RMPAR (refer to Chapter 5). The RMPAR call should immediately follow the EXEC 7 call to prevent the B-Register from being modified before RMPAR has used it to access the parameters.

Note that when a call to RMPAR is used, the optional parameters must be included in the GO command. If they are not included when the program is rescheduled, RMPAR will use the restored B-Register contents as the address of parameters that don't exist. If the parameters are included on a periodic basis, the program should be coded to check for optional parameters and by-pass the RMPAR call if they are not included (refer to the example below).

If an EXEC 7 call is unsuccessful, error information will be returned in the A- and B-Registers (refer to "EXEC Call Error Returns"). Note that it is illegal to use an EXEC 7 call to suspend a program running in the batch environment (refer to the Batch and Spooling Reference Manual). If attempted, an SC00 error is returned.

OTHER SUSPENSION OPERATIONS---The SS operator command described in the RTE-6/VM Terminal User's Reference Manual performs the same function as the EXEC 7 call. The FORTRAN PAUSE statement performs the same function as the EXEC 7 call and additionally displays an optional "pause-number" on the session console (refer to the appropriate FORTRAN Reference Manual).

EXAMPLE: Suspend the calling program. Use the FORTRAN function EXEC call technique (previously discussed) to implement optional parameter passage check. If parameters are included in GO command, call RMPAR to recover them. If parameters are not included, by-pass the RMPAR call.

```
      PROGRAM SUPND
      DIMENSION IPAR(5),IAB(2)
      EQUIVALENCE(REG,IAB(1))  !SET REG EQUAL TO THE A&B REGISTERS
      REG=0.0
      REG=EXEC(7)                        !SUSPEND THE CALLING PROGRAM
      IF(IAB(2)) 200,200,100  !DOES THE B-REG POINT TO PARAMETERS?
  100 CALL RMPAR(IPAR)                      !PARAMETERS WERE PASSED
      process parameters
       .
       .
  200 CONTINUE                           !PARAMETERS WERE NOT PASSED
       .
       .
      END
```

Note that REG=0.0 indirectly causes the A- and B-Registers to be set to zero.

Program Segment Load — EXEC 8

Loads the calling program's segment from disc into memory and transfers control to the segment's entry point. This EXEC call may not be used in a multi-level segmentation program (program loaded with the MLS-LOC Loader). Refer to the RTE-6/VM Loader Reference Manual for a detailed explanation of program segmentation.

```
CALL EXEC(ICODE,INAME[,IOP1][,IOP2][,IOP3][,IOP4][,IOP5])
```

ICODE	Request code. 8 = segment load.
INAME	Segment name. 3-word array containing the ASCII name of the segment to be loaded.
IOP1 thru IOP5	Optional parameters; passed to segment specified in INAME.

COMMENTS:

OPTIONAL PARAMETER PASSAGE---The calling program can pass up to five optional parameters to the segment specified in INAME. The calling program places the values to be passed into IOP1 through IOP5. When the segment load call is executed, these values are placed into the five word temporary area of the program's ID segment and the address is placed in the B-Register. When control is transferred to the segment's entry point, the segment can recover these values by calling the library routine RMPAR (refer to Chapter 5). Note that the call to RMPAR should be the first executable statement in the segment's code to prevent the B-Register from being modified by the segment's execution before RMPAR has used it to access the parameters.

A- AND B-REGISTER RETURNS---When control has been successfully transferred to the segment via its entry point, the contents of the A- and B-Registers will be as follows:

* A-Register will contain the segment's ID segment address.

* B-Register will not be changed unless optional parameters are being passed, in which case the B-Register will contain the address of the parameters (used by RMPAR).

If the no-abort bit is set and an error occurs, it is indicated in the A- and B-Registers (refer to "EXEC Call Error Returns"). If the segment to be loaded does not exist, an SC05 error is returned.

If the program making this EXEC request is a multi-level segmentation program, an SC12 error is returned.

PROGRAM TYPES---The main program must be a type 2, 3, 4, or 6 program. The segment must be a type 5 program. The program's type can be defined in its program definition statement (refer to Appendix D for a table of program types).

EXAMPLE: Main program loads its segment and passes it three parameters.

```
      PROGRAM MAIN,3
      DIMENSION INAME(3)
      DATA INAME/6HSEGMT /                                  !SEGMENT NAME
       .
       .
      place values in IOP1,IOP3,and IOP5
       .
       .
      CALL EXEC(8,INAME,IOP1,IOP2,IOP3,IOP4,IOP5) !LOAD THE SPECIFIED
       .                                                    !SEGMENT
       .
      END

      PROGRAM SEGMT,5
      DIMENSION IPAR(5)
      CALL RMPAR(IPAR)                  !PICK UP THE PASSED PARAMETERS
      process parameters obtained from main
       .
       .
      END
```

Program Scheduling — EXEC 9, 10, 23, and 24

Schedules a program for execution and optionally passes up to five parameters and/or a buffer to the program.

```
CALL EXEC(ICODE,INAME[,IOP1][,IOP2][,IOP3][,IOP4][,IOP5]
  [,IBUFF]<,ILEN>)
```

ICODE - Request Code. 9=immediate schedule with wait.
10=immediate schedule without wait.
23=queue schedule with wait.
24=queue schedule without wait.

INAM - 3-word array containing ASCII name of program to be scheduled.

IOP1 thru IOP5 - Optional parameters; passed to program specified in INAME.

IBUFF - Optional buffer; passed to program specified in INAME.

ILEN - Data length; positive number of words or negative number of characters (bytes) to be passed from IBUF to the program specified in INAME.

COMMENTS:

FATHER AND SON---When one program schedules another, the program that did the scheduling is known as the "Father" and the program that was scheduled is known as the "Son". The system places a pointer in word 20 of the son's ID segment that points back to the father's ID segment. If a program is not scheduled by another program, e.g., has no father, the pointer is set to zero. This would be the case if a program was scheduled by the RU command, an interrupt, or scheduled from the time list. If the father/son relationship exists and the son completes or is terminated, the pointer is cleared.

Note that the father will receive an error return (SC11) if it tries to schedule a son that has been placed in the time list from another session.

SCHEDULING WITH OR WITHOUT WAIT---When a father schedules a son without wait (ICODE=10,24), the son is scheduled for execution according to its priority. The father continues executing at its priority without waiting for the son to complete.

When a father schedules a son with wait (ICODE=9,23), the father is placed in the general wait list (state 3). While in this state, the father is swappable. The son executes according to its priority. When the son completes or is terminated, the father resumes execution at the point immediately following the scheduling EXEC call.

ICODE=9 OR 10; IMMEDIATE SCHEDULE (WITH OR WITHOUT WAIT)---In order for the father program to immediately schedule a son (ICODE=9 or 10), the son must be dormant (state 0). If the son is not dormant, it is not scheduled. The father can determine whether the son was scheduled or not by examining the contents of the A-Register after the execution of the scheduling call (refer to Table 2-6). The A-Register contents will be as follows:

* If the son was successfully scheduled (dormant before call) the A-Register will contain zero.

  Note that if the son was suspended by a previous EXEC 6 call (refer to "Program Completion - EXEC 6"), only the four least significant bits of the A-Register will be zero.

* If the son could not be scheduled (was not dormant) the A-Register will contain the status of the son.

  Note that if the son puts itself in the time list using self schedule with offset, the father's schedule attempt will be honored, the son will proceed from the current point of suspension, and no parameters may be passed to the son. This is only true if the father/son relationship exists as mentioned earlier. Also, the son will remain in the time list; if dormant when time scheduled, it will execute from its current point of suspension or from the beginning of the program in case it had terminated normally.

ICODE = 23 OR 24; QUEUE SCHEDULE (WITH OR WITHOUT WAIT)---The EXEC 23 and 24 calls are similiar to the EXEC 9 and 10 calls, respectively, except that the system places the father in a queue if the son is not dormant. While in the queue, the father is suspended until the son can be scheduled. When the potential son goes dormant, and can therefore be scheduled, the system reissues the father's request and execution proceeds like the immediate schedule calls described above (refer to Table 2-6). That is, for an EXEC 23, when the son becomes available, the father is removed from the queue (rescheduled) and execution continues as an EXEC 9. An EXEC 24 would continue as an EXEC 10.

Note that the father does not have access to the son's status via the A-Register as with the immediate schedule calls.

For information on scheduling FMGR programmatically, refer to Appendix J.

SON TERMINATION CONSIDERATIONS---When a father performs an immediate schedule with wait (ICODE = 9) or a queue schedule with wait (ICODE = 23), it is suspended until the son completes or is terminated. The father can determine if the son terminated normally or not by examining the system's copy of optional parameter 1 (refer to "Optional Parameter Passage" below). The father does this by calling the RMPAR routine which is described in Chapter 5. The returned contents of IOP1 are summarized below:

* IOP1 = 100000B. Indicates that the son terminated abnormally for one of the following reasons:

    1. System aborted son

    2. Operator issued OF command (refer to RTE-6/VM Terminal User's Reference Manual).

    3. Son performed an EXEC (6,0,2) or EXEC (6,0,3) self-termination call (refer to "Program Completion - EXEC 6").

* IOP1 = Original value or value passed back by son indicates that the son terminated normally.

OPTIONAL PARAMETER PASSAGE---The optional parameters (IOP1 - IOP5) can be used to pass values from the father to the son. The father places the values to be passed into the optional parameters and then makes the scheduling call. When the son begins executing, it can obtain these values by calling the library subroutine RMPAR. Note that the RMPAR call should be the first executable statement in the son program. See Special Considerations under EXEC 9,10.

If the father performed a schedule with wait call, the son can pass back up to five parameters to the father. To do this, the son would make a call to the library routine PRTN (refer to Chapter 5) to place the values to be returned into a five-word array. The son should then terminate itself using an EXEC 6 call to prevent B-Register modification.

The son's termination causes the father to be rescheduled. The father can then obtain the returned values by calling RMPAR. Note that if the son terminates abnormally, the first returned parameter will be overwritten with 100000B (refer to "Son Termination Considerations", above).

OPTIONAL BUFFER PASSAGE---In addition to five optional parameters, the father can pass an optional buffer to the son. The father places the data to be passed into IBUFF and specifies the length of the data in ILEN. When the father makes the scheduling call, the buffer is moved into SAM. The son can obtain the buffer by making an EXEC 14 call (refer to "String Passage - EXEC 14"). If there is not enough SAM currently available to hold the buffer, the father is memory suspended (state 4). If there will never be enough SAM available, the father is aborted or, if the no-abort bit is set, the error return is taken (refer to "EXEC Call Error Returns"). Note that the length of the buffer is limited only by the amount of usable SAM available.

If the father performed a schedule with wait call, the son can pass a buffer back to the father by using an EXEC 14 call and the father can recover the buffer with an EXEC 14 call. If a program is scheduled by an operator command, e.g., the RU, ON, or GO command described in the RTE-6/VM Terminal User's Reference Manual, an EXEC 14 call can be used to obtain the command string or a call to the library routine GETST can be used to obtain the parameter string. GETST is described in Chapter 5. The command string and the parameter string are described with the discussion of the EXEC 14 call later in this section.

A- AND B-REGISTER RETURNS---For successful EXEC scheduling calls, the contents of the A- and B-Registers are used to determine status and parameter location information. These register returns have been discussed in the above text with the calls that they pertain to. For convenience they are briefly summarized below:

* A-Register (for immediate schedule). 0 if son was scheduled, son's status if son could not be scheduled.

* A-Register (for queue schedule). Meaningless.

* B-Register (when son is scheduled with wait). Address of parameters, if optional parameters are being used. If optional parameters are not used, B-Register is cleared.

For unsuccessful calls, the A- and B-Register will contain error information (refer to "EXEC Call Error Returns").

The following table summarizes the program scheduling EXEC calls (9,10,23,24).

Table 2-6. Summary of EXEC Calls 9,10,23,24

EXEC CALLS	SON DORMANT	SON ACTIVE
9-IMMEDIATE WITH WAIT	Son scheduled. Father goes into state 3 until son completes.	Son not scheduled. Son's state in A-Reg.
10-IMMEDIATE WITHOUT WAIT	Son scheduled. Father stays in state 1.	Son not scheduled. Son's state in A-Reg.
23-QUEUE WITH WAIT	Son scheduled. Father goes into state 3 until son completes.	Father goes into state 3 until son can be scheduled. Son scheduled. Father stays in state 3 until son completes.
24-QUEUE WITHOUT WAIT	Son scheduled. Father stays in state 1.	Father goes into state 3 until son can be scheduled. Son scheduled. Father goes into state 1.

EXAMPLE---Father schedules a son (queue schedule with wait, ICODE=23). Father passes the son five parameters. Son returns five parameters to father. Father checks for EXEC errors.

```
      PROGRAM FATHR
      DIMENSION IPAR(5),INAM(3)
      DATA INAM/6HSONNY /                    !NAME OF SON PROGRAM
        .
        .
      place values in optional parameters (IOP1-IOP5)
        .
      ICODE=23+100000B                    !SCHEDULE SON WITH WAIT
      CALL EXEC(ICODE,INAM,IOP1,IOP2,IOP3,IOP4,IOP5,*100)
   10 CALL RMPAR(IPAR)           !PICK UP PARAMETERS SENT BY SON
      IF(IPAR(1).EQ.100000B) GO TO 200
      process five parameters returned by son.
        .
        .
  100 CALL ABREG(IA,IB)
      error processing
        .
        .
  200 CONTINUE
        .
        .
      END

      PROGRAM SONNY
      DIMENSION IP(5)
      CALL RMPAR(IP)             !PICK UP PARAMETERS FROM FATHER
      process five values passed by father
        .
        .
      place values to be returned to father in IP
        .
      CALL PRTN(IP)                    !SEND PARAMETERS BACK AND
      CALL EXEC(6)                                    !TERMINATE
      END
```

Program Time Scheduling — EXEC 12

Places calling program or another program into the time scheduling list. The EXEC 12 call has two modes of operation:

1. Absolute Time Mode - a program can be scheduled to run once at an absolute time, or to run at specified intervals, with the first run scheduled at an absolute time.

2. Initial Offset Mode - a program can be scheduled to run once at a time offset from the current time, or to run at specified intervals, with the first run scheduled at a time offset from the current time.

```
Initial offset:
     CALL EXEC(ICODE,INAME,IRESL,IMULT,IOFST)

Absolute time:
     CALL EXEC(ICODE,INAME,IRESL,IMULT,IHRS,IMIN,ISEC,IMSEC)
```

ICODE - Request code. 12 = time scheduling.

INAME - Program name. 3-word array containing ASCII name of program to be scheduled. Set to 0 if calling program is to be scheduled.

IRESL - Resolution code. Specifies units to be used with IMULT and IOFST.

1 = 10's of milliseconds	3 = minutes
2 = seconds	4 = hours

IMULT - Execution multiple. Integer (0-4095) that specifies the time interval between runs for programs that run repeatedly. Used in conjunction with IRESL. 0 indicates that program is to run once.

IOFST - Initial offset. Negative integer that specifies offset from current time that a program will first run. Used in conjunction with IRESL.

IHRS
IMIN
ISEC
IMSEC - Specifies the hours, minutes, seconds, and milliseconds, respectively, when a program will first run.

COMMENTS:

INITIAL OFFSET---The initial offset version of the EXEC 12 call can be used to schedule programs (other than the calling program) as follows:

* Run once - IMULT is set to zero to specify that a program is to run once. IRESL and IOFST are set to indicate the offset from the current time that the program is to be run. For example, if IMULT=0, IRESL=3, and IOFST=-45, the program specified in INAME will run once, 45 minutes from the current time. Note that the program to be scheduled must be dormant at that time or the call is ignored.

* Run Repeatedly - IMULT is set to specify the interval between run times; IRESL and IOFST indicate the offset from the current time that the program will first run. For example, if IMULT=30, IRESL=3, and IOFST=-60, the program specified in INAME will run every 30 minutes, with the first run being 60 minutes from the current time. The program to be scheduled must be dormant at the indicated scheduling times or the call is ignored for that time period. The future scheduling times are not affected.

ABSOLUTE TIME---The absolute start time version of the EXEC 12 call can be used to schedule programs (other than the calling program) as follows:

* Run once - IMULT is set to zero to specify that the program is to run once. IRESL is unused. The time for execution is specified in IHRS, IMIN, ISEC, and IMSEC. For example, if IHRS=18, IMIN=45, ISEC=30, and IMSEC=0, the program specified in INAME would be scheduled to execute at 18:45:30:0. If the program is not dormant at the specified time, the scheduling call is ignored. If a time is specified that has already passed, the program will be scheduled in the next 24-hour period of the clock.

* Run repeatedly - IMULT is set to indicate the interval between run times. IRESL specifies the units to be used with IMULT. IHRS, IMIN, ISEC, and IMSEC specify the first time that the program will run. For example, if IRESL=3, IMULT=30, IHRS=18, IMIN=30, ISEC=0, and IMSEC=0, the program specified in INAME would be scheduled to execute every 30 minutes with the first run to be at 18:30:0:0. The program must be dormant when a scheduling time arrives, or the call is ignored for that time period. Future scheduling times are not affected. If the specified first-run time has already passed, the programs scheduling cycle will be started in the next 24-hour period of the clock.

LIMITATIONS---For IRESL=3, IMULT and IOFST must be less than 1440. For IRESL=4, IMULT and IOFST must be less than 24.

SELF TIME SCHEDULING---If INAME is set to zero, the calling program time schedules itself.

* Initial Offset - After the calling program executes the EXEC 12 call, the program is set dormant and rescheduled at the specified offset from the current time. The point of suspension (after the EXEC 12 call) is saved, and the execution of the program continues from that point when it is rescheduled.

  The calling program can be scheduled to run repeatedly with the EXEC 12 call. IMULT is set to specify the interval between run times. The calling program is set dormant after the EXEC 12 call and is rescheduled at the specified offset from the current time. The program continues executing at the point of suspension until completion. The program then schedules itself for future execution starting at the beginning of the program. The program can be designed to avoid executing the EXEC 12 call repeatedly.

* Absolute Time - The program is placed into the timelist to be rescheduled at the specified starting time. The calling program continues execution, does not go dormant, but schedules itself for future execution starting at the beginning of the program.

  For special considerations with father/son relationships, when the son uses the initial offset method, see comments under EXEC 9,10 scheduling calls.

A- AND B-REGISTER RETURNS---Upon successful completion of the call, the A-Register contents will be meaningless and the B-Register contents will be unchanged. For unsuccessful calls, the A- and B-Registers will contain error information (refer to "EXEC Call Error Returns").

Note that the EXEC 12 call performs a function similiar to the IT operator command described in the RTE-6/VM Terminal Users' Reference Manual.

Note also that the calling program will receive an error return (SC11) if it attempts to time schedule a program that is currently scheduled (or in the time list) for another session. If an attempt is made to time schedule a program that is currently scheduled for your session, the system will ignore the attempt.

EXAMPLE 1--Schedule program TIMER to execute every two hours with the first run at 10:30:0:0.

```
PROGRAM SCHD1
DIMENSION INAME (3)
DATA INAME/6HTIMER /
  :
IRESL=4                                              !HOURS
IMULT=2                    !RUN REPEATEDLY EVERY 2 HOURS
IHRS=10                                    !AT 10:30:0:0
IMIN=30
ISEC=0
IMSEC=0
  :
CALL EXEC (12,INAME,IRESL,IMULT,IHRS,IMIN,ISEC,IMSEC)
  :
END
```

EXAMPLE 2--Schedule calling program to execute every 30 minutes between runs. (Note that each time this program runs, the EXEC 12 call is executed again, causing a one minute offset after the EXEC 12 call.)

```
PROGRAM SCHD2
  :
IRESL=3                                            !MINUTES
IMULT=30          !RUN REPEATEDLY AT 30 MINUTE INTERVALS
IOFST=-1                    !EACH RUN IS OFFSET 1 MINUTE
  :
CALL EXEC(12,0,IRESL,IMULT,IOFST)
  :
END
```

String Passage — EXEC 14

Allows a program to retrieve a buffer from the program that scheduled it (father) or retrieve the command string if it was scheduled by an operator command. Allows a son to pass a buffer back to its father.

```
CALL EXEC(ICODE,IRWCD,IBUFF,ILEN)
          =====
```

ICODE - Request code. 14 = string passage.

IRWCD - Retrieve/write code. 1 = retrieve buffer or command string. 2 = write buffer to father.

IBUFF - Buffer into which retrieved data or command string is placed (IRWCD = 1) or a buffer into which son places date to be returned to father (IRWCD = 2).

ILEN - Positive number of words or negative number of characters (bytes) to be transferred.

COMMENTS:

COMMAND STRING---When a program is scheduled using an operator command (refer to RU, ON, or GO commands in the RTE-6/VM Terminal Users' Reference Manual), a command string is placed in a buffer in SAM. The command string is a copy of the command used to schedule the program. For example, if the RU command with optional parameters was used to schedule a program, the command string would appear as:

```
RU,PROG,IP1,IP2,IP3,IP4,IP5,STRING
```

where:

PROG = name of program to be scheduled
IP1 - IP5 = one-word parameters
STRING = ASCII string

If a program that was scheduled by an operator command performs an EXEC 14 call (IRWCD=1), the command string contained in the SAM buffer is retrieved and placed in IBUFF. It is the program's responsibility to parse the string into separate parameters.

Note that the program could call the library utility routine GETST to retrieve the parameters following the second comma in the command string (known as the parameter string). GETST is described in Chapter 5.

FATHER/SON BUFFER PASSAGE---If a father includes the optional buffer as a parameter in the EXEC call used to schedule a son (refer to "Program Scheduling - EXEC 9, 10, 23 or 24"), the contents of the buffer are placed in SAM when the call is executed. If the son performs an EXEC 14 (IRWCD=1), the buffer in SAM is retrieved, the contents are placed in IBUFF, and the SAM buffer is deallocated. The son program should perform the EXEC 14 prior to any other executable statement which may cause the SAM buffer to be overwritten or deallocated (e.g., on EXEC 23, scheduling D.RTR may overwrite the SAM buffer that was intended to be passed from the father to the son).

If the son was scheduled with wait (EXEC 9 or 23), the son can pass a buffer back to the father. The son would place the data to be returned into IBUFF, specify the length in ILEN, and perform an EXEC 14 call with IRWCD=2. The son should then terminate itself (refer to "Program Termination - EXEC 6"), allowing the father to be rescheduled. The father could obtain the returned buffer from SAM by performing an EXEC 14 call with IRWCD=1.

Note that the EXEC 14 write call (IRWCD=2) deallocates any block of SAM associated with the father, which the father created with an EXEC 14 call, and allocates a new block for the father into which the returned buffer will be placed.

ILEN---Only ILEN words or characters are transmitted. If the command string on a retrieve operation, or the buffer on a retrieve or write operation, is longer than ILEN, the excess data will be lost. If an odd number of characters are requested for a retrieve operation, the least significant byte of the last word is undefined.

A- AND B-REGISTER RETURNS---Upon successful return from a retrieve operation (IRWCD=1), the A- and B-Registers can be interpreted as follows:

* A-Register = 0 if operation was successful

* A-Register = 1 if no string could be found

* B-Register = positive number of words or characters transferred.

For unsuccessful calls, error information is returned (refer to "EXEC Call Error Returns").

EXAMPLE---Father schedules a son (immediate schedule without wait, ICODE=9) and passes it two optional parameters and an optional buffer. The son retrieves the buffer and prints its contents. The optional parameters contain the list and log LU to be used by the son.

```
      PROGRAM SENDR
      DIMENSION IBUF1(10),INAME(3)
      DATA INAME/6HGETER /        !NAME OF PROGRAM TO BE SCHEDULED
        :
      ICODE=9+100000B
      IOP1=1
      IOP2=6
      ILEN=10
        :
      place data in IBUF1
        :
C     SCHEDULE THE PROGRAM GETER WITH WAIT;
C     SENDING PARAMETERS AND THE BUFFER.
      CALL EXEC(ICODE,INAME,IOP1,IOP2,IOP3,IOP4,IOP5,IBUF1,ILEN,*100)
   10 CONTINUE
        :
  100 error processing
        :
      END

      PROGRAM GETER
      DIMENSION IBUF2(10),IPAR(5)
      CALL RMPAR(IPAR)              !PICK UP THE PASSED PARAMETERS
      LOGLU=IPAR(1)
      LSTLU=IPAR(2)
      ICODE=14+100000B
      CALL EXEC(ICODE,1,IBUF2,10,*100)   !RETRIEVE THE BUFFER FROM
   10 CALL ABREG(IA,IB)                                !THE FATHER
      IF(IA.NE.0) GO TO 200
      CALL EXEC(2+100000B,LSTLU,IBUF2,IB,*100)
   20 CONTINUE
        :
  100 CALL ABREG(IA,IB)
      WRITE(LOGLU,101)IA,IB
  101 FORMAT("A-REG=",I6,5X,"B-REG=",I6)
      CONTINUE
        :
  200 WRITE(LOGLU,'BUFFER NOT FOUND')
      CONTINUE
        :
      END
```

Program Swapping Control — EXEC 22

Allows a program to lock itself into memory, i.e., program performing call will not be swapped out if a higher priority program needs its partition. Allows program to release the lock.

```
CALL EXEC(ICODE,ILOCK)
```

ICODE - Request Code. 22 = swapping control

ILOCK - Lock control. 1 = program cannot be swapped
0 = program can be swapped

COMMENTS:

MEMORY LOCK CONSIDERATIONS - In order for a program to lock itself into a memory partition, the memory lock feature must be enabled at generation time (refer to RTE-6/VM On-Line Generator Manual). If the memory lock feature is not enabled and a program attempts to perform a memory lock, a SC07 error results.

When a program performs a memory lock (ILOC=1), bit 6 of word 15 of its ID segment is set. When a program performs a memory "unlock" (ILOCK=0), the bit is cleared. The bit is also cleared if the program aborts or terminates except if the program terminated because of an EXEC program completion call saving resources (refer to "Program Completion Call - EXEC 6").

A- AND B-REGISTER RETURNS---Upon successful completion of the call, the A-Register will be meaningless and the B-Register will be unchanged. If an error occurred, the A- and B-Registers can be examined for the cause (refer to "EXEC Call Error Returns").

EXAMPLE: Program locks itself into memory, performs critical processing, and unlocks itself. Checks for EXEC errors.

```
      PROGRAM LOCK
       .
       .
       .
      ICODE=22+100000B                      !SET THE NO-ABORT BIT
      ILOCK=1
      CALL EXEC(ICODE,ILOCK,*100)  !LOCK THE PROGRAM INTO MEMORY
   10 CONTINUE
       .         \
       .          > critical code
       .         /
       .        /
      ILOCK=0
      CALL EXEC(ICODE,ILOCK,*100)!UNLOCK THE PROGRAM FROM MEMORY
   20 CONTINUE
       .
       .
  100 CALL ABREG(IA,IB)
      error processing
       .
       .
       .
      END
```

Status EXEC Calls

The status EXEC calls are used to obtain information about the operating environment. This includes obtaining the current time indicated by the 24-hour system clock, obtaining size and status information about memory partitions, and determining the present status of an I/O device.

The status EXEC calls are listed below in the order of their presentation:

* TIME REQUEST - EXEC 11
* I/O STATUS - EXEC 13
* PARTITION STATUS - EXEC 25
* MEMORY SIZE - EXEC 26

Time Request — EXEC 11

Request the current time indicated by the 24-hour real-time clock.

```
CALL EXEC(ICODE,ITIME,[IYEAR])
          -----  -----
```

ICODE - Request code. 11 = time request.

ITIME - 5-word array. Time indicated by a real-time clock is returned by the system as follows:

```
ITIME(1) = 10's of milliseconds
ITIME(2) = Seconds
ITIME(3) = Minutes
ITIME(4) = Hours
ITIME(5) = Day of the year
```

IYEAR - Optional one-word variable. Current year is returned, by the system as a four-digit number.

COMMENTS:

A- AND B-REGISTER RETURNS---Upon successful completion of this call, the A-Register contents will be meaningless and the B-Register will be unchanged. If an error occurs, it is indicated in the A- and B-Register (refer to "EXEC Call Error Returns").

ALTERNATE METHODS---The EXEC 11 call is similiar to the TI system command described in the RTE-6/VM Terminal Users' Reference Manual. The library routine FTIME can also be used to obtain the current time (refer to the RTE-6/VM Relocatable Library Manual).

EXAMPLE: Obtain the current time on the real-time clock as well as the current year. Print out date and time as hrs:mins day/year on printer (LU 6).

```
      PROGRAM TRQST
      DIMENSION ITIME(5)
       .
       .
      ICODE=11
      CALL EXEC(ICODE,ITIME,IYEAR)        !GET THE TIME AND YEAR
      IHR=ITIME(4)
      IMIN=ITIME(3)
      IDAY=ITIME(5)
      WRITE(6,20)IHR,IMIN,IDAY,IYEAR
   20 FORMAT ("CURRENT TIME AND DATE ",I2,":",I2," ",I3,"/",I4)
      CONTINUE
       .
       .
      END
```

I/O Status — EXEC 13

Requests information (status and device type) about device associated with a specified LU.

```
CALL EXEC(ICODE,ICNWD,IST1[,IST2][,IST3])
```

ICODE - Request code. 13 = I/O status.

ICNWD - Session LU of device that status is requested on.

IST1 - Status word. Word 5 of the devices' EQT entry (refer to the Comments below).

IST2 - Status word. Word 4 of devices' EQT entry (refer to the Comments below).

IST3 - Status word. Specifies whether device is "up" or "down". Also indicates subchannel associated with the device.

COMMENTS:

STATUS WORDS---The contents of IST1 and IST2 are obtained from words 5 and 4, respectively, of the device's Equipment Table (EQT) entry. Since the status is obtained from the EQT and not the driver, the status obtained from a buffered device or a device with multiple subchannels, may not reflect the current device status. The format of these words is shown in Table 2-7 and Table 2-8.

The contents of IST3 is obtained from the Device Reference Table (DRT) entry associated with the device. Bit 15 of IST3 will be set to "1" if the device is down. Bit 15 will be cleared to "0" if the device is up. Bits 5 through 0 of IST3, will indicate the subchannel associated with the device. Refer to Appendix E for the Device Reference Table format.

If an LU greater than 63 is specified in ICNWD, the status of the LU with the value of (ICNWD-63) will be returned.

A- AND B-REGISTER RETURNS---The contents of the A- and B-Registers for a successful EXEC 13 call are meaningless. For unsuccessful calls, error information is returned (refer to "EXEC Call Error Returns").

EXAMPLE: Program ISTAT checks the LU passed to it in IPAR(1) to determine if it is associated with device type 05 or 07. If it is, execution continues. If it is not, the program terminates.

```
      PROGRAM ISTAT
      DIMENSION IPAR(5)
      CALL RMPAR(IPAR)        !GET LU PASSED IN THE RUN COMMAND
      ICNWD=IPAR(1)
      MASK5=2400B
      MASK7=3400B
      CALL EXEC(13,ICNWD,IST1) !GET I/O STATUS OF SPECIFIED LU
      ITEST=IAND(IST1,037400B)  !ITEST EQUALS EQUIP. TYPE CODE
      IF(ITEST.EQ.MASK5) GO TO 100       !CHECK IF LU IS DVR05
      IF(ITEST.EQ.MASK7) GO TO 100       !CHECK IF LU IS DVR07
      GO TO 200
       .
       .
  100 CONTINUE
       .
       .
  200 CALL EXEC(6)
      END
```

IAND is described in the RTE-6/VM Relocatable Library Manual.

Table 2-7. I/O Status Word (ISTA1/ISTA2) Format

WORD	CONTENTS															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
4	D	B	P	S	T	Unit #					Channel #					
5	AV		EQUIP. TYPE CODE					STATUS (see Table 2-8)								

ISTA2

- D = 1 if DMA required.
- B = 1 if automatic output buffering used.
- P = 1 if driver is to process power fail.
- S = 1 if driver is to process time-out.
- T = 1 if device timed out (system sets to zero before each I/O request).
- Unit = Last sub-channel addressed.
- Channel = I/O select code for device (lower number if a multi-board interface).

ISTA1

- AV = I/O controller availability indicator:
  - 0 = available for use.
  - 1 = disabled (down); for UP/DOWN status of LU see ISTA3.
  - 2 = busy (currently in operation).
  - 3 = waiting for an available DMA channel.
- EQUIP. TYPE CODE = type of device. When this number is linked with "DVR." it identifies the device's software driver routine:
  - 00 to 07_8 = paper tape devices (or system control devices)
  - 00 = teleprinter (or system keyboard control device)
  - 01 = photo-reader
  - 02 = paper tape punch
  - 05 subchannel 0 = interactive keyboard device (or system keyboard control devices)
    - subchannel 1,2 = HP mini-cartridge device
    - subchannel 3 = graphics
    - subchannel 4 = aux. printer
  - 07 multipoint driver
  - 10 to 17 = unit record devices
  - 11 = card reader
  - 12 = line printer
  - 15 = mark sense card reader
  - 20 to 37 = magnetic tape/mass storage devices
  - 31 = 7900 moving head disc
  - 32 = 7905 moving head disc
  - 33 = flexible disc
  - 40 to 77 = instruments
- STATUS = the actual physical status or simulated status at the end of each operation. For paper tape devices, two status conditions are simulated: Bit 5 = 1 means end-of-tape on input, or tape supply low on output.

Table 2-8. Device Status Table (EQT Word 5)

Device/Status	7	6	5	4	3	2	1	0
DVR05,DVA05 262X Terminal 263X Terminal 264X Terminal	BF		CD	DNR*	PE*		TEN	
Cartridge Tape Unit	EOF	TLP	EOT	RE	LCA	CWP	EOD	CNI/ DB
DVA12 2607 Line Printer		TOF		ID	PSE	OL		
2610/14 Line Printer		TOF		ID	SSE	PO		
2613/17/18 Line Printer		TOF		ID	ON	NR	V9	V12
2631 Line Printer		TOF		BR	ON	PO		
DVR12 2607A Line Printer	TOF	DM	ON	RX			APE	
DVR23 7970 Magnetic Tape	EOF	ST	EOT	TE	I/OR	NW	TE	OL
DVR31 7900 Disc Drive	--	NR	EOT	AE	FC	SC	DE	EE
DVR32/DVA32 7905/06/20/25 Disc Drives	PS	FS	HF	FC	SC	NR	DB	EE
DVM33 CS80 Disc Drives	WP	RER/ UD	EOF/ EOV	UN	FA	NR	CHE	SE

Device Status Table Key

AE = Address Error
APE = Auto Page Eject
BF = Buffer Flushed
BR = Buffer Ready
CD = Control-D Entered
CHE = Channel Error
CNI = Cartridge Not Inserted
CWP = Cartridge Write Protected
DB = Device Busy
DE = Data Error
DM = Demand (1=idle)
DNR = Data Set Not Ready
EE = Error Exists
EOD = End-of-Data
EOF = End-of-File
EOT = End-of-Tape
EOV = End-of-Volume
FA = Drive or Controller Fault
FC = Flagged Track
FS = Driver Format Switch is Set
HF = Hardware Fault
ID = Idle
I/OR= I/O Rejected
LCA = Last Command Aborted
NR = Not Ready
NW = No Write (Ring Missing or Rewinding)
OL = Off-Line
ON = On-Line
PE = Parity Error
PO = Paper Out
PS = Protect Switch Set
PSE = Print Switch Enabled
RD = Release Drive
RE = Read Error
RER = Recoverable Error
RX = Ready (0=Power On)
SC = Seek Check
SE = Severe Error
SSE = Start Switch Enabled
ST = Start of Tape
TE = Timing Error
TEN = Terminal Enabled
TLP = Tape at Load Point
TOF = Top-of-Form
UD = Unrecoverable Data
UN = Unitialized Media
V9 = VFU Channel 9
V12 = VFU Channel 12
WP = Write protect

* These apply only to DVA05 over modem.

Partition Status — EXEC 25

Requests system to return status information about a specified memory partition.

```
CALL EXEC(ICODE,IPART,IPAGE,INPGS,IPST)
                        _____ _____ ____
```

ICODE - Request code. 25 = partition status.

IPART - Number of partition that information is being requested on.

IPAGE - Starting page number of partiton.

INPGS - Number of pages in partition (including base page).

IPST - Partition status word.

COMMENTS:

PARTITION STATUS WORD (IPST). The format of IPST is as follows:

```
    (RS) (RT) (M) (S) (C) (E)
    ---------------------------------------------------
    | 15| 14| 13| 12| 11| 10| XX| XX| 7| 6| 5| 4| 3| 2| 1| 0|
    ---------------------------------------------------
                                    |                       |
                                    | --ID segment number-- |
```

The meaning of the IPST fields are summarized below:

* bit 15 (RS). Set to "1" if partition is reserved.
* bit 14 (RT). Set to "1" if partition is a real-time partition.
* bit 13 (M). Set to "1" if partition is a mother partition.
* bit 12 (S). Set to "1" if partition is a sub-partition of a mother partition.

* bit 11 (C). Set to "1" if chain-mode is in effect, i.e., subpartition is linked to an active mother partition.

* bit 10 (E). Set to "1" if partition is a shareable EMA data partition or is part of a shareable EMA partition (i.e., subpartition of a mother that is a shareable EMA partition).

* bit 9-8. Not used.

* bit 7-0. ID segment number. Set to the ordinal number (index into keyword block) of the ID segment for the program that occupies the partition. Set to zero if partition is not occupied.

A- AND B-REGISTER RETURNS---For successful calls the A-Register contents will be meaningless and the B-Register will be unchanged. For unsuccessful calls, error information is returned (refer to "EXEC Call Error Returns").

If the partition number (IPART) is illegal, -1 will be returned in INPGS and zero will be returned in IPAGE. For more information on partitions, refer to Chapter 1, "Memory Management".

Note that the number of pages and the starting page number of all partitions can be obtained by running the utility program WHZAT described in the RTE-6/VM Utility Programs Reference Manual.

EXAMPLE: Obtain size information on all the partitions in the system. Print results as a utility report on session console (LU 1).

```
      PROGRAM PUTIL
        .
        .
      IPART=1
   50 CALL EXEC(25,IPART,IPAGE,INPGS,IPST) !GET THE PARTITION STATUS
      IF(INPGS.EQ.-1)GO TO 150          !CHECK FOR THE LAST PARTITION
      WRITE(1,100)IPART,IPAGE,INPGS
  100 FORMAT(" PARTITION NUMBER ",I2," STARTS AT PAGE ",I4,
     *" AND IS ",I2," PAGES LONG ")
      IPART=IPART+1
      GO TO 50
  150 CONTINUE
        .
        .
      END
```

Memory Size — EXEC 26

Returns the memory limits of the partition that the calling program is currently executing in.

```
CALL EXEC(ICODE,IFAW,ILMEM,INPGS[,IMAP])
          ----  -----  -----  ----
```

ICODE - Request code. 26 = memory size.

IFAW - Address of first available word behind the calling program, e.g., last word of program, plus length of segment space, plus 1.

ILMEM - Number of words available between the last word of the program and the last word of the program's address space.

INPGS - Length of the partition (including base page) that program currently resides in.

IMAP - 32-word array into which a copy of the currently enabled user map is returned.

COMMENTS:

PARAMETER RELATIONSHIPS---The system calculates ILMEM by subtracting IFAW from the address of the last word of the program's logical address space. A program's logical address space is determined at load-time and can be equal to the program's size (default) or greater than the program's size (using LOADR or MLLDR size-override option). Furthermore, the partition that the program resides in may be greater than or equal to the program's logical address space.

For EMA programs, ILMEM corresponds to the number of words between IFAW and the beginning of MSEG (refer to Chapter 4, "VMA and EMA Programming").

Figure 2-1 illustrates the relationships between the EXEC 26 returned parameter values.

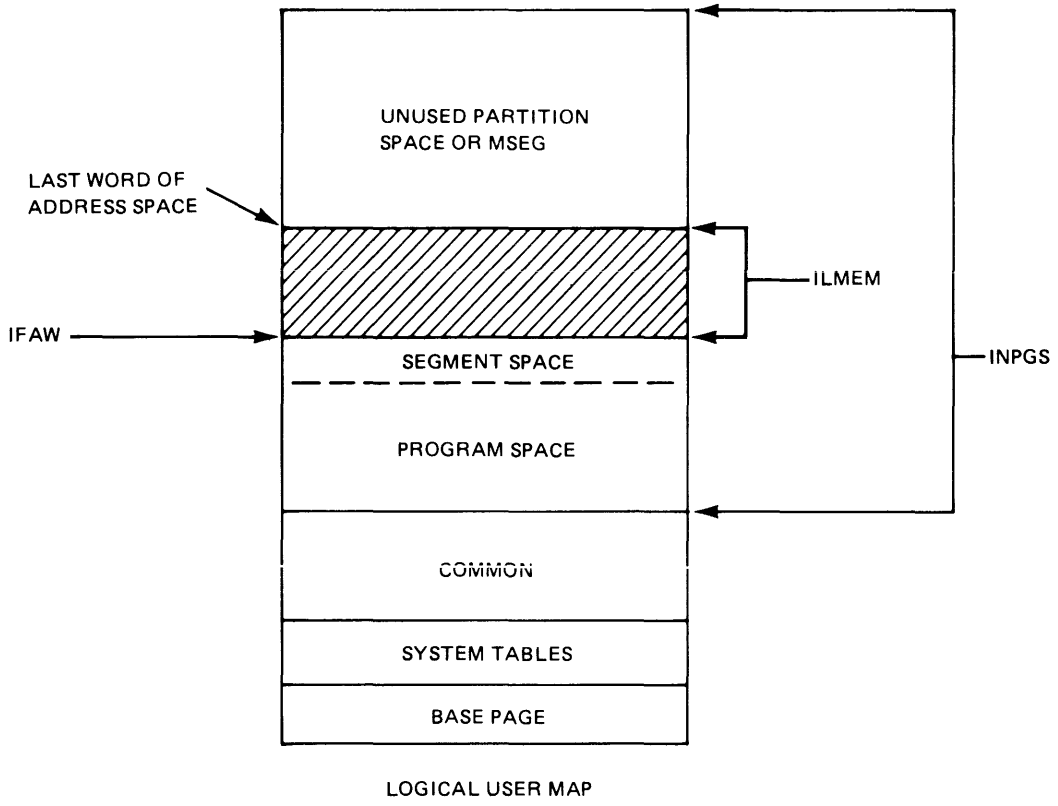


Figure 2-1. EXEC 26 Parameter Relationships.

For MLS programs the values of IFAW and ILMEM can change depending on the disc and/or memory-resident nodes which are currently mapped in the segment space. Note that the last word of the address can be calculated as (IFAW+ILMEM-1).

A- AND B-REGISTER RETURNS---For successful EXEC 26 calls, the returned A-Register contents will be meaningless and the B-Register contents will be unchanged. For unsuccessful calls, error information is returned (refer to "EXEC Call Error Returns").

EXAMPLE: FORTRAN program makes an EXEC 26 call to determine the size of the unused portion of its logical address space. The size and the address of the first word of the space are passed to a user-written Macroassembler routine (USER) that uses the space for temporary data storage.

```
PROGRAM DYALC
 .
 .
ICODE=26
CALL EXEC(ICODE,IFAW,ILMEM,INPGS) !GET THE MEMORY LIMITS
 .                                !OF THE CALLING PROGRAM
 .
CALL USER(ILMEM,IFAW,..)
 .
 .
```

Disc Track Management EXEC Calls

The Disc Track Management EXEC calls allow the user to allocate disc tracks for data storage and release disc tracks when they are no longer required. After allocation, the disc tracks can be accessed using EXEC Read and Write calls (refer to "Standard EXEC I/O Calls"). The tracks are allocated on the system or auxiliary cartridges (LU 2 or 3). The tracks allocated or released are logical tracks. For MAC and ICD discs a logical track is identical to a physical track. For CS80 discs, a logical track is a set of contiguous blocks which may or may not align with a physical track.

Disc tracks can be allocated as either local or global tracks. Locally allocated tracks can only be written to, or released by, the program that they are allocated to; they can be read by any program. Globally allocated tracks can be read from, written to, or released by any program.

When tracks are allocated, the system makes an entry in the system table called the Track Assignment Table (TAT). For a local allocation, the TAT entries corresponding to the allocated tracks will contain the ID segment address of the "owning" program. For global allocations, the TAT Entries will contain 077777 (octal). TAT entries for available tracks (released or never assigned) will contain zero. The TAT can be examined using the LGTAT utility described in the RTE-6/VM Utility Programs Reference Manual.

When tracks are allocated, they are supplied in units of complete tracks. i.e., one track is the minimum that can be allocated and no fraction of a track can be allocated. Tracks are also allocated on a contiguous basis. If six tracks are requested, six adjacent tracks will be allocated and they will be on the same cartridge.

Automatic track switching is provided. The user can write or read a record that crosses track boundaries, while the driver handles the track switching.

The EXEC calls included in this section are listed below in the order of their presentation:

* EXEC 4 or 15 - Disc Track Allocation

* EXEC 5 or 16 - Disc Track Release

Disc Track Allocation — EXEC 4 or 15

Allocate contiguous disc tracks for use by a single program (locally) or by multiple programs (globally).

```
CALL EXEC(ICODE,ITRAK,ISTRK,IDISC,ISECT)
          -----  -----  -----  -----  -----
```

ICODE - Request code. 4 = local allocation , 15 = global allocation.

ITRAK - Number of contiguous tracks requested. Program is suspended if tracks are not available; rescheduled when tracks become available. If bit 15 is set (ITRAK=ITRAK+100000B), the program is not suspended if tracks are not available.

ISTRK - Starting track number; returned by system. Set to -1 if tracks are not available.

IDISC - LU number of disc cartridge where tracks were allocated (2 or 3); returned by the system.

ISECT - Number of 64 word sectors per track; returned by the system.

COMMENTS:

A- AND B-REGISTER RETURNS---For successful completion of the call, the contents of the A- and B-Registers are meaningless. For unsuccessful calls, error information is returned (refer to "EXEC Call Error Returns").

EXAMPLE---Allocate two contiguous disc tracks globally and write 128 words into the sixth and seventh sector of the second track. If tracks are not available, do not suspend program; forget about write and continue processing. Check for errors after each call. Place location information in COMMON for use by other programs.

```
      PROGRAM DSCWT
      DIMENSION IBUFF(128)
      COMMON IDISC,ISTRK
         .
         .
      ICODE=15+100000B                  !ALLOCATE TWO CONTIGUOUS
      ITRAK=2+100000B                     !DISC TRACKS GLOBALLY
      CALL EXEC(ICODE,ITRAK,ISTRK,IDISC,ISECT,*100)
   10 IF(ISTRK.EQ.-1) GO TO 200 !CHECK IF TRACKS ARE AVAILABLE
         .
         .
      place data in IBUFF
         .
      ICODE=2+100000B
      ICNWD=IDISC
      ILEN=128
      IOP1=ISTRK+1           !WRITE THE BUFFER IN THE ALLOCATED
      IOP2=6                      !TRACKS USING AN EXEC 2 CALL.
      CALL EXEC(ICODE,ICNWD,IBUFF,ILEN,IOP1,IOP2,*100)
   20 CONTINUE
         .
         .
  100 CALL ABREG(IA,IB)
      error processing
         .
         .
  200 CONTINUE
         .
         .
      END
```

Disc Track Release — EXEC 5 or 16

Release contiguous disc tracks that were previously allocated either locally or globally.

```
CALL EXEC(ICODE,ITRAK[,ISTRK][,IDISC])
```

ICODE - Request code. 5 = release local tracks, 16 = release global tracks.

ITRAK - Number of contiguous disc tracks to be released. If ICODE = 5, ITRAK can be set to -1 and all tracks allocated locally to the calling program will be released.

ISTRK - Starting track number of tracks to be released. Not used if ICODE = 5 and ITRAK = -1.

IDISC - Disc LU (2 or 3) where tracks to be released reside. Not used if ICODE = 5 and ITRAK = -1.

COMMENTS:

A- AND B-REGISTER RETURNS---For successful calls with ICODE = 5, A- and B-Register returns are meaningless. If ICODE = 16, the B-Register return is meaningless and the A-Register return can be interpreted as follows:

* A = 0. Tracks have been released.

* A = -1. No tracks have been released. This indicates that at least one of the tracks was in use, i.e., a program was queued to read or write on one of the tracks at the time of the release request.

* A = -2. No tracks have been released. This indicates that at least one of the tracks to be released was not assigned globally.

Error information is returned in the A- and B-Registers for unsuccessful calls (refer to "EXEC Call Error Returns").

If a program aborts with tracks allocated locally to itself, the tracks are released automatically by the system. Global tracks will not be released; this is the user's responsibility.

The local track release function (ICODE=5) can also be accomplished interactively using the RT command described in the RTE-6/VM Terminal Users' Reference Manual.

EXAMPLE---Read the data placed on disc in the previous example. Release the tracks and verify that they were released. Check for EXEC errors. Obtain the track number and disc LU from COMMON.

```
      PROGRAM DSCRD
      DIMENSION IBUFF(128)
      COMMON ISTRK,IDISC
         .
         .
      ICODE=1+100000B              !READ FROM THE ALLOCATED DISC
      ICNWD=IDISC                   !TRACKS USING THE PARAMETERS
      ILEN=128                                !PASSED IN COMMON.
      IOP1=ISTRK+1
      IOP2=6
      CALL EXEC(ICODE,ICNWD,IBUFF,ILEN,IOP1,IOP2,*100)
   10 CONTINUE
         .
         .
      ICODE=16+100000B             !DEALLOCATE THE GLOBAL TRACKS
      ITRAK=2
      CALL EXEC(ICODE,ITRAK,ISTRK,IDISC,*100)
   20 CALL ABREG(IA,IB)
      IF(IA.EQ.0) GO TO 200      !CHECK IF TRACKS WERE RELEASED
      tracks-not-released processing
         .
         .
  100 CALL ABREG(IA,IB)
      error processing
         .
         .
         .
  200 CONTINUE
         .
         .
         .
      END
```

Chapter 3
File Management Via FMP

Introduction

File management is performed through program calls to the File Management Package (FMP) library and by interactive operator commands to the program FMGR or CI. The FMP calls mainly control input to and output from disc files or peripheral devices treated as files. The file management capability is increased by using FMGR or CI for interactive program development, disc cartridge and volume manipulation, and batch job control.

This section describes the FMP calls available under RTE-6/VM. For more details on FMGR operator commands refer to the RTE-6/VM Terminal User's Reference Manual. For more details on CI commands refer to the CI Users Manual.

The services available to the user via FMP Calls for FMGR are:

* FILE DEFINITION
* FILE ACCESS
* FILE POSITIONING
* SPECIAL PURPOSE CALLS

NOTE: THIS CHAPTER ONLY pertains to File Management in the FMGR environment although many concepts are the same as the CI file system. Refer to the CI Users Manual for additional information and features of the CI file system.

Files

Files are a collection of information logically organized into records. They can be stored on disc or they may reference non-disc peripheral devices by name. The information in files may be programs or the data used by the programs. Data may be in binary or ASCII code. Programs may be in the form of ASCII source code, or binary code in either relocatable or absolute form. Programs may also be in memory-image form, a form used by RTE for programs ready to be executed.

Eight file types are defined by the file system. Additional types may be defined by the user. Only the first four types differ in format; all subsequent types differ only in the type of data the file system expects the file to contain. The file types may be divided into three categories as shown in Table 3-1. The first category contains one type, type zero. This type includes all non-disc devices defined as files and accessible by name. The second category contains two file types, types 1 and 2. These fixed-length record files are used for quick random access. The remaining file types all belong to the third category of files with variable-length records designed for sequential access. All files may be extended automatically as needed.

Table 3-1. Categories of File Types

CATEGORY	TYPE	DESCRIPTION
Control	0	Non-disc files.
Fixed-length, random access	1	Fixed-length 128 word record files
	2	Fixed user-defined record length files
Variable-length sequential access	3	Variable-length records; any data type
	4	Source program file; ASCII
	5	Object program file; relocatable binary
	6	Executable program file; memory-image
	7	Absolute binary
	8-32767	User-defined data format

FMP file types are summarized below:

Type 0 Files

Type 0 files are used to reference non-disc devices by name. They afford a measure of device independence in that the standard file commands can be used to control the device. A directory entry is made for the device as if it were a file. The File Directory entry for a file of this type contains special entries that specify logical unit number and the operations allowed on the particular device. A type 0 file is created and purged with a FMGR command, not with an FMP call.

Type 1 Files

Type 1 files have fixed-length records of 128 words and are extendable. Because the File Management Package transfers data to and from disc in 128-word blocks, this file type allows direct access between disc and the user's buffer area in his program, thereby eliminating the need to go through an intermediate packing buffer (the Data Control Block). As a result, type 1 files have the fastest transfer rate. Any disc file (not type 0), may be opened and accessed as a type 1 file in order to take advantage of the faster transfer rate. However, if the file being transferred has less than 128-word logical records, the user must be able to recognize where the records begin and end within the 128 words. If the records are longer, the user must be able to work with part of a record at a time. The end-of-file is defined to be the last word of the last block allocated to the file.

Type 2 Files

The record lengths of type 2 files are fixed, but the length is defined by the user at file creation. Type 2 files are also extendable. Like type 1 files, the end-of-file is defined to be the last word of the last block allocated to the file. Only one logical record is transferred at a time, but unlike type 1 files, the transfer must go through a packing buffer (the Data Control Block). For this reason, type 2 files have a slower transfer rate than type 1 files.

Type 3 Files

These files have variable-length records, are extendable, and may contain data, source code, relocatable or absolute binary code. Only one logical record is transferred at a time and the transfer must be made through the packing buffer (Data Control Block). The first and last words of each record as written on disc always contain the number of words in the record (minus the two length words). A zero-length record, consisting of two zero words, can be used to separate groups of records into sub-files. The end-of-file is marked by a -1 as the first length word in the next record. Words following the end-of-file are undefined. However, FMP can write records beyond the end-of-file by replacing the end-of-file with a new record followed by an end-of-file mark.

Type 4 Files

This file type is the same as type 3, except the file system expects these files to contain ASCII data. Typically, source program files are type 4.

Type 5 Files

Same as type 3 files, except the file system assumes type 5 files contain relocatable binary code. Typically, compiler object program files are type 5. The FMGR expects relocatable binary code in these files and always checks the type of code when transferring data to or from these files.

Type 6 Files

This type file is the same as a type 3 file, except the system assumes it contains a program in memory-image format that is ready to run. Type 6 files are created by the FMGR SP command which copies a program stored on the system track into a type 6 file on the FMP tracks of any disc cartridge. These files are always accessed by the File Management Package as type 1 files. The first two sectors of a type 6 file are used to record ID segment information for the program. As a result, this file type can be used for programs that do not have a permanent ID segment.

Type 7 Files

Same as type 3 files, except the system expects type 7 files to contain absolute binary code.

File Types Greater Than 7

Same as type 3, but the content is user-defined. FMP does no special processing based on file type for types greater than 7. For instance, any checksums must be specifically requested. Content is also user-defined; it may be source, relocatable binary, memory-image format, etc.

File Access

Type 1 and type 2 files contain fixed-length records which makes it possible to calculate the position of a desired record (random access). On the other hand, type 3 files and above contain variable-length records, so the system must access the disc at least once, and in some cases several times, in order to position to the desired record location. For this reason, random access takes longer for file types greater than type 2.

File Extents

All disc files can be automatically extended whenever a write request points to a location beyond the range of the currently allocated file. The extent is created by FMP with the same name and size as the main file, and access continues. FMP numbers each extent starting with one. The extent number and location is kept in the file directory entry for the extent. When a file with extents is referenced by its file name, any extents are provided automatically. A file may have up to 255 extents.

For type 3 and above files, all extents are sequential. That is, the main is created first followed by extent 1,2,3 and so on. The end of the file is signified by an EOF mark. At close, the extents may be removed to provide more disc space or they may be retained.

For type 1 and 2 files, extents may be created sequentially or randomly. That is, the main is created first and sequential writes to the file may cause sequential extents to be created. However, a random write to a record that would fall into an extent will cause that extent to be created whether or not the extent immediately preceding it exists. Extents which have not been allocated are called "missing extents".

For example, assume a type 1 file with a main file size of 10 blocks is created. Writing to record 11 of this file will cause extent 1 to be created. Now the main and only extent 1 exist. A write to record 31 will cause extent 3 to be created, not extent 2. Now the file consists only of the main, extent 1, and extent 3.

Externally, type 1 and type 2 files are treated as logically contiguous. As in the example above, writing 1408 words starting at record 31 causes 10 records (1280 words) to be written in extent 3, and one record in extent 4. Also, it is possible to read a record from an extent that does not exist as long as that record is not beyond the last word of the last extent in the file. No disc access is performed; the user's buffer will be zero-filled. For example, a read to record 22 of the file created above will return with no error, the return length will be 128 words, the user's buffer will be zero-filled, and the next record is set to 23.

The physical end-of-file for type 1 and type 2 files is defined as the last word of the last extent allocated to the file.

When a type 1 or type 2 file is copied using the FMGR ST and DU commands, the resultant file may occupy more disc space than the original file occupied. This is because records from missing extents are physically read into the destination file. This feature is especially important when compacting the extents of the file with the ST and DU commands. The resultant file must be physically contiguous as well as logically contiguous to guarantee the same displacement. The CO command preserves missing extents.

File Updates

Files may be opened for access in either update or non-update mode. Update mode is used to access existing files that are to be modified in a random manner. Non-update mode is used to access files in a sequential manner or to enter the original data into a file. Update, mode insures that only the data being modified is changed in the file.

In update mode, the entire data block containing the record is automatically read into the DCB before the record is modified. After modification, the entire block is written to the disc. This is done to insure that the DCB always contains the unmodified as well as the modified data, thereby guaranteeing restoration of the block to the disc.

Reading or positioning a file is not affected by the update or non-update modes of access.

Cartridges

Files managed by the File Management Package, whether they are program files, data files, or spool files, are kept on FMP disc cartridges. An FMP disc cartridge is a logical entity that may correspond directly to a disc platter or may be a subdivision of the disc platter. Most cartridges cross platter boundaries.

Each cartridge is defined as a contiguous block of tracks, and is assigned a logical unit number (LU). A Cartridge Reference Number (CRN) or the LU may be used to reference the cartridge. Files on the same cartridge must have unique names. Duplicate names may be used as long as the duplicates are on separate cartridges, so the file can be uniquely identified by its name and a cartridge identifier.

Files are stored in a consecutive fashion on a disc cartridge. Generally, files are located contiguously, but if the file has extents, the extents may not be contiguous. User files begin in the lowest numbered track and work up. Directory entries for user files begin in the highest numbered track and work down. Removable cartridges containing FMP files are interchangeable between drives of the same type within any system, provided that logical track 0 refers to the same physical track on every disc unit. (Refer to Figure 3-1 for an illustration of disc organization using one cartridge on the system disc starting at the first FMP track, and one on a peripheral disc starting at track 0.)

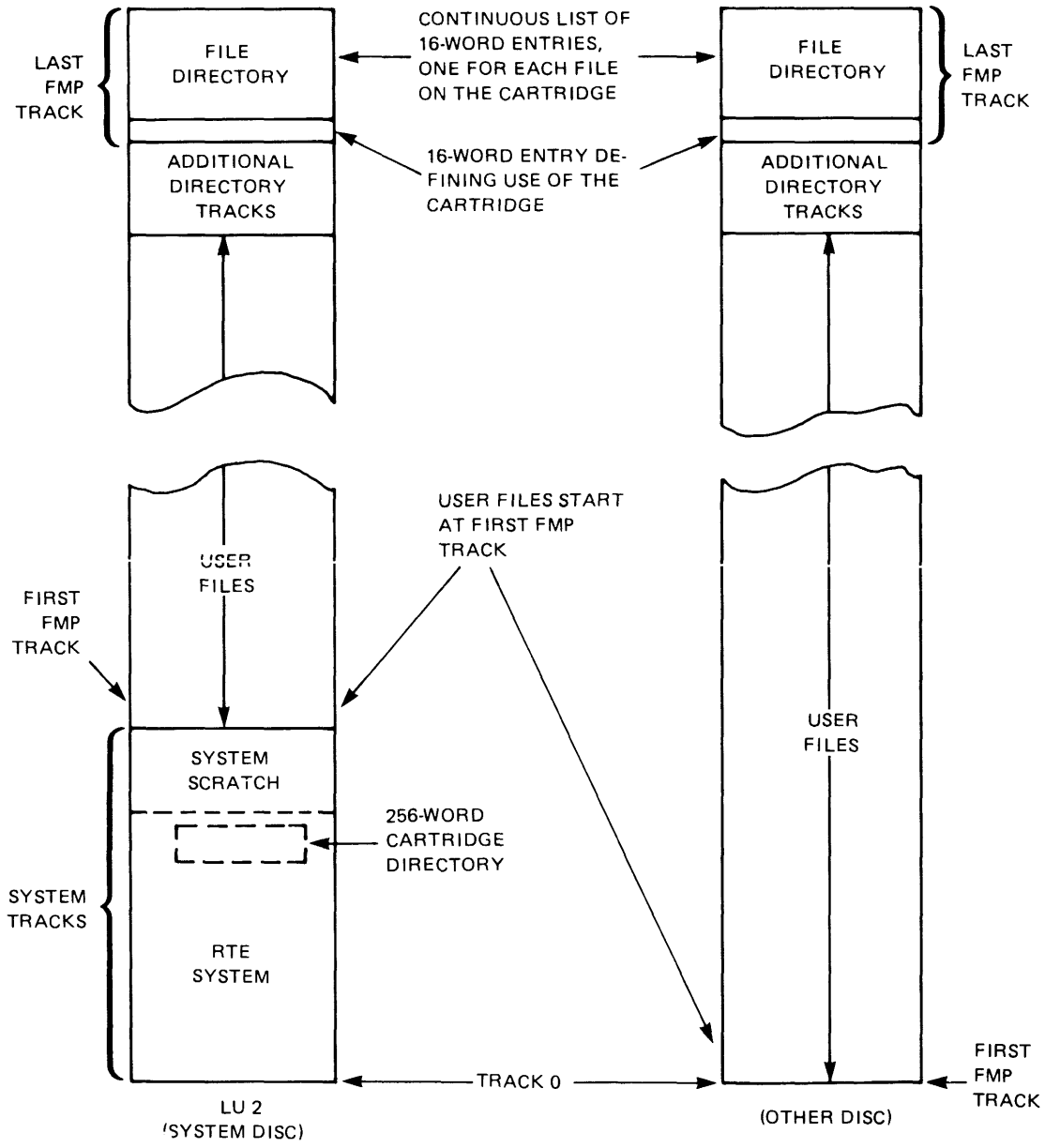


Figure 3-1. Disc Cartridge Organization

At cartridge initialization, the number of directory tracks for that cartridge is specified. The first cartridge track must be assigned at initialization; the number of sectors per track may be specified at this time, but is supplied by FMP as a default if not.

Files may cross track boundaries, but may not cross cartridge boundaries. Files are subject to being moved whenever a cartridge is packed. This causes files to be relocatable within a cartridge and no absolute file addresses should be kept in any file or program.

Files always start on even sector boundaries and all accesses are multiples of 128 words addressed to even sectors.

Disc errors are passed back to the user for action. Error codes are printed on the system log device when using the FMGR operator commands, or passed to the user program when calling a File Management Package library routine. You may report bad tracks to the system through the FMGR IN command. Bad tracks discovered by the system result in an error return to the calling program.

Cartridge and File Directories

Two directory types are maintained by the file system; the FMP cartridge directory on the system disc, and the file directory on each cartridge. The program D.RTR updates and maintains the directories.

The cartridge directory is a master index to all mounted FMP cartridges. It is maintained in the system area of LU 2 (refer to Figure 3-1). Its length is two blocks, and it has an entry for all currently mounted cartridges. The directory has room to describe up to 63 cartridges using four words for each.

A file directory, maintained on each cartridge, contains information on each file on that particular cartridge. Each directory starts in sector 0 of the last track available to FMP. The first 16-word entry in this directory contains label and track information for the cartridge itself. Each subsequent 16-word entry has information on a user file. The last entry is followed by a zero word. When a file is purged, the first word in the directory entry for the file is set to -1 to indicate that the directory entry is to be ignored. When the cartridge is packed, the directory entry for any purged file is cleared and the cartridge area where the file was located is overwritten by non-purged files wherever possible.

The formats of the cartridge and file directory entries are shown in Appendix G.

File Security

In addition to the security features provided through the Session Monitor (refer to the discussion below), the File Management Package provides two additional security levels: file system security and individual file security.

During FMP initialization, a master security code is assigned to the file system. If the code is zero, no security is provided. If non-zero, the master security code must be known in order to get directory listings that include the specific file security codes or to initialize cartridges in a non-session environment.

Each file also has a security code. This code may be zero, positive, or negative. A zero code allows the file to be opened to any caller (who has access to the disc cartridge containing that file) with no restrictions; in effect this code provides zero security. A positive code restricts writing on the file but not reading; that is, a user who does not know the code may open the file for read only, but may not write on the file. A negative code restricts all access to the file; this code must be specified in order to open a file protected by it. An attempt to open a file so protected without the correct security code results in an error message.

Cartridges in the Session Environment

In the session environment, cartridge security is provided by identifying them as to whom they are mounted.

When a disc cartridge is mounted to the system, an ID is put in the cartridge list entry which identifies the category of the cartridge and to whom the cartridge was mounted.

There are four categories of cartridges in the session environment as follows:

1. System cartridges.
2. Private cartridges.
3. Group cartridges.
4. Non-session cartridges.

System cartridges are those cartridges that only the System Manager can mount or dismount. The cartridge where the system tracks reside must be defined during system generation as LU 2. An auxiliary system cartridge, defined as LU 3 at system generation, can also be defined.

LU 2 and LU 3 are read-only cartridges for session users. Only the System Manager and non-session users can write on these cartridges. LU 2 contains:

1. Operating System
2. System Library
3. System Scratch Tracks
4. Cartridge Directory
5. FMP Tracks

LU 3 can be defined if more room is required for additional system tracks.

In addition to LU 2 and LU 3, system global cartridges can be defined which provide both read and write access to all users on the system. Global cartridges are cartridges other than LU 2 or 3 that are mounted by the System Manager.

Private cartridges are those that are accessible only to the user who mounts them to his session and the System Manager (who has access to all cartridges in the system).

Group cartridges are those that are accessible only to members of a group who have it mounted to their sessions and the System Manager.

Non-session cartridges are only accessible to users operating in a non-session environment or through the system console (when not enabled for session). Non-session cartridges are those mounted to a user not operating under session control. The System Manager also has access to non-session cartridges.

When the session system is initialized, the System Manager can create a spare cartridge pool from which session users can mount additional private or group cartridges to their session. The cartridges "taken" from the pool are typically used for temporary storage and are released back to the pool by the user when they are no longer needed.

The Session Monitor provides session users with protection for their mounted cartridges by restricting cartridge access to only those cartridges mounted in the user's Session Control Block (SCB). Whenever a cartridge is specified in a call, FMP checks that the cartridge is mounted not only to the system, but is also mounted to the user's SCB (refer to Appendix H for the SCB format).

There are some exceptions to the above check:

1. The System Manager is allowed access to all cartridges mounted to the system by using the SL command described in the RTE-6/VM Terminal User's Reference Manual.

2. Some internal subsystems need to have, and are given, access to all cartridges mounted to the system.

3. Procedure files can be set up by the System Manager on LU 2 or LU 3 and may be run by any session user. The commands in these procedure files are not subject to capability checking or cartridge access restrictions.

4. User message files may reside on LU 2 and LU 3. Session users will be able to open, read, and write into these files via the ME and SM commands described in the RTE-6/VM Terminal User's Reference Manual.

5. Read only access will be allowed on LU 2 and LU 3, and read/write access will be allowed on system global cartridges when operating under session.

6. A user can store and purge type 6 files on LU 2 or LU 3.

When accessing a file, if a particular cartridge is not specified, the user's private discs are searched in the order that they are mounted in the system cartridge directory. The user's group cartridges are then searched in the order that they appear in the system cartridge directory. Finally, system cartridges are searched in the order that they appear in the directory.

To summarize, session users have access to their private and group cartridges, system global cartridges, and have restricted access to the system cartridges LU 2 and LU 3.

Non-session users have access to non-session cartridges and unrestricted access to all system cartridges. Non-session users do not have access to cartridges mounted to session users.

Note that cartridges are mounted and dismounted using the MC, AC, and DC commands described in the RTE-6/VM Terminal User's Reference Manual.

FMP Calls

The FMP program calls provide an interface between programs and the File Management utility routines. With these calls, the user can open, close, read from, and write to files. In addition, the calls can be used to create or purge disc files, position either disc or non-disc files and directly control non-disc files.

Table 3-2 lists all the FMP calls according to general function and indicates the status, before and after the call, of the Data Control Block (DCB). It also indicates when and if the file directory is accessed by the call.

The Data Control Block

The Data Control Block (DCB) is a block of words defined within your program that acts as an interface between the program and the File Management Package. You must supply one DCB for each open file. The DCB is an array which contains control information for the file including the file name, type, size, and location on disc if the file is a disc file. In addition, it acts as a buffer for the physical transfer of data between a file and your program. The DCB is used to:

* Avoid directory access for file information

* Keep track of current record position in file

* Provide a buffer for transfer of data between a file and the program.

Once a file is open, the DCB is referenced for file information and the file name is no longer needed or used in FMP calls.

Each DCB is an array with a minimum of 144 words. The first 16 words are a control block to provide all the file information required by the FMP calls. The remaining words are a packing buffer used for the transfer of data in blocks of 128 words. The 16-word control area is maintained and used only by FMP and must not be modified directly. Refer to Appendix G for the format of the DCB.

Table 3-2. FMP Call Summary

CATEGORY	ROUTINE	FUNCTION	DCB STATUS		DIRECTORY ACCESS
			AT ENTRY	AT RETURN	
File Definition	CREAT ECREA	Enter file in directory; open exclusively for update.	CBO	OPNX	YES
	CRETS	Enter scratch file in directory; open exclusively for update.	CBO	OPNX	YES
	PURGE	Close file and remove from directory.	CBO	CLOS	YES
	OPEN OPENF	Open file.	CBO	OPN	YES
	CLOSE ECLOS	Close file.	MBO	CLOS	YES
File Access	READF EREAD	Transfer record from file to user buffer.	MBO	OPN	EXTENTS
	WRITF EWRIT	Transfer record from user buffer to file.	MBO	OPN	EXTENTS
File Position	LOCF ELOCF	Retrieve current position and status of open file.	MBO	OPN	NO
	APOSN EAPOS	Position disc file to a particular record.	MBO	OPN	EXTENTS
	POSNT EPOSN	Position disc or non-disc file relative to current record.	MBO	OPN	EXTENTS
	RWNDF	Position file to first record.	MBO	OPN	EXTENTS
Special Purpose Routines	FCONT	Specify control functions for non-disc file.	MBO	OPN	NO
	FSTAT	Retrieve contents of cartridge directory.	—	—	—
	IDCBS	Retrieve actual size of DCB buffer used by FMP.	MBO	OPN	NO
	NAMF	Rename existing file.	CBO	CLOS	NO
	POST	Write DCB buffer to disc.	MBO	OPN	NO

Legend:

CBO	Can be open: DCB can be assigned to open file; that file will be closed and, in case of CREAT and OPEN, file specified in call will be opened.
MBO	Must be open: DCB must be assigned to open file.
OPN	Open: File assigned to DCB is opened or is left open.
OPNX	Open: New file is assigned to DCB and is opened exclusively for update.
CLOS	Closed: File assigned to DCB is closed; DCB is available for other use.
EXTENTS	Directory is accessed only if call changed extents.

A packing buffer of 128 words is the minimum that can be specified. The buffer may be as large as available memory, but any file can be accessed with the minimum 128-word buffer regardless of the buffer size specified at creation.

Data Transfer

In addition to the DCB, another buffer must be defined in your program for transferring individual records. This buffer, the user buffer, is where a record to be written is specified and into which a record is read. The relation between the user buffer, the DCB, and a disc file is illustrated in Figure 3-2.

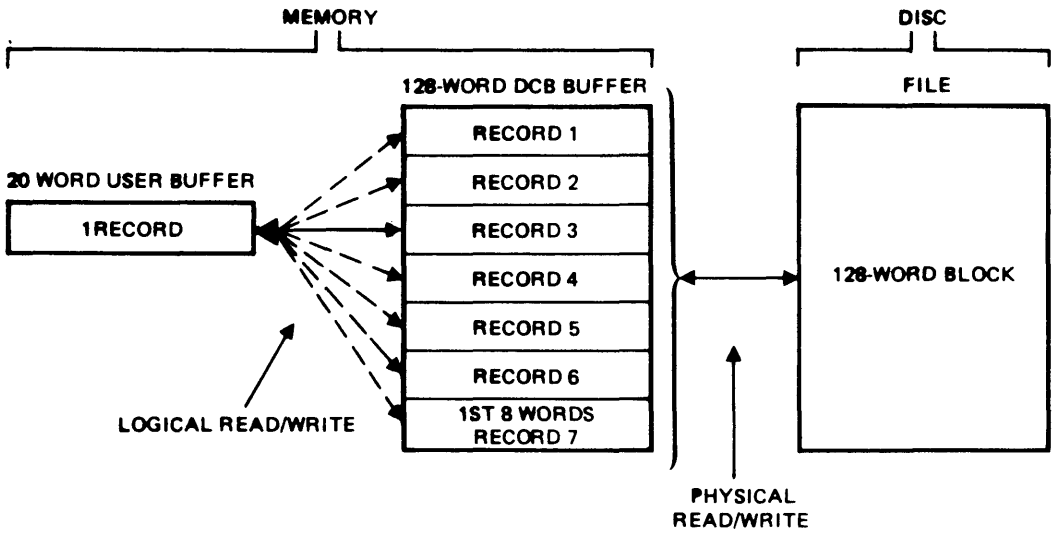


Figure 3-2. Sequential Transfer Between Disc File and Buffers

Each call to read or write a record transfers one record between the user buffer and the DCB packing buffer. This type of transfer within memory is known as a logical read or write.

A physical read or write transfers a 128-word block between the disc file and the DCB packing buffer. A physical write is performed automatically when the packing buffer is full, when the file is closed, or when a specific request is made with the POST FMP call. On a read request, a block of data is physically read into the packing buffer from the disc only if the requested record is not already in that buffer. Any time a record being read or written is not wholly contained in the packing buffer (refer to record 7 in Figure 3-2), then the File Management Package reads or writes blocks until the entire record has been transferred.

When type 2 files are accessed randomly, the process is essentially the same as the sequential access illustrated in Figure 3-2 except that physical transfers may be more frequent since successive references are less likely to be to records in the same block in the packing buffer.

Since each record in a type 1 file is 128 words, the intermediate transferred to the DCB packing buffer is omitted and each record is transferred directly between the user buffer and the file as illustrated in Figure 3-3. This type of access is the most efficient. A full 144 word DCB must still be specified in the user program.

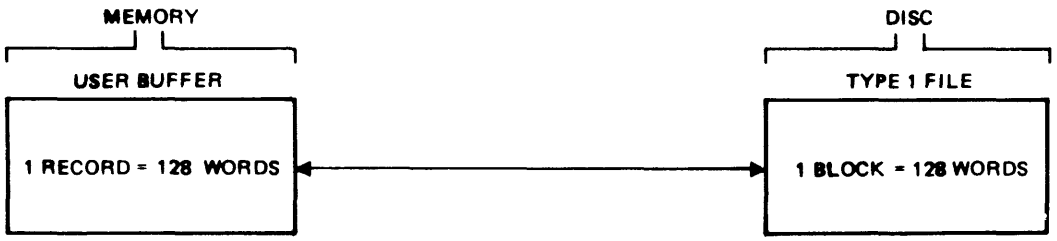


Figure 3-3. Data Transfer With Type 1 Files

Type 0 (non-disc) files also bypass the DCB during transfers. Records in these files are written or read directly to or from the device identified as a type 0 file. A specific number of words, rather than a record, is the unit transferred by a read or write request to this file type. The transfer can thus be tailored to the particular record length of the device.

FMP Call Formats

When a calling sequence is encountered during execution of a user program, the File Management Package executes the call according to the value of the parameters in the calling sequence, and/or returns information to areas defined in the parameter list. The number, type, and meaning of the parameters depend on the subroutine and are described in detail in this section. Note that a VMA or EMA variable cannot be used as a parameter in a FMP call.

For FORTRAN, the general form of the calling sequence is:

```
CALL name(p1,p2, . . . ,pn)
```

where:

name is the subroutine name.

p1 through pn are the parameters; parameters name a real or an integer array or variable or, if the parameter is one-word, a value. Position determines parameter meaning.

For Pascal/1000, the general form of the calling sequence is:

```
        :
        p1
        :
        pn
PROCEDURE name;
  (VAR parm1 :typex;
         :
   VAR parmn :typex);
EXTERNAL;
  :
name (p1...pn);
```

where:

name is the subroutine name; it must always be defined as an external procedure.

p1 through pn are the parameters used in the subroutine call. These are defined as variables.

parm1 through parmn are the corresponding parameters in the procedure. All the parameters in the procedure must be a previously defined type.

NOTE: If FMP calls are used to access a Pascal file, the following should be taken into consideration.

- For the DCB parameter of the FMP call, specify the Pascal file variable.
- Pascal-managed file status information is not updated using FMP calls directly. This status information is used by Pascal file-handling routines.

Refer to the Pascal/1000 Reference Manual for more details.

For Macro/1000, the general form of the calling sequence is:

```
      EXT  name
       :
       JSB  name
       DEF  RTN (or * +(n+1) where n is the number of parameters)
       DEF  p1
       DEF  p2
       :
       DEF  pn
RTN    (return location)
```

where:

name is the subroutine name; it must always be defined as an external with an EXT statement before it is called.

RTN is the label of the location to which the subroutine returns upon completion; it must always follow the last parameter in the calling sequence.

p1 through pn are the parameters; each parameter names an integer array or variable.

Common Parameters

Parameters used frequently in FMP calls are described here and under "Optional Parameters".

IDCB

This parameter specifies the array used as a DCB. It must be at least 144 words, 16 words for file control information and 128 for the minimum packing buffer. For faster processing, a larger buffer can be specified. In general, the larger the usable buffer, the faster the transfer rate. For example, a usable buffer of 256 words (IDCB=272 words) will nearly double the transfer rate for sequential accesses.

While a file may be created with a large DCB buffer, it may be accessed with any packing buffer that is at least 128 words long (IDCB=144 words). All transfers of data use the full actual buffer size. This size is always 128, or a multiple of 128, words. The actual size used is determined from the size requested for the DCB in conjunction with the file size in the following manner:

* If the requested buffer size is greater than or equal to the file size, FMP allocates an actual DCB equal to the file size +16 control words.

* If the requested buffer size is less than the file size, FMP determines the actual DCB buffer size according to the following rules:

  - It must be a multiple of 128 words.

  - It must be less than or equal to the size specified for IDCB by the user.

  - It can be evenly and exactly divided into the total file size.

To illustrate, Table 3-3 shows the relation between file size, requested packing buffer size and the actual size assigned by FMP for two file sizes, one with factors and the other a prime number. File size is given in 128-word blocks for convenience.

When a file is opened or created via an FMP call and the DCB is currently open to a file, the current file associated with the DCB is closed and the file specified in the FMP call is opened.

Table 3-3. Relation of Actual to Requested Packing Buffer Size

FILE SIZE IN BLOCKS	REQUESTED DCB BUFFER* BLOCKS	WORDS	ACTUAL DCB BUFFER BLOCKS	WORDS
10	1	128	1	128
	2	256	2	256
	3	384	"	"
	4	512	"	"
	5	640	5	640
	6	768	"	"
	7	896	"	"
	8	1024	"	"
	9	1152	"	"
	10	1280	10	1280
13	1	128	1	128
	2	256	1	128
	.	.	"	"
	.	.	"	"
	.	.	"	"
	12	1536	1	128
	13	1664	13	1664

*16 words must be added to the buffer size when dimensioning the Data Control Block array (IDCB).

A call to routine IDCBS may be made to determine the actual size of the DCB (actual buffer + 16 control words).

IERR

When an error occurs during a subroutine call, a negative error code is returned in this parameter and usually in the A-Register. (A list of the FMP error codes and their meaning is included in Appendix A). For successful OPEN or OPENF calls, the file type is returned as a positive integer in IERR; for successful CREAT calls the number of disc sectors used is returned in IERR as a positive integer. When using the CLOSE or ECLOS calls, always check the IERR parameter. In calls where IERR is the last parameter, it is optional. It should be omitted only if errors are checked in the A-Register. Negative error codes allow for easy error checking and error checking should not be omitted as a general practice.

INAM

For calls requiring a file name, the name is specified in this 3-word integer array. File names are six ASCII characters and must conform to the following rules:

* only upper case ASCII characters (41B-137B) may be used.
* first character must not be a digit.
* if less than six characters, must be padded with trailing blanks.
* beginning or embedded blanks are not allowed.
* plus (+), minus (-), colon (:), or comma (,) are not allowed.
* Cannot be a FORTRAN 77 character string.

Duplicate file names are not allowed on the same cartridge.

IBUF

This array is the user buffer and is included in the calls that transfer records. It should be as long as the longest record to be transferred. The record to be written must be in this array; when a record is read it is placed in this array.

Optional Parameters

Most subroutines have one or more optional parameters. They always appear at the end of the calling sequence and may be omitted only from the end. If an optional parameter is needed that is preceded by other unnecessary optional parameters, all the parameters up to the desired parameter must be included. Unused optional parameters should be set to zero.

ISC

The file security code is specified in ISC. It can be a positive or negative integer or two ASCII characters representing a positive integer. When characters are used, FMP converts them to their integer equivalent. If omitted, ISC is set to zero.

ISC considerations are summarized below:

CREATION	FILE PROTECTION	FILE REFERENCE
ISC = 0	unprotected	ISC = any value for any access
ISC = + n	write protected	ISC = +n or −n to write or purge ISC = any value to open or read
ISC = − n	read/write protected	ISC = −n for any access

ICR

This parameter is specified to restrict the file search to a particular cartridge or LU. The search is for space if the call is CREAT or ECREA, or for a file name in other calls. ICR may be a positive or negative integer. If omitted, it is set to zero.

If ICR = 0 The file search is not restricted to a particular cartridge. A CREAT or ECREA call locates the file on the first cartridge with enough room; other calls using ICR search the cartridges in the following order:

1) Private cartridges as they appear in the user's cartridge list.

2) Group cartridges as they appear in the user's cartridge list.

3) System cartridges as they appear in the user's cartridge list.

>0 File search is restricted to the cartridge identified by the CRN, an identifier assigned to all cartridges in the system. A list of CRNs can be obtained with the FSTAT call or the FMGR CL command.

<0 File search is restricted to the cartridge associated with the LU. To illustrate, if ICR is -14, the file search is restricted to LU 14.

If ICR specifies a cartridge not mounted to the user's SCB, an error is returned.

IDCBZ

When a DCB larger than 144 words is specified in parameter IDCB, then parameter IDCBZ must also be specified. It informs FMP of the number of words available in the DCB packing buffer for data transfer. Any positive number can be specified; the actual usable buffer is always determined by FMP as described in the discussion of the IDCB parameter. This size is never larger than the size specified in IDCBZ, but it may be smaller. For example, if IDCBZ is less than 256 (any value between 1 and 255), then 128 words are used for the DCB packing buffer (144 for the entire DCB).

Normally, you will specify IDCBZ as 16 words less than the array size specified for IDCB.

FMP Call Description Conventions

In the subsections that follow, certain conventions are used to describe FMP calls. These conventions are summarized below:

* Parameters that are underlined, such as:

```
CALL OPEN(IP1,IP2,IP3)
          ---  ---
```

  have values returned by the system, e.g., the value is not supplied by the user.

* Parameters that are double underlined, such as:

```
CALL OPEN(IP1,IP2,IP3)
          ===
```

  have values that are supplied by the system in some cases and user-supplied in other cases. The comments associated with the call description should be consulted for details concerning their use.

* Parameters enclosed in square brackets, such as:

```
CALL OPEN(IP1,IP2[,IP3])
```

  are optional.

* Parameters enclosed in angle brackets, such as:

```
CALL OPEN(IP1,IP2<,IP3>)
```

  are optional in some cases and required in others. The comments associated with the call description should be consulted for details concerning their use.

* Parameters with no qualifiers, i.e., square brackets, angle brackets, or underlines, are required and their value is supplied by the user.

* All FMP call descriptions use the FORTRAN subroutine call format. If desired, the description of FMP call general formats included at the beginning of this section can be consulted to convert the calls to Pascal/1000 or Macro/1000 format.

File Definition FMP Calls

A file is defined in terms of its name, size, type, and where it is located. In addition, a security code can be assigned to a file; restricting access by unauthorized programs. A file is defined when it is created using the CREAT or ECREA call. When a file is created, an entry is made for the file in the file directory (refer to Appendix G for the format of the file directory).

Once defined, the file may be opened for access by any program using the proper security code in an OPEN or OPENF call. When a file is opened, necessary information (location, size, etc.) is transferred from the file directory to the control words of the DCB associated with the file, thereby making a logical connection between the file name in the directory and the DCB.

When access to a file is no longer necessary, the logical connection between the file directory and the DCB is removed by closing the file using a CLOSE or ECLOS call. Closing a file frees the DCB for other uses, but the file remains defined in the file directory.

The PURGE call is used to flag the directory entry associated with a file so that the file is no longer defined. Note that the PURGE call closes the file before performing the purge operation.

Scratch files provide temporary disc space for any program needing scratch area during its execution. When a process is completed and the caller has no further use for these files, PURGE may be used to eliminate them. If the information on a scratch file should be saved, either copy it to another file or rename the file using the NAMF call before closing the file. Scratch files (which have not been renamed to a standard filename) are automatically purged when the cartridge containing them is searched by the File Management System. The only other difference using these files is the create function CRETS. This call will create a scratch file using the number passed by the caller and unique information about the program. Up to 100 scratch files are allowed per program. When the file is created, it is opened exlusively to the caller and the name of the file is returned. Refer to the section on the CRETS call for more information concerning scratch files.

The File Definition FMP calls are listed below in the order of their presentation:

* CREAT,ECREA
* CRETS
* OPEN,OPENF
* CLOSE,ECLOS
* PURGE

CREAT and ECREA Calls

A call to CREAT or ECREA creates a disc file by making an entry in the File Directory and allocating disc space for the file. The ECREA call performs the same function as the CREAT call except that larger files can be created. CREAT can define files up to 16383 blocks in size, ECREA can define files up to 32767x128 blocks in size.

Following the creation call, the file is left open in update mode for exclusive use of the calling program. To put the file in a different mode, or for non-exclusive use, an additional call must be made to the OPEN or OPENF routines described later in this section.

Note that the CREAT and ECREA calls can only create disc files. To create non-disc files (type 0), the FMGR CR command must be used (refer to the RTE-6/VM Terminal User's Reference Manual).

```
CALL CREAT (IDCB,IERR,INAM,ISIZE,ITYPE[,ISC][,ICR][,IDCBZ])
   or              ----
CALL ECREA (IDCB,IERR,INAM,ISIZE,ITYPE[,ISC][,ICR][,IDCBZ]
                ----                                [,JSIZE])
                                                      -----
```

IDCB -- DCB. An array of 144 +n words where n is positive or zero.

IERR -- Error return. A one-word variable in which a negative error code is returned. If no error occurs, IERR is set to the number of 64-word sectors in the created file (for CREAT); for ECREA calls, IERR is indeterminant.

INAM -- File name. 3-word array containing ASCII file name.

ISIZE -- File size. For CREAT call, a 2-word array with the number of blocks in the first word (must be <=16,383); record length (in words) in the second word (used only for type 2 files).
For ECREA call, a 2-entry array with each entry being a double word integer. The first entry specifies the file size in blocks (up to 32767 x 128 blocks). The second entry specifies the record length in words (used for type 2 files only).

ITYPE -- File type. A one-word variable between 1 and 32767. Types 1 - 7 are FMP defined; the rest are user-defined special purpose files.

ISC -- Security code. An optional one-word variable. It can be a positive or negative integer, or two ASCII characters. If 0 (or omitted), the file is not protected. If positive, the file is write protected only. If negative, the file is write and read protected.

ICR -- Cartridge identifier.An optional one-word variable. If positive, specifies a CRN. If negative specifies an LU. If 0 (or omitted), first available cartridge with enough room will be used. If in session environment, the ICR must specify a cartridge mounted to the user's session. If -32768, specifies that the file is to be created on the scratch cartridge (defined by the VL operator command). If the requested cartridge is full, a FMGR-033 error is returned.

IDCBZ -- Packing buffer size. An optional one-word variable set to the number of words in the packing buffer if more than 128 are requested. If omitted, FMP assumes DCB size (control words + packing buffer) is 144 words regardless of the IDCB dimension.

JSIZE -- Created file size. Used with ECREA call; optional double word parameter in which the actual file size created (in sectors) is returned if the call was successful.

COMMENTS:

FILE SIZE---Records are addressed by number, therefore, the number of records contained in a file must not exceed the maximum accessible by the file access calls (described later in this section).

If a file is to be accessed with the standard access calls (READF, WRITF, etc.), the number of records contained in a file must not exceed (2**15)-1. If a file is to be accessed with the extended access calls (EREAD, EWRIT, etc), the number of records must not exceed (2**31)-1.

The number of records contained in type 1 or type 2 file can be determined as follows:

$$\text{number of records} = \frac{\text{words in file (blocks in ISIZE x 128)}}{\text{record length in words}}$$

When the exact size is not known, the rest of the cartridge may be allocated by setting ISIZE to -1 (valid only for files of type 3 or greater). The unused area may be returned to the system by truncating the file when it is closed (refer to the CLOSE and ECLOS call described later). For CREAT call the rest of the cartridge but not more than 16383 blocks will be allocated. For ECREA the rest of the cartridge but not more than 32767 x 128 blocks will be allocated.

The ECREA call allocates file space in block increments. After 16,383 blocks have been allocated, then ECREA begins allocating file space in 128 block increments. If a file size (ISIZE) is specified that is not evenly divisible by 128, a full 128 block is allocated to accommodate the remainder.

FILE TYPES -- FMP defined file types are as follows:

Type 1. Fixed-length, 128-word records; random access.

Type 2. Fixed-length, user-defined record lengths; random access.

Type 3 (and greater). Variable-length records; sequential access.

Type 4. Source program

Type 5. Relocatable program

Type 6. Memory-image program

Type 7. Absolute binary program

File types greater than 7 are user defined but are treated by FMP as type 3. Any special processing based on file type is not provided as a default, it must be specified.

When any file of type 3 or greater is created, FMP writes an EOF mark at the beginning of the file. As records are written to the file, the EOF is moved automatically to follow the last record.

When a file of type 1 or 2 is created, extents are not automatically created unless the file is opened via the OPEN FMP call with the EX bit (bit 5) set.

Further details on file types are given at the beginning of this section.

EXAMPLE 1: Create a type 2 file called FIX with 100 blocks, 62 words per record, security code AB, and a DCB packing buffer of 128 words:

```
DIMENSION NAM1(3),ISIZE(2),IDCB1(144)       !16 CONTROL WORDS +
                                              !128-WORD BUFFER
DATA NAM1/6HFIX   /
ISIZE(1)=100                                 !NUMBER OF BLOCKS
ISIZE(2)=62                                       !RECORD SIZE
ITYPE=2                                             !FILE TYPE
ISC=2HAB                                     !SECURITY CODE AB
.
CALL CREAT(IDCB1,IERR,NAM1,ISIZE,ITYPE,ISC)        !CREATE FILE
IF (IERR .LT. 0) GO TO 900           !PROCESS ANY ERRORS AT 900
CONTINUE
:
```

EXAMPLE 2: Create a type 3 file called PROG1, use rest of cartridge, no security code, using a 256 word DCB packing buffer, and locate it on LU 14.

```
DIMENSION NAM2(3),ISIZE(2),IDCB2(272)         !16-CONTROL WORDS
                                              + 256-WORD BUFFER
DATA NAM2/6HPROG1 /
ISIZE(1)=-1         !FILE USES REST OF CARTRIDGE, RECORD LENGTH
                                                !UNSPECIFIED.
ICR=-14                                  !FILE LOCATED ON LU 14.
ITYPE=3                                    !FILE TYPE IS 3
IDCBZ=256                                 !DCB BUFFER SIZE
:
CALL CREAT(IDCB2,IERR,NAM2,ISIZE,ITYPE,0,ICR,IDCBZ)
IF (IERR .LT. 0) GO TO 900                        !ERROR CHECK
CONTINUE
:
```

Another method is to use literals for all one-word variables:

```
DIMENSION NAM2(3),ISIZE(2),IDCB2(272)
DATA NAM2/6HPROG  /
.
.
CALL CREAT(IDCB2,IERR,NAM2,-1,3,0,-14,256)
IF (IERR .LT. 0) GO TO 900
CONTINUE
.
.
```

Care should be taken when literals are used as parameters since this practice can result in problems if the values are changed by the call routines.

EXAMPLE 3: Create a type 2 file (PROGD) that is 76,864 blocks in size. Each record is 64 words long; DCB packing buffer is 128 words.

```
DIMENSION NAM3(3),ISIZE(4),IDCB(144)
DATA NAM3/6HPROGD /
ISIZE(1) = 1
ISIZE(2) = 11328
ISIZE(3) = 0
ISIZE(4) = 64
ITYPE = 2
.
.
CALL ECREA(IDCB,IERR,NAM3,ISIZE,ITYPE,0,0,IDCBZ,JSIZE)
IF (IERR .LT. 0) GO TO 900
CONTINUE
.
.
```

The double-integer parameter in the above example is manipulated "by-hand". Note that 76864 - 65536 = 11328. Since 65536 = 2**16, bit 16 of the double-word integer must be set. This is equivalent to bit 0 of ISIZE(1). ISIZE(2) is then set to 11328 with the net result being a double-integer representation of 76864 in ISIZE(1) and ISIZE(2).

CRETS Call

A call to CRETS creates a temporary or scratch disc file by making an entry in the File Directory and allocating disc space for the file. CRETS can define files up to 32767x128 blocks in size.

Following the CRETS call, the file is left open in update mode for exclusive use of the calling program. The file is also given a unique file name.

Upon program termination the scratch file is purged, and the disc space, if possible, is returned to the system. If the information in the file is to be made available for normal use, either copy it to another file or change its name using the NAMF call.

Note upon termination the file created by the program is purged unless renamed to a standard FMGR file name. If the file is renamed, the user must purge the file when it is no longer needed. The File Management System purges all scratch files as it searches directories and finds unopen scratch files.

```
CALL CRETS (IDCB,IERR,INUM,INAM[,ISIZE][,ITYPE][,ISC][,ICR]
                    ----      ----
            [,IDCBZ][,JSIZE])
                     -----
```

IDCB -- DCB. An array of 144 +n words where n is positive or zero.

IERR -- Error return. A one-word variable in which a negative error code is returned.

INUM -- Scratch file number. A one-word integer between 0 and 99.

INAM -- File name created; 3-word array containing ASCII file name.

ISIZE -- File size. A 2-entry array (or 4 word array) with each entry being a double word integer. The first entry specifies the file size in blocks (up to 32767 x 128 blocks). The second entry specifies the record length in words (used for type 2 files only). Default is 24 blocks.

CONTINUED NEXT PAGE

CONTINUED FROM PREVIOUS PAGE

ITYPE -- File type. A one-word variable between 1 and 32767; types 1-7 are FMP defined, the rest are user-defined special purpose files. Default is type 3.

ISC -- Security code. An optional one-word variable. It can be a positive or negative integer, or two ASCII characters. If 0 (or omitted), the file is not protected. If positive, the file is write protected only. If negative, the file is write and read protected.

ICR -- Cartridge identifier. An optional one-word variable. If positive, specifies a CRN. If negative, specifies an LU. If 0 (or omitted), first available cartridge with enough room will be used. If in session environment, the ICR must specify a cartridge mounted to the user's session. If -32768, specifies that the file is to be created on the scratch cartridge (defined by the VL operator command). If no room exists on the requested cartridge a FMGR-033 error is returned.

IDCBZ -- Packing buffer size. An optional one-word variable set to the number of words in the packing buffer if more than 128 are requested. If omitted, FMP assumes DCB size (control words + packing buffer) is 144 words regardless of the IDCB dimension.

JSIZE -- Created file size. Optional double word parameter in which the actual file size created (in sectors) is returned if the call was successful.

COMMENTS

FILE SIZE---Records are addressed by number, therefore, the number of records contained in a file must not exceed the maximum accessible using the file access calls described later in this section.

If a file is to be accessed with the standard access calls (READF, WRITF, etc), the number of records contained in a file must not exceed (2**15)-1. If a file is to be accessed with the extended access calls (EREAD, EWRIT, etc), the number of records must not exceed (2**31)-1.

The number of records contained in a type 1 or type 2 file can be determined as follows:

$$\text{number of records} = \frac{\text{words in file (blocks in ISIZE x 128)}}{\text{record length in words (ISIZE(3) and ISIZE(4))}}$$

When the exact size is not known, the rest of the cartridge may by allocated by setting ISIZE to -1. The unused area may be returned to the system by truncating the file when it is closed (refer to the CLOSE and ECLOS call described later). The rest of the cartridge but not more than 32767 x 128 blocks will be allocated.

The CRETS call allocates file space in block increments. After 16,383 blocks have been allocated, then CRETS starts allocating file space in 128 block increments. If a file size (ISIZE) is specified that is not evenly divisible by 128, a full 128 block is allocated to accommodate the remainder.

Users of the scratch files are encouraged to use the default size. The reason for this is that when a file is created requesting a specific size, a check is made in the directory for a purged file of exactly the same size. If such a file is found, the new file replaces it; if such a file is not found, the new file is placed after all other files on the cartridge. Therefore, if most programs request the same size (that is, the default size) scratch files, the space on the disc is used more efficiently and packing the disc is not required as often.

Further details on file types are given at the beginning of this section.

EXAMPLE 1: Create a type 3 scratch file with 24 blocks, security code HP, CRN AB, and a DCB packing buffer of 128 words. At the end of the program, the file will be renamed to DATA05.

```
    :
IMPLICIT INTEGER (A-Z)
DIMENSION IDCB (144), NAME (3), NUNAME (3), ISIZE (4)
DATA ISIZE /0,24,0,0/, NUNAME/6HDATA05/
ITYPE = 3
ISC = 2HHP
ICRN = 2HAB
INUM = 23
    :
CALL CRETS (IDCB,IERR,INUM,NAME,ISIZE,ITYPE,ISC,ICRN) !CREATE FILE
IF (IERR.LT.0) GO TO 500
    :
CALL NAMF (IDCB,IERR,NAME,NUNAME,ISC,ICRN)  !RENAME FILE TO DATA05
IF (IERR.LT.0) GO TO 700
    :
```

OPEN and OPENF Calls

These routines open a file for access and position it at the first record. The file must have been created prior to the OPENF or OPEN call. The file opened may be a disc or non-disc (type 0) file. Type 0 files may be opened with a function code specified at creation or the original function code may be overridden.

Files may be opened for exclusive use of the calling program or for non-exclusive use of up to seven programs. A file may be opened for update or for standard sequential write operations.

By using the OPENF call, an LU can be passed in the first word of the INAM parameter causing a DCB to be created to allow type 0 access to the LU. No type 0 file is necessary and none is created by the call.

```
CALL OPEN(IDCB,IERR,INAM[,IOPTN][,ISC][,ICR][,IDCBZ])
      or       ----
CALL OPENF(IDCB,IERR,INAM[,IOPTN][,ISC][,ICR][,IDCBZ])
                 ----
```

IDCB -- DCB. An array of 144+n words where n is positive or zero.

IERR -- Error return. A one-word variable that a negative error code is returned to for unsuccessful calls. File type is returned for successful calls.

INAM -- File name or LU. This is either a three-word array containing the ASCII file name (for OPEN or OPENF) or an integer containing an LU (OPENF only).

IOPTN -- Open options. An optional one-word variable set to octal value to specify non-standard opens. If omitted or set to zero, the file is opened by default as follows:

* Exclusive use - only the calling program can access the file.
* Standard sequential output - each record is written following the last, destroying any data beyond the record being written.
* File type defined for file at creation is used for access.

To open a file with other options, set IOPTN as described below under OPEN OPTIONS.

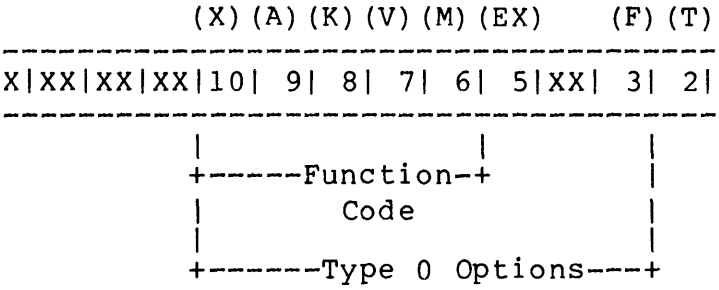
ISC -- Security code. An optional one-word variable. If 0 (or omitted), the file is not protected. If positive, the file is write protected only. If negative, the file is write and read protected.

ICR -- Cartridge identifier. Optional 1-word variable. If set, FMP searches that cartridge for file. If omitted, it searches cartridges in the user's cartridge list and opens the first file found with specified name.

IDCBZ -- DCB buffer size. Optional 1-word variable. Set to number of words in DCB packing buffer if larger than 128. If omitted, FMP assumes DCB size (control words + buffer) is 144 words regardless of IDCB dimensions.

COMMENTS:

OPEN OPTIONS -- The IOPTN parameter is defined as follows:



The following bits may be set for any file type:

E (bit 0) = 0 File opened exclusively for this program. In an OPENF of an LU or OPEN of a type 0 file, if the open is exclusive and the device is not interactive, the device will be locked. The device becomes unlocked when the DCB is closed.

1 File may be shared by up to seven programs

U (bit 1) = 0 File opened for standard (non-update) write (disc files only)

1 File opened for update

T (bit 2) = 0 Use file type defined at creation (disc files only)

1 File type is forced to type 1

EX (bit 5) = 0 File extents cannot be accessed (type 1 and 2 files only).

1 File extents are created and may be accessed. (Type 1 and 2 files only.)

The following bits are used for type 0 files only (they are ignored when opening other file types):

F (bit 3) = 0 Use function code defined at creation (refer to the FMGR CR command described in the RTE-6/VM Terminal User's Reference Manual). If an LU is used in an OPENF call, the defaults are as shown in Table 3-4.

1 Use function code defined in bits 6-10 of IOPTN.

Bits 6-10 correspond exactly to the function code used for the Read or Write EXEC call. These are driver dependent and the appropriate Driver Reference Manual should be consulted for more information.

Table 3-4. OPENF Defaults

DEVICE	DEVICE TYPE	EOF CODE	SPACING	READ/WRITE
Reader or Punch	2	LE	BO	BO
Minicartridge	5 subchannel 1,2	EO	BO	BO
Mag/Tape,mass storage,other	>17	EO	BO	BO
All others	--	PA	BO	BO
Bit Bucket	--	PA	BO	WR

EOF CODE	SPACING	READ/WRITE
EO=subfunction 0100	FS=forward space	RE=read
LE=subfunction 1000	BS=back space	WR=write
PA=subfunction 1100	BO=both	BO=both

EXCLUSIVE OPEN---By default, a file is opened for exclusive use of the calling program. An exclusive open is granted to only one program at a time. If the call is rejected because the file is open to another program, the call must be made again; it is not stacked by FMP. Exclusive open is useful in order to prevent one or more programs from destructively interfering with each other.

NON-EXCLUSIVE OPEN---If more than one program needs to access the file, it should be opened non-exclusively by setting the IOPTN E bit. A non-exclusive open may be granted to as many as seven programs per file at one time. A non-exclusive open will not be granted if the file is already opened exclusively. Each time an open is requested for the file, all programs currently having the file open are checked. If any program is dormant, the file is closed to that program. That type of close does not free the DCB and does not post the contents of the DCB buffer to the file. Any open flag will also be cleared if it does not point to a valid ID segment.

UPDATE OPEN---In update mode, IOPTN U bit set, the block containing the record to be written is automatically read into the DCB before it is modified. This insures that existing records in the block will not be destroyed. This mode of open has no effect on reading or positioning, it is only necessary when writing to a file that already contains valid data, or when building a file in a random manner.

Update mode should be used to write to type 2 files. A type 2 file should be opened in standard mode (non-update) only when originally writing the file or adding to the end of the file, and then only if it is to be written sequentially.

Type 1 files, like type 2 files, are designed for random access with fixed-length records and the end-of-file in the last word of the last block; each record is the same length as the block transferred so that there is no danger of writing over existing records.

NOTE: A type 1 file that is opened in update mode and attempts to write past the current EOF returns a FMGR error from WRITF.

For type 3 and above files, update mode is not generally used; most writes are sequential with an end-of-file mark written after each record. These files should be opened for update only if a record previously written to the file is being modified. In this case, care must be taken not to change the length of the modified record. If it is changed, an FMGR-005 error is issued. Regardless of the mode of open (update or standard) a record written beyond the end-of-file replaces the end-of-file and is followed by a new end-of-file.

TYPE 1 ACCESS---Any file may be forced to type 1 access by setting the IOPTN T bit. Type 1 access is faster because it bypasses the DCB buffer and transfers a block of data directly to the user buffer defined as IBUF. The file type defined at creation is not affected; the file is treated as type 1 only for the duration of this open. The program is responsible for any packing or unpacking of records in files forced to type 1. For a type 1 or type 2 file to be extendable, bit 5 of the IOPTN parameter must be set.

OPEN FLAGS---Occasionally files are left open to a program upon termination. The operating system and the program D.RTR cooperate to close these files when necessary. A termination sequence counter is kept in the ID segment. Each time a program is terminated or removed from an ID segment, the termination counter in that ID segment is incremented. The open flag placed in a file's directory entry whenever a file is opened or created

contains both the ID segment number and the termination sequence counter. Whenever D.RTR finds an open flag whose termination counter is not current with the ID segment, or the program occupying the ID segment is dormant, it removes the flag.

EXAMPLE 1: Open a type 2 file named FIX for update in non-exclusive mode. The security code at creation was AB.

```
DIMENSION NAME(3),IDCB1(144)
DATA NAME/6HFIX   /
IOPTN=3B                          !SET BIT 1 FOR UPDATE AND BIT 0
ISC=2HAB                                    !SECURITY CODE AB
.
CALL OPEN(IDCB1,IERR,NAME,IOPTN,ISC)
IF (IERR .NE. 2) GO TO 900   !TEST FOR ERROR; FILE TYPE EXPECTED
CONTINUE
.
.
```

EXAMPLE 2: Open type 3 file (PROG1) with default options. The file is located on LU 14:

```
DIMENSION NAM(3),IDCB2(144)
DATA NAM/6HPROG1 /
.
.
CALL OPEN(IDCB2,IERR,NAM,0,0,-14)
IF (IERR .NE. 3) GO TO 900
CONTINUE
.
```

Although PROG1 was created with a DCB buffer of 256 words (refer to CREAT examples), it can be opened and accessed with the minimum DCB buffer of 128 words.

EXAMPLE 3: Open type 0 file PTAPE (created with CR command) and set option so that binary data is punched on paper tape without leader.

```
DIMENSION NAME(3),IDCB3(144)
DATA NAME/6HPTAPE /
IOPTN=2310B                    !SET M, V, AND X FOR "HONESTY MODE"
:
CALL OPEN(IDCB3,IERR,NAME,IOPTN)
IF (IERR .NE. 0) GO TO 900
:
```

For a description of "honesty mode", refer to the DVR00 Programming and Reference Manual.

EXAMPLE 4: Use OPENF call to allow programmatic type 0 access to LU 6 (line printer). Use defaults for options (IOPTN).

```
DIMENSION NAM(3),IDCB(144)
NAM(1)=6
:
CALL OPENF(IDCB,IERR,NAM)
IF (IERR .LT. 0) GO TO 900
CONTINUE
:
```

CLOSE and ECLOS Calls

A call to CLOSE or ECLOS removes the logical connection between the DCB and the File Directory entry associated with the file. The DCB is freed for association with other files; the entry for the file is maintained in the File Directory.

A disc file opened for exclusive use of the calling program may be truncated freeing unused disc space. Up to 32767 blocks may be truncated with the CLOSE call. Up to 32767 x 128 blocks may be truncated with the ECLOS call.

```
CALL CLOSE(IDCB<,IERR>[,ITRUN])
                       ----
    or
CALL ECLOS(IDCB<,IERR>[,ITRUN])
                       ----
```

IDCB---DCB. An array of 144+n words, where n is positive or zero; previously associated with a file in a create or open FMP call.

IERR---Error return. A one-word variable in which a negative error code is returned if a truncation operation was unsuccessful; only required when ITRUN is specified.

ITRUN--Truncation word. For CLOSE call, optional one-word variable containing integer number of blocks to be deleted from the file at closing. If zero or omitted, the file is closed without truncation. If negative, only extents are truncated. ITRUN must be <= 32767 for CLOSE call.

For ECLOS call, ITRUN is an optional double-word variable specifying the number of blocks to be deleted from the file at closing. If zero or omitted, the file is closed without truncation. If negative, only extents are truncated. ITRUN must be <= (2**31)-1 for ECLOS call.

COMMENTS:

FILE TRUNCATION---When a file has been created with more blocks than are actually needed to accommodate the data in it, it can be truncated at closing to save disc space. A file may be truncated only if:

* the file is a disc file.
* the current position is in the main file, not in an extent.
* the file was opened with the correct security code.
* the file was opened for exclusive use of the calling program.
* the number of blocks deleted are less than or equal to the total number of blocks in the file.

If all these conditions are met, the value of ITRUN can be a:

* positive integer - to specify the number of blocks to be deleted from the end of the main file. Any extents are automatically truncated. If equal to the total number of blocks in the file, the file is purged.

* negative integer - to specify that any extents be deleted from the file; the main file is not affected.

If a main file larger than 16383 blocks is to be truncated (previously created with an ECREA call), it is done on a 128 block basis. For example, if 1 <= ITRUN <= 127, the file size is not changed. If 128 <= ITRUN <= 255, 128 blocks are deleted. If the file size is less than or equal to 16,383 blocks, the actual number of blocks specified in ITRUN are truncated.

The value of ITRUN (when positive) can be determined from information returned by a previous call to LOCF or ELOCF. Assuming the current position is at the end of the data in the file, the block number of the block containing the next record to be written or read (IRB+1) is subtracted from the total blocks with which the file was created (JSEC/2) and assigned to ITRUN. When negative, ITRUN can be any value. The number of extents need not be known. If the file is currently positioned in an extent, it can be re-positioned to the main file with a RWNDF call.

A zero value for ITRUN is exactly the same as omitting this parameter; a standard close is performed with no truncation.

If the file is write protected, and a truncation is requested, CLOSE will return with a -7 error (bad file security code or illegal write to LU 2 or 3). This error indicates that the truncation was not successful, but the file has been closed anyway.

EXAMPLE 1: Close file FIX (IDCB1) with no truncation; assume FIX has been opened:

```
DIMENSION IDCB1(144)          !FILE NAME ASSOCIATED WITH IDCB1
:
CALL CLOSE(IDCB1,IERR)
IF (IERR .LT. 0) GO TO 900                        !ERROR RETURN
CONTINUE
:
```

IDCB1 is freed for association with other files.

EXAMPLE 2: Close type 3 file PROG1 and truncate it to exactly the number of blocks used. Assume the file was written sequentially and the current location is at the end of the data in the file.

```
DIMENSION IDCB2(144)
:
CALL LOCF(IDCB2,IERR,I,IRB,I,JSEC)   !CALL LOCF FOR FILE LENGTH
ITRUN=(JSEC/2)-(IRB+1)                          !AND NEXT BLOCK
CALL CLOSE(IDCB2,IERR,ITRUN)     !IERR REQUIRED WITH TRUNCATION
IF (IERR .LT. 0) GO TO 900
:
```

Since JSEC contains the number of sectors it must be divided by 2 for the number of blocks. IRB is the next block number to be written to; a one must be added since blocks are numbered starting with 0.

EXAMPLE 3: Close the same file (PROG1), but delete only the extents; use the RWNDF call to insure that current position is in the main file.

```
      DIMENSION IDCB2(144)
      :
      IF(RWNDF(IDCB2))900,10,10
   10 CALL CLOSE(IDCB2,IERR,-1)      !ITRUN NEGATIVE TO DELETE ALL
      IF (IERR .LT. 0) GO TO 900                      !EXTENTS
      CONTINUE
      :
```

PURGE Call

A call to PURGE removes the named file from the system along with any extents associated with the file. Only disc files can be purged through a program call; non-disc files may be purged with the FMGR PU command (refer to RTE-6/VM Terminal User's Reference Manual). When a file is purged, the file directory entry is no longer available. If the file was open, it is closed freeing the DCB. A file that is open to any program other than the calling program cannot be purged until it is closed by the other program(s).

```
CALL PURGE(IDCB,IERR,INAM<,ISC><,ICR>)
           ----
```

IDCB---DCB. An array of 144+n words, where n is positive or zero.

IERR---Error return. A one-word variable that a negative error code is returned to for unsuccessful calls. Zero is returned for successful calls.

INAM---File name. A three-word array containing the ASCII name of the file to be purged.

ISC----Security code. An optional one-word variable; must be specified if file to be purged has a security code; may be omitted if file does not have a security code.

ICR----Cartridge identifier. Optional one-word variable. If specified FMP purges the named file on the specified cartridge. If omitted FMP searches the cartridge directory and purges the first file found in the user's disc addressing space with the specified name.

COMMENTS:

PURGING FILES OPENED NON-EXCLUSIVELY---When a file is opened to more than one user, each user should first close the file before it is purged to insure a successful purge. Only the calling program may purge the file without first closing it, and then, only if it is closed to all other programs.

A type 6 file cannot be purged if an ID segment pointing to the disc space occupied by the type 6 file exists (the file has been RP'ed).

RECOVERY OF FILE AREA FOLLOWING PURGE---The area on disc occupied by a purged file is returned to the file system automatically only if the purged file is the last file on the cartridge. If the file is the last file, all the area from the beginning of the file to the end of the cartridge is returned to the file system. If the file is followed by other files, the cartridge must be packed in order to recover the file space. Packing is accomplished with the FMGR PK command (refer to the RTE-6/VM Terminal User's Reference Manual).

Note that space from a purged file in the middle of the cartridge will be reused if a file is created with exactly the same size as the file that was purged.

EXAMPLE 1: Purge the type 2 file named FIX created with security code AB.

```
DIMENSION NAM1(3),IDCB1(144)
DATA NAM1/6HFIX   /
:
CALL PURGE(IDCB1,IERR,NAM1,2HAB)
IF (IERR .LT. 0) GO TO 900                    !PROCESS ERRORS
CONTINUE
:
```

EXAMPLE 2: Purge Type 3 file (PROG1) created without a security code on LU 14.

```
DIMENSION NAM2(3),IDCB2(144)
DATA NAM2/6HPROG1 /
:
CALL PURGE(IDCB2,IERR,NAM2,0,-14)
IF (IERR .LT. 0) GO TO 900
CONTINUE
:
```

File Access

Information in files is accessed with the READF, EREAD, WRITF, and EWRIT routines. Calls to these routines are basically the same whether the file is a device (type 0) or a disc file (type 1 and above). Whether reading or writing, you can specify that exactly one record is to be transferred or you may specify a particular number of words. In general, it is good practice to specify the number of words since this permits generality among file types.

The normal mode of access for files of type 3 and above is sequential. Such files are created with an end-of-file in the first record. The first record written overrides the end-of-file and a new end-of-file is written immediately following the record. As each subsequent record is written, the process is repeated so that the end-of-file always follows the last record written.

Variable-length records are assumed for these file types. For this reason it is necessary to specify the number of words when the record is written. On a read, if the number of words is not specified, FMP determines its length and reads exactly one record.

For file types 1 and 2, random access is the normal mode. No end-of-file is written. The end-of-file is calculated according to the file size defined at creation. Since each record is a fixed-length determined at creation, the file is easily positioned to a particular record. Generally, one record is written or read at a time, although more may be transferred when accessing a type 1 file.

When accessing a type 0 file, the number of words should be specified unless a zero-length record is to be read or written or a record skipped. End-of-file marks are not written automatically to type 0 files; you must specify the end-of-file.

The record formats for fixed-length and variable-length records is shown in Appendix I.

READF and EREAD Calls

These routines read a record from an open file into a user program's buffer. One full record or a specified number of words can be read. The record to be read may be the record at which the file is currently positioned or, for type 1 and 2 files, it may be any specified record. The READF call can specify a record number up to (2**15)-1. The EREAD call can specify a record number up to (2**31)-1.

```
CALL READF(IDCB,IERR,IBUF[,IL][,LEN][,NUM])
   or           ---- ----       ---
CALL EREAD(IDCB,IERR,IBUF[,IL][,LEN][,NUM])
             ---- ----       ---
```

IDCB---DCB. An array of 144+n words where n is positive or zero; previously specified in a create or open operation.

IERR---Error return. A one-word variable in which a negative error code is returned for unsuccessful calls. Zero is returned for successful calls.

IBUF---Data buffer. An array into which the requested data is placed by the system. It should be large enough to contain the largest record expected.

IL-----Data length requested. An optional one-word variable that specifies the number of words to be read. Refer to Table 3-5 for details.

LEN----Data length read. An optional one-word variable in which actual number of words read is returned. Set to -1 if end-of-file is read.

NUM----Record number. An optional one-word (for READF) or double-word variable (for EREAD) used to specify the record number to be read (if positive) or the number of records to backspace (if negative). Used only for type 1 or 2 files. If omitted, the record at the current position is read.

COMMENTS:

RELATION OF IL TO FILE TYPE---It is a good idea to specify IL for file type 0 and it does not hurt to specify it for other file types. If you do not know the length of a disc-file record, IL can be specified as the user buffer length to prevent reads beyond this area. If the record is shorter than IL, the exact record length is read for file types greater than type 1. Table 3-5 illustrates the effect of IL depending on file type.

Table 3-5. Effect of IL Parameter in READF

IL VALUE	FILE TYPE 0	FILE TYPE 1	FILE TYPE > 1
IL > 0	Up to IL words are read; if less than IL, file defined record length is read.	Exactly IL words are read; IL may be more or less than 128-word record.	Up to IL words are read; if less than IL, actual record length is read.
IL = 0 (not recommended)	Zero length record is read; usually record is skipped and counted as read.	No action. (Zero-length read, no position change.)	Record is skipped and counted as read.
IL omitted	Zero-length record is read; usually record is skipped and counted as read. (Not recommended.)	128-word record is read.	Actual record length is read.

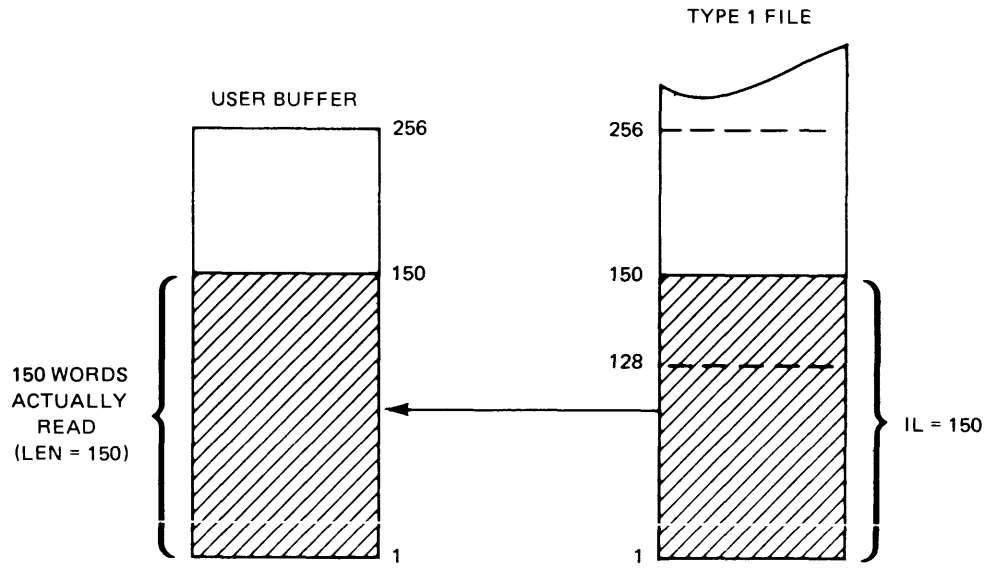


Figure 3-4. Read Type 1 File When IL Greater Than 128

As illustrated in Figure 3-4, a type 1 file is read directly to the user buffer. Other file types may be forced to type 1 when opened, in order to benefit from this type of transfer.

USING LEN---Upon completion of a read, the actual number of words transferred to the user buffer is returned in LEN. If, however, the number of words in LEN is equal to IL, more words may actually have been in the disc record. This is because LEN is never set to a value greater than IL.

LEN can be used to test for possible overflow of the user buffer. Except for type 1 files, the user buffer and IL can be specified one word larger than the largest expected record. If, when tested, LEN equals this size, it is a good indication that the record read was too large for the buffer. Do not use this test for type 1 files since exactly IL words are read for this file type.

Another use of LEN is to test for end-of-file in all file types except 1 and 2. For types 1 and 2, an end-of-file is reported as an error in IERR. Depending on file type, reading an end-of-file results in the following:

* Type 0 — LEN is set to -1 when EOF is read. No error occurs and access may continue beyond the end-of-file.

* Type 1 — LEN is set to the number of words which were transferred to the user buffer before the EOF was encountered. IERR is set to -12 indicating an error. Access is not permitted beyond the end-of-file unless the file was opened allowing extension.

* Type 2 — IERR is set to -12 indicating an error. Access is not permitted beyond the end-of-file unless the file was opened allowing extents.

* Type 3 & up — LEN is set to -1 for the first EOF read. No error occurs but an attempt to read past this EOF causes an error (IERR = -12). You may not read past the end-of-file, but you may write beyond it.

Note that length words in variable-length records (file types 3 and above) are not transferred to the user buffer and are not counted in LEN.

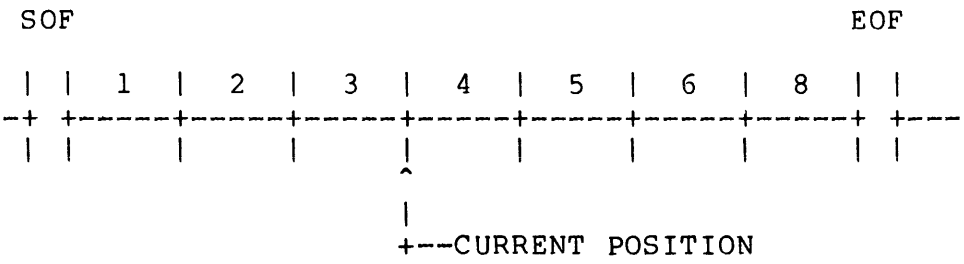
POSITIONING WITH NUM---NUM is used only to position file types 1 and 2; it may be specified for other file types but is ignored. If positive, NUM specifies the record number of the record to be read; records are numbered from the first record in the file starting with 1 and proceeding sequentially upward. If negative, NUM specifies the number of records to backspace from the current position in the file. If the number of records to backspace goes beyond the start-of-file, IERR is set to -12 indicating an error and the file is not read (type 1 and 2 files only).

To illustrate, assume the file is positioned at the beginning of record 4:

1. If NUM=0 or is omitted, read record 4.

2. If NUM=6, read record 6.

3. If NUM=-3, read record 1.



EXAMPLE 1: Read records in a type 0 file until an end-of-file is reached. The record length is 80 words but 81 words are assigned to the buffer in order to test LEN; assume the file is positioned at the first record.

```
          DIMENSION IDCB0(656),IBUF(81)
          .
          .
          .
   100    CALL READF(IDCB0,IERR,IBUF,81,LEN)
          IF(IERR .LT. 0) GO TO 900
          IF(LEN .EQ. -1) GO TO 500          !TEST FOR END-OF-FILE

          IF(LEN .GT. 80) GO TO 550        !TEST FOR RECORD GREATER
           .                                        !THAN 80 WORDS
           .                                      !PROCESS RECORD
           .
          GO TO 100                                !READ NEXT RECORD
          .
          .
          .
   500    CALL CLOSE(IDCB0,IERR)          !CLOSE FILE AT END-OF-FILE
          IF(IERR .LT. 0) GO TO 910
          .
          .
```

EXAMPLE 2: Read record 24 from a type 2 file having 256 word records; use 257 word buffer.

```
      DIMENSION IDCB1(272),IBUF1(257)
      .
      .
      .
      CALL READF(IDCB1,IERR,IBUF1,257,LEN,24)
      IF(IERR .LT. 0) GO TO 900              !TEST FOR ERROR OR EOF
      IF(LEN .EQ. 257) GO TO 550          !TEST FOR TOO LONG RECORD
      .
      .                                            !PROCESS RECORD
      .
```

EXAMPLE 3: Read file with variable-length records until first end-of-file is reached. Assume the file is positioned at the first record and that no record exceeds 128 words.

```
       DIMENSION IDCB2(144),IBUF2(129)
       .
       .
       .
100    CALL READF(IDCB2,IERR,IBUF2,129,LEN)
       IF(IERR .LT. 0) GO TO 900
       IF(LEN .EQ. -1) GO TO 500              !TEST FOR END-OF-FILE
       IF(LEN .EQ. 129) GO TO 150          !TEST FOR BUFFER OVERFLOW
       .
       .                                            !PROCESS RECORD
       .
       GO TO 100                                   !READ NEXT RECORD
       .
       .
       .
       .
       .
500    CALL CLOSE(IDCB2,IERR)             !CLOSE FILE AT END-OF-FILE
       .
       .
       .
```

WRITF and EWRIT Calls

These routines transfer a record from the user's buffer to an open file. For type 0 files or type 3 (and above) files, the specified number of words are written. For type 1 files, records are transferred in blocks of 128 words. For type 2 files, records are transferred in the lengths specified when the file was created.

For type 1 and 2 files, a specific or relative record number can be specified in NUM. For the WRITF call, the record number specified must be less than, or equal to, (2**15)-1. For the EWRIT call, the record number can be up to (2**31)-1.

```
CALL WRITF(IDCB,IERR,IBUF[,IL][,NUM])
   or           ----
CALL EWRIT(IDCB,IERR,IBUF,[IL],[NUM])
                ----
```

IDCB---DCB. An array of 144+n words where n is positive or zero; previously specified in a create or open call.

IERR---Error return. A one-word variable in which a negative error code is returned if the call was unsuccessful. Zero is returned for successful calls.

IBUF---Data buffer. An array where the user places the data to be written. It should be large enough to contain the largest expected record.

IL-----Buffer length. An optional one-word variable used to specify the number of words to be written. If omitted, one record is written to type 1 and 2 files; zero-length record is written for the other file types. Refer to Table 3-6 for details.

NUM----Record number. An optional one-word variable (for WRITF) or double-word variable (for EWRIT) used to specify the record to be written to (if positive), or the number of records to backspace (if negative). Used only for type 1 and 2 files. If omitted, record at current file position is written to.

COMMENTS:

RELATION OF IL TO FILE TYPE---IL should be specified for all but type 2 files and may be specified for all files. It is ignored by type 2 files but can be used with type 1 files to write more than one 128-word record at a time. For files of type 3 and above, it is essential to specify record length in IL. To omit IL for these file types is the same as setting IL=0, a zero-length record is written. Refer to Table 3-6. Type 2 files should be opened in update mode when writing to the file (refer to the OPEN FMP call).

IL can also be used to write an end-of-file on files of type 0, type 3 or greater. An attempt to write an end-of-file to a type 1 or 2 file is ignored; no error is indicated.

Table 3-6. Effect of IL Parameter in WRITF

IL VALUE	FILE TYPE 0	FILE TYPE 1	FILE TYPE 2	FILE TYPE > 2
IL > 0	Exactly IL words are written.	IL is rounded up to 128 or a multiple of 128.	IL is ignored; file defined record length is written.	Exactly IL words are written.
IL = 0	Zero length record is written.	No action.	IL is ignored; file defined record length is written.	Zero length record is written.
IL omitted	Zero length record is written.	128 words are written.	IL is ignored; file defined record length is written.	Zero length record is written.
IL=-1	End-of-file is written.	No action.	No action.	End-of-file is written.
IL <-1 not recommended	IL is treated as a character count.	No action.	No action.	Undefined.

When IL is not 128 or a multiple of 128 for a type 1 file, it is rounded up so that 128 or a multiple of 128 words are always transferred. The user buffer need be no larger than the size specified in IL. If the exact size is always read, no problems result from the transfer of words beyond the buffer. Figure 3-5 illustrates a write to a type 1 file with IL = 150 words. In this case, 256 words (the shaded area) are actually transferred.

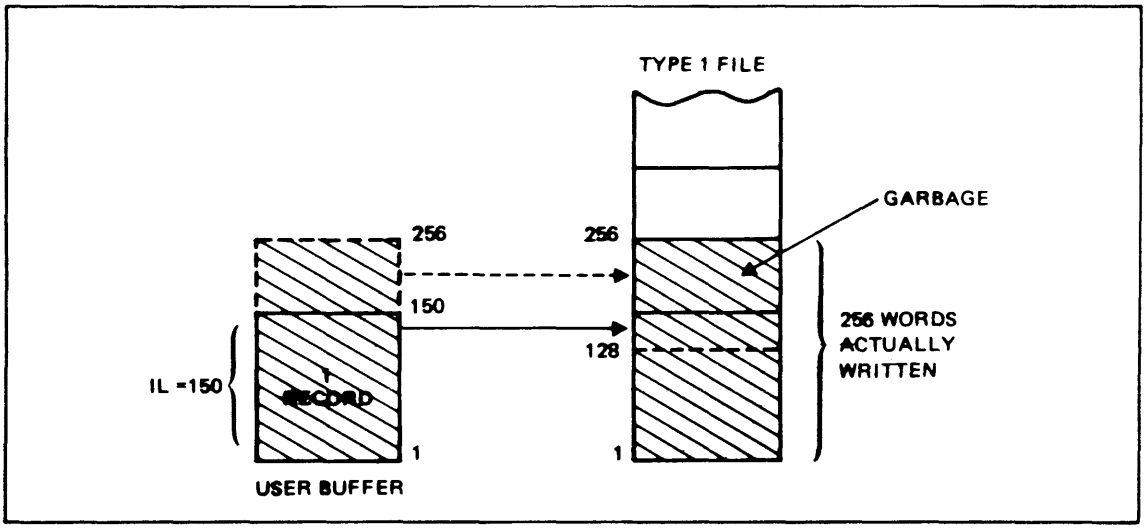


Figure 3-5. Sample WRITE to Type 1 File

POSITIONING WITH NUM---NUM is used only to position file types 1 and 2; if specified for other file types, it is ignored. A positive value causes a write to the specified record number; records are numbered relative to the start of the file beginning with 1. When negative, NUM specifies the number of records to backspace from the current file position. If the number of records to backspace goes beyond the start-of-file, IERR is set to -12 indicating an error and the data is not written to the file (type 1 and 2 files only).

To illustrate, assume the file is positioned at the beginning of record 5:

1. If NUM=0 or is omitted, write is to record 5.

2. If NUM=6, record number 6 is written.

3. If NUM=-3, record number 2 is written.

```
START OF FILE                              END OF FILE
    SOF                                        EOF

    |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |
 ---+ +---+---+---+---+---+---+---+---+---+ +---
    |  |   |   |   |   |   |   |   |   |  |
                        |
                        |
                        +---CURRENT POSITION
```

NOTE

Although it is possible to rewrite specific records in files of type 3 and above, great care must be taken. If the length of the existing record and that of the replacing record are not identical, an FMGR-005 error is issued.

EXAMPLE 1: Write records sequentially to a file starting at record 1; when all records are written, write an end-of-file; set IL to the exact record length of each record using a maximum record length of 100. File could be type 0 or a type 3 or above:

```
      DIMENSION IDCB0(144),IBUF0(100)
      .
      .
      .
100                                      !MOVE RECORD TO IBUF0
      .
      .
      .                                  !NUMBER OF WORDS IN RECORD
      IL =                    !(AFTER LAST RECORD, SET IL = -1)
      .
      .
      CALL WRITF(IDCB0,IERR,IBUF0,IL)
      IF(IERR .LT. 0) GO TO 900                !CHECK FOR ERROR
      IF(IL) 110,100
110   CALL CLOSE(IDCB0,IERR)                   !CLOSE FILE AFTER EOF
      .
      .
      .
```

EXAMPLE 2: Write record number 24 to a type 2 file with a record length of 256 words:

```
      DIMENSION IDCB2(272),IBUF2(256)
      .
      .
      .                                        !MOVE RECORD TO IBUF2
      CALL WRITF(IDCB2,IERR,IBUF2,0,24)
      IF(IERR .LT. 0) GO TO 900
      .
      .
      .
```

File Positioning

The FMP positioning routines provide a variety of ways to position a file. The methods are summarized below:

* APOSN and EAPOS -- Can be used to position all file types except type 0. Used to position a file to specific record; if file contains variable-length records, LOCF or ELOCF must be used in conjunction with APOSN or EAPOS, respectively.

* POSNT and EPOSN -- Can be used to position all file types. Used to position a file forward or backward (if backspace legal) from the current file position. Can also be used to position a file to a specific record.

* RWNDF -- Can be used to position a file at the first record for any file that permits backspacing.

Disc files with fixed-length records (type 1 and 2) can also be positioned to a particular record or backspaced with the NUM parameter in the READF, EREAD, WRITF, or EWRIT calls.

All disc files, minicartridge and magnetic tape files can be backspaced. Non-disc files, other than minicartridge or magnetic tape, usually have fixed-length records but cannot be backspaced.

In addition to being used in conjunction with the APOSN and EAPOS routines for file positioning, LOCF and ELOCF can also be used to return status information (size, position, etc.) on any open file.

Note that when a file is opened, it is automatically positioned at the first record.

LOCF and ELOCF Calls

A call to these routines retrieves status and location information on an open file. The information is obtained from the DCB associated with the file. The minimum information returned is the next record number; the other information is optional.

LOCF can only be used with files that are up to 16383 blocks in size and contains up to (2**15)-1 records. ELOCF can be used with files up to 32767 x 128 blocks in size that contain up to (2**31)-1 records.

When calling these routines on a file that has extents, the file size is returned as the number of sectors in the main only.

```
CALL LOCF(IDCB,IERR,IREC[,IRB][,IOFF][,JSEC][,JLU][,JTY][,JREC])
       or     ---- ----  ---  ----  ----  ---  ---  ----
CALL ELOCF(IDCB,IERR,IREC[,IRB][,IOFF][,JSEC][,JLU][,JTY][,JREC])
           ---- ----  ---  ----  ----  ---  ---  ----
```

IDCB--DCB. An array of 144+n words, where n is positive or zero; previously specified in a create or open call.

IERR--Error return. A one-word variable that a negative error code is returned to if the call is unsuccessful. Zero is returned for successful calls.

IREC--Next record. A one-word variable (for LOCF) or a double-word variable (for ELOCF) that the number of the next record to be accessed is returned to.

IRB---Next block. An optional one-word variable (for LOCF) or double-word variable (for ELOCF) that the relative block number containing the next record to be accessed is returned to.

IOFF--Next word. An optional one-word variable that the offset of the next record in the block is returned to; not returned for type 0 files.

JSEC--File size. An optional one-word variable (for LOCF) or double-word variable (for ELOCF) that the file size in sectors is returned to; not returned for type 0 files. JSEC/2 equals number of blocks in file.

JLU---LU. An optional one-word variable that the LU to which the file is allocated is returned to.

JTY---File type. An optional one-word variable that the file type defined when the file was opened is returned to.

JREC--Record length. Optional one-word variable that the record length (for type 1 or 2 files) or the read/write code (for type 0 files) is returned to. Meaningless for type 3 (and above) files.

COMMENTS:

LOCATION INFORMATION---Together, IREC, IRB, and IOFF provide the current position within a disc file; they are not set for non-disc files. The values in these parameters may be passed directly to APOSN (or EAPOS) to position the file to this location. The values returned in IRB and IOFF give the exact physical location of the record pointer in the file. The values of IRB and IOFF are based on a DCB buffer size of 128 words. If the actual DCB buffer size is greater than 128, these values are adjusted automatically by APOSN or EAPOS. Note that ELOCF and EAPOS must be used together because for each, the record number and relative block position are double-word variables.

IREC numbers records starting with one for the first record, two for the second, and so forth. IREC alone is sufficient to find the location in type 1 files. IRB numbers blocks starting with zero for the first block in the file, one for the second, and so forth. IRB includes extent information and is specified as:

(blks in main file) x (extent #) + (blk # in current extent)

IOFF numbers the words in a block beginning with zero. Since the DCB packing buffer is assumed to be 128 words the range of IOFF is 0 through 127.

STATUS INFORMATION---JSEC is always an even number, with two 64-word sectors for each 128-word block in a disc file. It is not applicable to non-disc files.

JLU is the LU to which a file, disc or non-disc, is allocated.

JTY is the file type of the file; if forced to type 1 at open, then one is returned.

JREC as a record length is meaningful for type 2 files only; it is the length specified at creation. For type 1 files, whether actual or forced to type 1 at open, the record length is set to 128 on the first read or write access.

For type 0 files, JREC specifies the read/write access code as follows:

* bit 15 = 1 read legal

* bit 0 = 1 write legal

EXAMPLE 1: Determine the actual location of the record pointer in the open file PROG1 defined in IDCB2:

```
      DIMENSION IDCB2(144)
      .
      .
      .
      CALL LOCF(IDCB2,IERR,IREC,IRB,IOFF)
      IF(IERR .LT. 0) GO TO 900                    !PROCESS ERRORS
      .
      .
      .
```

EXAMPLE 2: Open an existing file (DATA) and create a new file (NEW) with the same file type and size; transfer all data from DATA to NEW.

```
      DIMENSION IDATA(272),INEW(272),NDATA(3),NNEW(3)
      DIMENSION IBUF(256),ISIZ(2)
      DATA NDATA/6HDATA  /,NNEW/6HNEW   /
      CALL OPEN(IDATA,IERR,NDATA,0,0,0,272)
      IF(IERR .LE. 0) GO TO 900       !TEST FOR ERROR OR TYPE 0
   10 CALL LOCF(IDATA,IERR,I,I,I,ISIZ(1),I,ITYP,ISIZ(2))
      IF(IERR .LT. 0) GO TO 920
   20 ISIZ(1)=ISIZ(1)/2           !SET ISIZ TO NUMBER OF BLOCKS
      CALL CREAT(INEW,IERR,NNEW,ISIZ,ITYP,0,0,256)
      IF(IERR .LT. 0) GO TO 920
   30 CALL READF(IDATA,IERR,IBUF,256,L)
      IF(IERR .LT. 0) GO TO 920
   40 CALL WRITF(INEW,IERR,IBUF,L)
      IF(IERR .LT. 0) GO TO 920
      IF(L)950,30    !RETURN FOR NEXT RECORD IF NOT END-OF-FILE
```

PROCESS ERRORS AND CLOSE FILES

```
  900 IF(IERR .NE. 0) GO TO 920
      WRITE (1,'("IDATA IS TYPE 0 FILE")')      !PROGRAM CANNOT
      GO TO 950                                !CREATE TYPE 0 FILE
  920 WRITE (1'("ERROR ATTEMPTING TO COPY IDATA TO NEW")')
  950 CALL CLOSE(IDATA,IERR)
 1000 CALL CLOSE(INEW,IERR)  !CLOSE FILES FOLLOWING END-OF-FILE
      END                                                OR ERROR
```

APOSN and EAPOS Calls

These routines are called to position any disc file to a specific record. For the APOSN routine, the record location can be determined by a prior call to LOCF. ELOCF would be used with the EAPOS call.

APOSN can be used to position files up to (2**15)-1 blocks in length and containing up to (2**15)-1 records. EAPOS can position files up to 32767 x 128 blocks in size and contain up to (2**31)-1 records.

APOSN and EAPOS are intended to position sequential files with variable-length records prior to a read or write request. They may also be used to position random files with fixed-length records (type 1 and 2) but cannot be used to position type 0 files (non-disc files). The POSNT and EPOSN routines described later can be used to position type 0 files.

```
CALL APOSN(IDCB,IERR,IREC[,IRB[,IOFF]])
      or       ----
CALL EAPOS(IDCB,IERR,IREC[,IRB[,IOFF]])
                 ----
```

IDCB--DCB. An array of 144+n words, where n is positive or zero; previously specified in an open or create call.

IERR--Error return. A one-word variable in which a negative error code is returned if the call is unsuccessful; zero returned for successful calls.

IREC--Record number. A one-word variable (for APOSN) a double-word variable (for EAPOS) that specifies the record number where the file is to positioned; can be determined by a prior call to LOCF or ELOCF.

IRB---Next block. An optional one-word variable (for APOSN) or double-word variable (for EAPOS) that specifies the block number containing the next record; can be determined by a prior call to LOCF or ELOCF. Required for files with variable-length records; omitted for files with fixed-length records.

IOFF--Next word. An optional one-word variable that specifies the offset in the block of the next record; can be determined by a prior call to LOCF or ELOCF. Required for files with variable-length records; omitted for files with fixed-length record.

COMMENTS:

PARAMETER CONSIDERATIONS---APOSN and EAPOS assume the value entered for the record address (IREC) is based on a 128-word DCB packing buffer. The routines adjust the values according to the buffer size currently in use if it is greater than 128-words.

IREC must be specified; the value can be determined by the user or obtained from a prior call to LOCF or ELOCF. In addition, IRB and IOFF can be retrieved by LOCF or ELOCF, permitting a file to be reset to a previous record position.

Note that when a file with variable-length records is positioned, the two parameters, IRB and IOFF, must be included.

EXAMPLE 1: Call LOCF to retrieve and save the current position parameters. After reading more of the file, re-position the file associated with IDCB2 to the position whose location was saved.

```
      DIMENSION IDCB2(144)
          .
          .
          .
      CALL LOCF(IDCB2,IERR,IREC,IRB,IOFF)     !RETRIEVE LOCATION AT
      IF(IERR .LT. 0) GO TO 900                     !START OF ACCESS
          .
          .                        !READ AND PROCESS RECORDS IN THE FILE
          .
      CALL APOSN(IDCB2,IERR,IREC,IRB,IOFF)  !POSITION TO PREVIOUSLY
      IF(IERR .LT. 0) GO TO 900                    !SAVED LOCATION
          .
          .
          .
```

The values of IREC and IOFF retrieved by the call to LOCF are simply passed to the call APOSN by using the same variable names in both calls.

EXAMPLE 2: Position type 2 file defined in DCB IDCB1 to the sixth record in the file and then read the next eight records in sequence.

```
          DIMENSION IDCB1(144),IBUF(128)
          INTEGER COUNT
          .
          .
          CALL APOSN(IDCB1,IERR,6)                      !POSITION FILE
          IF(IERR .LT. 0) GO TO 900
          COUNT=0
    10    CALL READF(IDCB1,IERR,IBUF1,128,LEN)
          IF(IERR .LT. 0) GO TO 900
    20    IF(LEN .EQ. -1) GO TO 50            !TEST FOR END-OF-FILE
          .
          .                                          !PROCESS RECORD
          .
          COUNT=COUNT+1
          IF(COUNT .LT. 8) GO TO 10
          .
          .
          .
```

POSNT and EPOSN Calls

These routines position files relative to the current file position or to a specific record number. They can be used to position all file types.

POSNT can be used to position files containing up to (2**15)-1 records. EPOSN can be used to position files containing up to (2**31)-1 records.

```
CALL POSNT(IDCB,IERR,NUR[,IR])
    or          ----
CALL EPOSN(IDCB,IERR,NUR[,IR])
                ----
```

IDCB--DCB. An array of 144+n words, where n is positive or zero; previously specified in a create or open call.

IERR--Error return. A one-word variable in which a negative error code is returned if the call is unsuccessful; zero is returned for successful calls.

NUR---Record position. A one-word variable (for POSNT) or double-word variable (for EPOSN) that specifies:

* number of records to position forward if positive (IR=0).
* number of records to backspace if negative (IR=0).
* absolute record number to position to (IR does not equal 0).

IR----Position mode flag. An optional one-word variable that performs the following function:

* if non-zero, indicates that NUR is to be interpreted as specific record number.
* if zero, indicates that NUR is to be used for relative positioning (forward or backspace).

Refer to Table 3-7 for details.

COMMENTS:

The relationship between NUR and IR are shown in Table 3-7 below:

Table 3-7. Relationship Between NUR and IR

NUR	IR = 0 OR OMITTED RELATIVE POSITION	IR ≠ 0 ABSOLUTE POSITION
NUR > 0	Position forward number of records specified.	Position to record number specified.
NUR = 0	No operation.	No operation
NUR < 0	Position backward number of records specified.	Error

POSITIONING TYPE 0 FILES---When the file is on a non-disc device, the forward or backward positioning specified by NUR must be legal for the device.

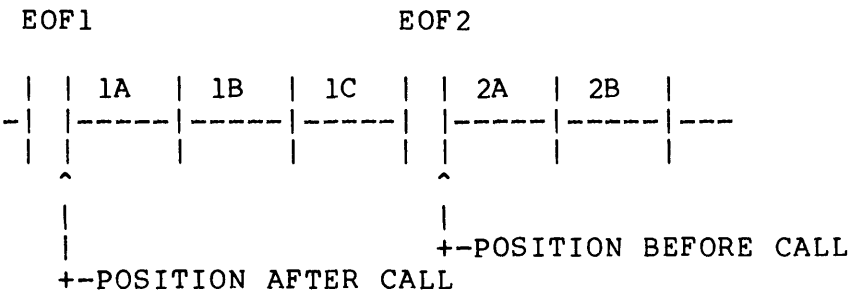
To forward position a type 0 file, the records are read until one less than the specified number of records is read or an EOF is encountered. In every case, an EOF terminates positioning.

When backspacing a type 0 file, the first record backspaced over may be an EOF. If an EOF is encountered not as the first record backspaced over, an FMP error (-012) is returned and the call terminates after forward spacing to position the file immediately after the EOF. If the number of records to backspace goes beyond the start-of-file, an FMP error (-003) is returned.

POSITIONING TYPE 1 AND 2 FILES---EPOSN or POSNT may be used to position these file types, however, file positioning for files with fixed-length records can be specified in the read or write requests and POSNT or EPOSN may not be necessary. If the number of records to backspace goes beyond the start-of-file, an FMP error (-012) is returned. These files may be positioned beyond the end-of-file, if the file was opened allowing extents.

POSITIONING TYPE 3 AND ABOVE FILES---These files are treated as magnetic tape files. To be correct, a backspace should be issued after an EOF is read and before continuing to write on the file. This action writes the next record over the EOF allowing a correct extension of the file for either disc or magnetic tape files. If the file is on disc, it can be extended without backspacing. If the number of records to backspace goes beyond the start-of-file, an FMP error (-012) is returned.

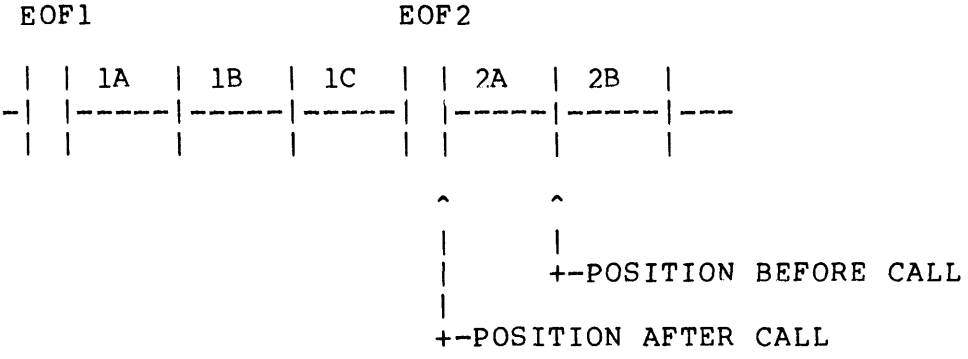
EXAMPLE 1: Current file position immediately follows EOF2. Backspace four records to final position immediately following EOF1 (no error). Note that EOF2 is counted as a record:




```
DIMENSION IDCB3(144)
.
.
.
NUR=-4
CALL POSNT(ICB3,IERR,NUR)
IF(IERR .LT. 0) GO TO 900
.
.
.
```

EXAMPLE 2: Current file position follows record 2A. Backspace five records resulting in a FMP-012 error because the end-of-file was not the first record backspaced over. The call is terminated and the file is left positioned immediately after EOF2.

```
      DIMENSION IDCB3(144)
      .
      .
      .
      NUR=-5
      CALL POSNT(IDCB3,IERR,NUR)
      IF(IERR .LT. 0) GO TO 900
      .
      .
      .
  900 WRITE(1,'("POSITION ERROR=",I5)')IERR
```



EXAMPLE 3: Position file with DCB JDCB to record number 10:

```
      DIMENSION JDCB(144)
      .
      .
      .
      NUR=10
      IR=1
      CALL POSNT(JDCB,IERR,NUR,IR)
      IF(IERR .LT. 0) GO TO 900
      .
      .
      .
```

RWNDF Call

A call to this routine rewinds a magnetic tape or positions a disc file to the first record in the file.

```
CALL RWNDF(IDCB[,IERR])
           ----
```

IDCB--DCB. An array of 144+n words, where n is positive or zero; previously specified in a create or open call.

IERR--Error return. An optional one-word variable that a negative error code is returned to if the call is unsuccessful. IERR can be omitted when the A-Register is checked for error conditions.

COMMENTS:

If the rewind cannot take place, the call is completed with the file position unchanged; no error is indicated. The rewind will not take place if, for instance, the file being rewound is a paper tape punch, the line printer, or some other device that cannot be backspaced.

EXAMPLE 1: Position a disc file to the first record:

```
DIMENSION IDCB2(144)
.
.
.
CALL RWNDF(IDCB2,IERR)
IF(IERR .LT. 0) GO TO 900
.
.
.
```

Special Purpose FMP Calls

The Special Purpose FMP calls provide functions not directly related to the standard I/O functions of defining, accessing, and positioning files. The special FMP routines are summarized below:

* FCONT -- Performs control operations on type 0 files. The control functions are identical to those provided by the I/O control EXEC call, except that the device is identified by a file DCB, instead of an LU.

* FSTAT -- If non-session environment, returns the status of all cartridges in the FMP cartridge directory. If under session control, returns status of cartridges in user's cartridge addressing space.

* IDCBS -- Retrieves the actual size of the DCB buffer for a currently open file. The usable buffer size allocated by FMP is determined by the total file size and the requested buffer size.

* NAMF -- Renames a created file. The file itself is not changed, but can no longer be opened or purged using the old name.

* POST -- Writes the contents of the DCB buffer to the disc. FMP normally performs this function when the file is closed or the buffer is full. POST is useful mainly to enable modification of records in a file opened for non-exclusive use.

FCONT Call

This routine performs control functions on a peripheral device. The device should have a type 0 file associated with the device by the FMGR CR command and opened as a type 0 file. The call has no effect on other file types. The FCONT call performs the same functions as the I/O Control EXEC call, i.e., rewind, write end-of-file to magnetic tape, etc.

```
CALL FCONT(IDCB,IERR,ICON1[,ICON2])
                ----        =====
```

IDCB---DCB. An array of 144+n words, where n is positive or zero; previously linked to a type 0 file by an OPENF FMP call.

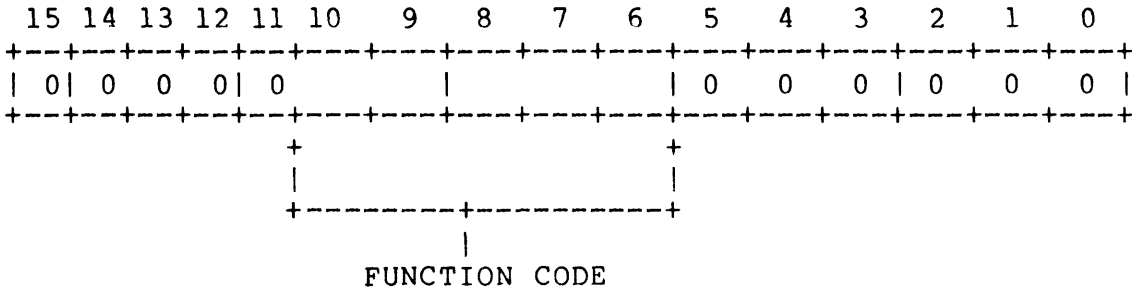
IERR---Error return. A one-word variable that a negative error code is returned to if the call is unsuccessful. Zero is returned for successful calls.

ICON1--Function code. A one-word variable set to an octal code defining the function to be performed. See Table 3-8 for specific codes.

ICON2--Used in conjunction with certain function codes (ICON1). If ICON1=11B (line spacing), ICON2 specifies the number of lines to be spaced. If ICON1=27B (find file) ICON2 specifies a file number. If ICON1=6 (device status), the status is returned in ICON2.

COMMENTS:

FUNCTION CODE---Bits 6 through 10 of ICON1 are used for the function code.



The function codes are defined in Table 3-8.

ICON1 AND ICON2---If the function code is 11 octal, then FCONT expects a value in ICON2. This value controls output line spacing on the line printer or a keyboard display device:

- 0 to suppress line spacing on the next line.
- >0 to indicate the number of lines to space before the next line.
- <0 to page eject on line printer; space specified lines on keyboard device.

If the function code is 27 octal, then FCONT expects a value in ICON2. This value declares the absolute file number to be located, in the range 1 through 255.

Table 3-8. FCONT Function Codes

FUNCTION CODE (OCTAL)	FUNCTION	DEVICE
00	Unused	
01	Write end-of-file	Magnetic tape Cartridge Tape Unit
02	Backspace one record	Magnetic tape Cartridge Tape Unit
03	Forward space one record	Magnetic tape Cartridge Tape Unit
04	Rewind	Magnetic tape Cartridge Tape Unit
05	Rewind standby Rewind	Magnetic tape Cartridge Tape Unit
06	Actual device status	Magnetic tape Cartridge Tape Unit
07	Set end-of-tape	Paper tape/TTY
10	Generate leader	Paper tape/TTY
	Write end-of-file if not just previously written or not at load point	Cartridge Tape Unit

Table 3-8. FCONT Function Codes (Continued)

FUNCTON CODE (OCTAL)	FUNCTION	DEVICE
*11	List output line spacing	Line printer
12	Write 3-inch interrecord gap	Magnetic tape
13	Forward space on file	**Magnetic tape Cartridge Tape Unit**
14	Backspace one file	**Magnetic tape Cartridge Tape Unit**
15	Conditional top-of-form	Line printer/display device
20	Enable terminal—allows terminal to schedule its program when any key is struck	Codes 20-27 are defined for a keyboard terminal (DVR00 and DVA05/DVR05). Refer to the DVR00 manual 29029-60001 or the DVR05/DVA05 manual 92001-90015.
21	Disable terminal—inhibits scheduling of terminal program	
22	Set time-out—sets new time-out interval	
23	Ignore all further requests until: • the request queue is empty or • an input request is received or • a restore control request is received	
24	Restore output processing (this request is usually not needed)	
26	Write end-of-data	**Cartridge Tape Unit**
**27	Locate file number	**Cartridge Tape Unit**

*When function code 11 is specified in ICON1, then ICON2 must be included in the parameter list to specify the particular line spacing.

**When function code 27 is specified in ICON 1, then ICON2 must be included in the parameter list to specify the particular file number.

In general, the operating system ignores control requests that are not defined for the specified device; however, some drivers will return error I007.

EXAMPLE 1: Backspace one record on file (MT) previously created and opened with DCB MTDCB.

```
      DIMENSION MTDCB(144)
      .
      .
      .
      ICONT=200B                                    !FUNCTION CODE 2
      CALL FCONT(MTDCB,IERR,ICONT)
      IF(IERR .LT. 0) GO TO 900
      .
      .
      .
```

EXAMPLE 2: Page eject to line printer created and opened as a file with DCB LPDCB.

```
      DIMENSION LPDCB(144)
      .
      .
      .
      ICONT=1100B                 !PAGE EJECT ON LINE PRINTER (WOULD
      LPSP=-1                      !SPACE 1 LINE ON KEYBOARD DEVICE)
      CALL FCONT(LPDCB,IERR,ICONT,LPSP)
      IF(IERR .LT. 0) GO TO 900
      .
      .
```

FSTAT Call

This routine returns the status of mounted cartridges in the cartridge directory. When under session control, ISTAT will contain information about only those discs mounted to the session's SCB and system discs. The caller may get information on all discs mounted to the system by specifying IOP non-zero. FSTAT gives the option of providing the information in one of two formats. The first format provides for each cartridge the LU, the last FMP track, the CRN, if a program has locked the cartridge, and its ID segment address. The second format provides the above information plus the identifier of who mounted the cartridge.

Note that in the second format, a locking program is represented by its ID segment number instead of its ID segment address.

```
CALL FSTAT(ISTAT,[ILEN],[IFORM],[IOP],[IADD])
           -----
```

ISTAT--Status buffer. An array that the status of the cartridges is returned to (see Table 3-9 and Table 3-10).

ILEN---Buffer length. An optional one-word variable that specifies the length in words of ISTAT. If omitted, ISTAT is assumed to be 125 words.

IFORM--Format. An optional one-word variable that specifies whether Format I or Format II will be used to return the cartridge directory (see Table 3-9 and Table 3-10). If zero, Format I is used; if non-zero, Format II is used.

IOP-----Option. An optional one-word variable that specifies the type of cartridges that information is to be returned about. If:

IOP=1 The status on all cartridges mounted to the system is returned in ISTAT; whether under session control or not.

IOP=0 If under session control, the status of private and group cartridges mounted to the session and system cartridges are returned (in that order).

IOP=0 If not under session control, status on system cartridges and non-session cartridges are returned (in that order).

IADD---System sets IADD to non-zero if not all the cartridge list could be returned in ISTAT (e.g., ISTAT not large enough). IADD is set to 0 if all cartridges were returned.

COMMENTS:

The two formats that can be used with the FSTAT call are shown in Table 3-9 and Table 3-10, below:

Table 3-9. ISTAT Format I

ISTAT		
WORD	CONTENTS	CARTRIDGE
1 2 3 4	Cartridge LU Last FMP track CRN Lock Word	First cartridge
5 6 7 8	Cartridge LU Last FMP track CRN Lock Word	Second cartridge
9 . . .	Cartridge LU . . .	. . .
	0 no more discs	

where:

Lock word - is ID segment address of locking program or 0 (not locked).

Table 3-10. ISTAT Format II

ISTAT		
WORD	CONTENTS	CARTRIDGE
1 2 3 4	Lock word | Cartridge LU Last FMP track CRN ID	First cartridge
5 6 7 8	Lock word | Cartridge LU Last FMP track CRN ID	Second cartridge
9 . . .	Lock word | Cartridge LU . . .	. . .
	0 no more discs	

where:

Lock word - is the offset of the ID segment in the Keyword Table or 0 (not locked).

ID - is the user account ID number, identifying who mounted the cartridge.

EXAMPLE 1: Find the CRN of the cartridge associated with LU 13 in order to create file XX on that cartridge. Use Format II and only search cartridges mounted to the current session.

```
      DIMENSION idcb(144), istat(253), isize(2)
      DATA isize /100,128/
      CALL FSTAT (istat,253,1,0)        ! Get cartridge directory
      DO i = i,248,4                    ! Check LU 13 (bits 0-5)
         IF (iand(istat(i),77B) .EQ. 13) GOTO 100
      END DO
           :
      < error, LU not found >
           :
  100 icr = istat(i+2)                  ! Get the CRN
      CALL CREAT(idcb, ierr, 6HXX, isize, 3, 0, icr) ! Create file
           :
      < etc. >
```

IDCBS Call

This function returns the number of words in a DCB actually used by the File Management Package for data transfer and file control.

ISIZE=IDCBS(IDCB)
IDCB---DCB. An array of 144+n words, where n is positive or zero; previously specified in a create or open call.

COMMENTS:

When a DCB larger than 144 words is specified for the file at open or creation, the File Management Package may not use the entire DCB buffer area. The actual size used depends on the file size as well as the requested buffer size. This call returns the actual DCB size; the buffer used plus 16 control words. Refer to the description of the parameter at the beginning of this section for more details.

EXAMPLE 1: A file has been opened using the DCB MBUF with a size of 5000 words. Use IDCBS to determine how much of MBUF is being used by FMP. If 144 or more words remain, create KFIL using the remainder of MBUF as a DCB.

```
DIMENSION NAM(3),NAM1(3)
DIMENSION ISZ1(2),MBUF(5000)
DATA NAM/6HMYFILE/,NAM1/6HKFIL  /
ICR=-14
CALL OPEN (MBUF,IERR,NAM,0,ICR,5000-144)
IF(IERR.LT.0) GO TO 150
ISZ1(1)=5
ITYPE=3
ISIZE=IDCBS(MBUF)
CALL CREAT(MBUF(ISIZE),IERR,NAM1,ISZ1,ITYPE,0,ICR,5000-ISIZE)
IF(IERR.LT.0) GO TO 150
```

NAMF Call

This routine renames an existing file. If the code was created with a security code, this code must be specified. If the file is open, it is closed and then renamed.

```
CALL NAMF(IDCB,IERR,INAM,MNAM[,ISC][,ICR])
          ----
```

IDCB---DCB. An array of 144+n words, where n is positive or zero.

IERR---Error return. A one-word variable that a negative error code is returned to if the call was unsuccessful. Zero is returned for successful calls.

INAM---File name. A three-word array containing the ASCII file name of the original file.

MNAM---File name. A three-word array containing the ASCII file name to replace INAM.

ISC----Security code. An optional one-word variable. Omitted if INAM was created without a security code (or 0); must match if INAM has security code.

ICR----Cartridge identifier. An optional one-word variable that specifies the cartridge that INAM resides on. If omitted or zero, the first file found that matches the original file name (INAM) is renamed.

COMMENTS:

EXAMPLE 1: Rename file PROG1 as MYFILE. Since a cartridge identifier is specified, the system will look for PROG1 on LU 14 only. No security code is needed.

```
      DIMENSION IDCB2(144),NAME(3),NNAME(3)
      DATA NAME/6HPROG1 /,NNAME/6HMYFILE/
      ICR=-14
      CALL NAMF(IDCB2,IERR,NAME,NNAME,0,ICR)
      IF(IERR .LT. 0) GO TO 900
      :
```

POST Call

This routine is called to post (write) the contents of the DCB buffer to a disc file (type 2 or above). Normally, this is done by the system when the buffer is full or the file is closed. POST provides direct control over the physical write to disc, assures that the next read is from disc, and can be used in a special case to save records in a file opened for non-exclusive use.

```
CALL POST(IDCB,[IERR])
               ----
```

IDCB---DCB. An array of 144+n words, where n is positive or zero; previously specified in an open or create call.

IERR---Error return. An optional one-word variable that a negative error code is returned to if the call is unsuccessful. Zero is returned for successful calls.

COMMENTS:

This call is ignored for files of type 0 or 1 since transfers to these files are always direct (bypass the DCB buffer).

USING POST FOR MODIFICATION---In conjunction with the library subroutine RNRQ (refer to Chapter 5), POST allows several programs to modify a file without requiring an exclusive open. RNRQ is used to lock the file for exclusive use of the calling program, and POST then clears the DCB buffer before modifying the record and again after modification.

The sequence to be followed is:

1. Open the file.
2. Read the file to retrieve the resource number (RN).
3. Call POST to clear the DCB buffer (no data is posted since none was written). This insures that the next read accesses the disc.
4. Call RNRQ to lock the file for exclusive use of the calling program.

5. Call READF to read the record to be modified.

6. Modify the record and call WRITF to write the record.

7. Call POST to transfer the updated record to the file from the DBC buffer.

8. Call RNRQ to unlock the file for use by other programs.

It is possible that WRITF in step 6 above causes the buffer to be posted to the disc, but POST should be called to insure the transfer.

EXAMPLE 1: Assume the resource number is in location IRN; modify record number 5 in a type 2 file opened non-exclusively.

```
      DIMENSION IDCB(144),IBUF(128)
   30 CALL POST(IDCB,IERR)                    !CLEAR DCB BUFFER
      IF(IERR .LT. 0) GO TO 150
      ICODE=1B                             !SET CODE TO LOCAL LOCK
   40 CALL RNRQ(ICODE,IRN,ISTAT)        !LOCK FILE FOR EXCLUSIVE
      IF(ISTAT .NE. 2) GO TO 150                !USE OF PROGRAM
   50 IL=80
      CALL READF(IDCB,IERR,IBUF,IL,LEN,5)       !READ RECORD 5
      IF(IERR .LT. 0) GO TO 150
      IF(LEN .GT. 80) GO TO 150
      .
      .                                    !MODIFY RECORD IN IBUF
      .
   60 CALL WRITF(IDCB,IERR,IBUF,IL,5)   !WRITE MODIFIED RECORD
      IF(IERR .LT. 0) GO TO 150
   70 CALL POST(IDCB,IERR)                    !CLEAR BUFFER AGAIN
      IF(IERR .LT. 0) GO TO 150
   80 ICODE=4B                           !SET CODE TO UNLOCK FILE
      .
      .
      .
      CALL RNRQ(ICODE,IRN,ISTAT)
      .
      .
      .
  150 process errors here and unlock Resource Number
      .
      .
```

Examples Using FMP Calls

The following example demonstrates the use of FMP calls in a program. The program updates the specified element in a record of a type 2 file.

```
      PROGRAM FMPCS
C
C     This program updates an element in any record of the data
C     file CSDATA (file type=2, record length=32, security
C     code=CS).
C
      DIMENSION IDCB(144),IBUF(32),INAM(3),JBUF(32)
      DATA INAM/6HCSDATA/
C
C     Open the data file in update mode
C
      ISC = 2HCS
      CALL OPEN(IDCB,IERR,INAM,2,ISC)
      IF (IERR.LE.0) GO TO 1000
C
C     Locate the specified record
C
      WRITE(1,*)'WHAT RECORD WOULD YOU LIKE TO UPDATE')
      READ(1,*) IREC
      CALL APOSN(IDCB,IERR,IREC)
      IF (IERR.LT.0) GO TO 1000
C
C     Determine which element in the record should be updated
C
      WRITE(1,'(/"THE ELEMENTS OF RECORD ",I6," ARE :")') IREC
C
C     Read in the record and write it to the terminal
C
      CALL READF(IDCB,IERR,IBUF,32)
      IF (IERR.LT.0) GO TO 1000
      DO 30 J=1,32
        WRITE(1,'("ELEMENT # ",I6," = ",I6)') J,IBUF(J)
   30 CONTINUE
C
      WRITE(1,*)'WHICH ELEMENT # WOULD YOU LIKE TO UPDATE?')
      READ(1,*) INUM
C
C     Verify the element to be updated
C
      CALL APOSN(IDCB,IERR,IREC)
```

```
      IF (IERR.LT.0) GO TO 1000
      CALL READF(IDCB,IERR,IBUF,32)
      IF (IERR.LT.0) GO TO 1000
      WRITE(1,'("THE VALUE OF THE ELEMENT = ",I6)') IBUF(INUM)
      WRITE(1,*)'ENTER THE NEW VALUE OF THE ELEMENT')
      READ(1,*) NEWVAL
C
C     Update the file
C
      IBUF(INUM) = NEWVAL
      CALL POSNT(IDCB,IERR,IREC,1)
      IF (IERR.LT.0) GO TO 1000
      CALL WRITF(IDCB,IERR,IBUF,32)
      IF (IERR.LT.0) GO TO 1000
C
C     Close the file
C
      CALL CLOSE(IDCB,IERR)
      IF (IERR.LT.0) GO TO 1000
C
      STOP
C
C     Errors were found
C
 1000 WRITE(1,'("FMP ERROR OCCURRED, ERROR CODE =",I6)') IERR
C
      END
```

Chapter 4
VMA and EMA Programming

Introduction

The RTE-6/VM Operating System provides the user with the capability of managing large data arrays. The RTE-6/VM Demand-Paged Virtual Memory system allows the user to access very large data areas up to 128 Megabytes. The program data resides in a Virtual Memory Area (VMA) located on disc and pages of data are swapped into memory when needed. The user can also handle large arrays that reside entirely in main memory. These arrays are a subset of Virtual Memory where the entire array is memory-resident as opposed to part of the array being on disc. This subset is called the Extended Memory Area (EMA). EMA is memory that can extend beyond the logical address space of a program up to the available physical memory. This allows an EMA program to process large amounts of data in main memory rather than swapping in blocks of data from disc resulting in a great increase in program speed. EMA data can be shared by other programs. However, VMA data is not shareable.

These large arrays are managed via the VMA/EMA software and firmware. In addition to being a powerful operating system feature, VMA/EMA is fast and simple to use. When programming in a high level language, all the user must do is indicate which arrays are to be treated as VMA/EMA arrays, and the system handles the rest. VMA, EMA, and shareable EMA, are all declared the same in the source program; distinction is made when loading. A VMA/EMA variable cannot be used as a parameter in an EXEC or FMP call.

This chapter provides an overview of VMA/EMA, shareable EMA, partition considerations, and programming with VMA/EMA. Appendix B contains VMA/EMA mapping subroutines. Certain terms used in discussing VMA/EMA are defined in Table 4-1.

Table 4-1. VMA and EMA Terms

TERM	DESCRIPTION
Logical Memory	The 32 page address space described by the currently enabled memory map; can be any of the possible pages of physical memory.
Physical Memory	The actual semiconductor memory installed in the machine (up to 1024 pages of physical memory).
Virtual Memory (VMA)	An area on disc that can be used to extend main memory. This disc memory can contain very large data arrays which are accessible by the user's program. Data in disc memory (virtual data) can be accessed via a simple program statement (i.e., I=J (5000000)). The disc memory is made to look like logical memory to the user.
Extended Memory Area (EMA)	A subset of Virtual Memory. This is an area of physical memory that extends beyond the user's logical address space and can be used for large data arrays. The EMA program and data can be as large as physical memory less the operating system.
Virtual Memory Mapping Segment (VSEG)	The last two pages of the user's logical address space that are used by the VMA/EMA firmware to 'map in' the VMA/EMA data accessed by the user's program.
Mapping Segment (MSEG)	The guaranteed maximum VMA/EMA record size (in pages) that can be present in the user program's logical address space.
Page Table (PTE)	The first page following the program space in physical memory. It indicates to the firmware which pages of the VMA data are currently in physical memory.
Page Fault	Condition that occurs when the page table indicates that the requested data does not reside in physical memory.
Working Set	The portion of virtual data that is currently in physical memory (not including the page table).
Backing Store File	The file on disc used for storage of the virtual data.

Virtual Memory Area (VMA)

Previous RTE Operating Systems required all of a program's data to reside in physical memory at one time in order for a program to execute. However, with the Demand-paged Virtual Memory system, provided with RTE-6/VM, data for executing programs can exceed the size of available physical memory by allowing the program's data to reside in a Virtual Memory Area (VMA) located on disc. The maximum amount of VMA data a single program can have is 128M bytes. Many Virtual Memory programs can execute at the same time; each accessing their own VMA.

When a program is first dispatched, none of the VMA array or variables reside in physical memory. When a VMA data element is accessed, the data must reside in physical memory and be mapped into the user's logical address space by the VMA/EMA software and firmware routines. These routines require use of a VMA page table in order to resolve virtual addresses to physical addresses. The page table is the first page following the program space in physical memory, as shown in Figure 4-1. The page table is not included in the VMA page size declared by the user.

A virtual memory array or variable is declared in the program as an EMA array or variable. When loading the program, the EMA area becomes a VMA area by using the loader VM, VS, or WS commands. Refer to the section on Programming With VMA/EMA for further details. To the programmer, VMA looks like an extension of main memory.

In actuality, only a part of the VMA data resides in physical memory, in an area called the "working set" as shown in Figure 4-1. The remainder of the VMA data resides in a "backing store file" located on disc. The maximum size of the backing store file is 65,536 pages. The sizes of the working set and the virtual memory area on disc can be modified at load time by using LOADR or MLLDR commands. The default sizes of the working set and the virtual memory are 31 pages and 8192 pages, respectively. These values can also by modified on-line by the WS and VS system commands as described in the RTE-6/VM Terminal User's Reference Manual. Refer to the RTE-6/VM Loader Reference Manual for a detailed description of the loader commands.

There are two modes of VMA operation. The default mode of operation is used when a program creates a large amount of temporary data. In this mode, the backing store is uninitialized at the beginning of program execution and purged after the program terminates. During program execution, VMA data is placed in the working set. The data is posted to the backing store file on a page-by-page basis only when the amount of data created exceeds the working set size. In this way, disc space for the backing store file is allocated only as it is needed. The default mode of operation is used when a program creates a large amount of temporary data.

The user may choose to use the alternate mode of VMA operation for programs which create a large amount of data to be used later, or use a large amount of existing data. In this mode, the program calls a subroutine to create a named backing store file or to open an existing file to be used as the backing store file. The backing store file can be manipulated using VMA file subroutines discussed later in this chapter. In this mode when the program references a VMA data element not currently in the working set, the pages containing the VMA data are swapped from the backing store file into the program's working set. If the working set is full and the data does not currently reside in the working set, one page of the working set is flushed to the disc, and the requested page of data is swapped into that page.

To increase the performance of your VMA program, VMA arrays should be accessed in column order (for FORTRAN) or row order (for Pascal) due to their internal representation. There are fewer accesses to the disc when the array is accessed in column order or row order. The VMA/EMA firmware will only swap in data from the disc when necessary.

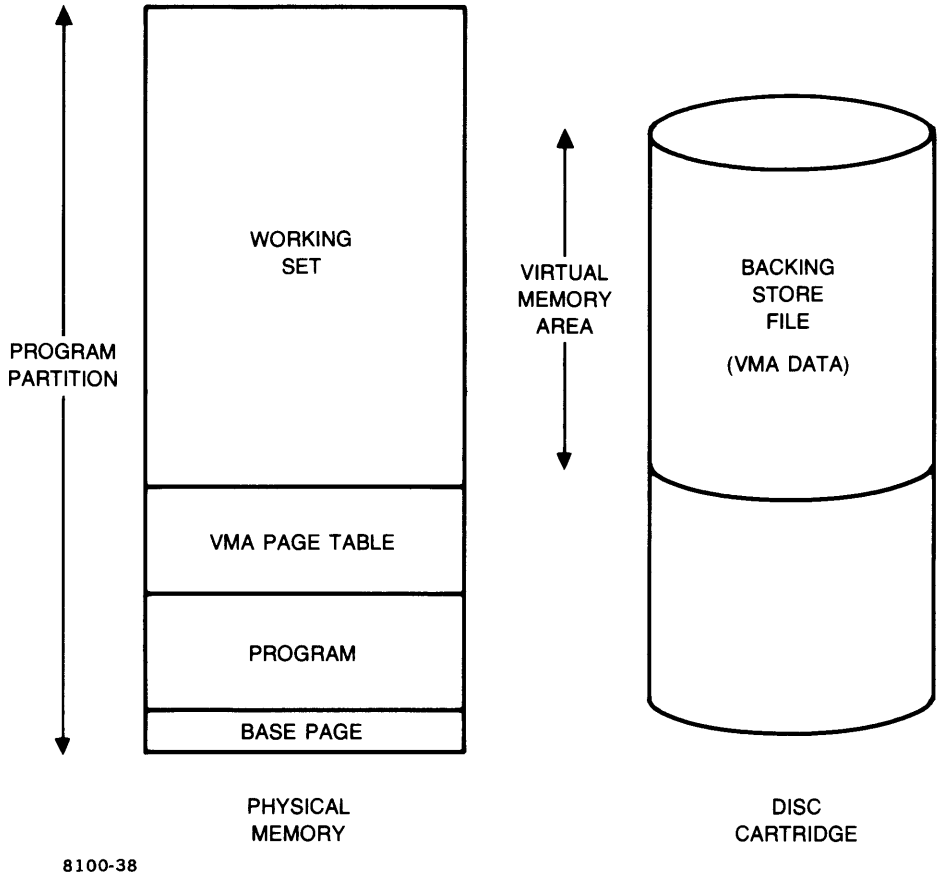


Figure 4-1. Working Set and Backing Store File

When using VMA, the data being accessed must also be included in the program's logical address space. The physical pages in memory containing the VMA data are mapped by the VMA/EMA mapping routines. Refer to Appendix B for a description of the mapping routines. In high level languages, calls to the VMA/EMA software and firmware routines are automatically emitted by the compiler without explicit user action. However, in a Macro/1000 program, explicit calls must be made by the user to the VMA/EMA subroutines in order to map in data.

Extended Memory Area (EMA)

The Extended Memory Area (EMA) is a subset of Virtual Memory. With EMA, all the program's data resides in physical memory within the program's partition. EMA data is mapped into logical memory via the VMA/EMA software and firmware routines described in Appendix B. However, since all the data resides in physical memory the array resolution is fast since the data will never be swapped to or from the disc. The virtual memory system requires one page in physical memory, the page table. For EMA, the page table indicates to the firmware that all of the data resides in memory. The structure of EMA and Memory is shown in Figure 4-2.

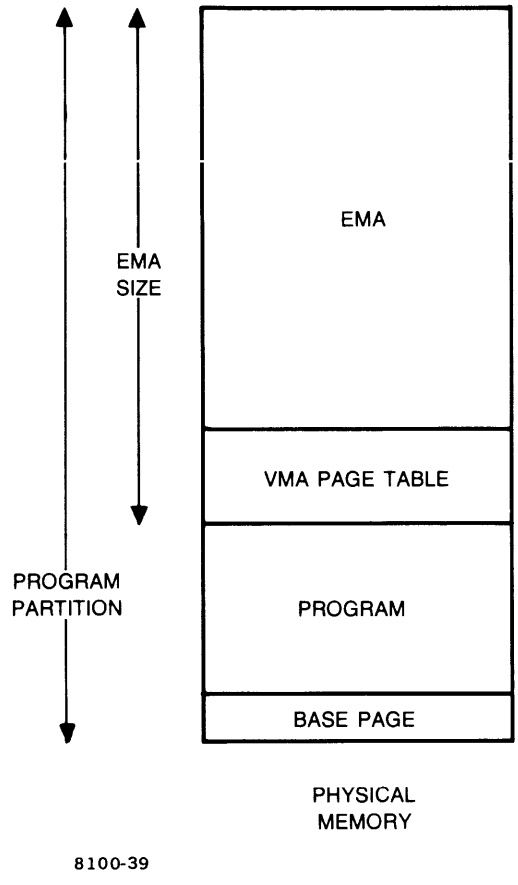


Figure 4-2. EMA In Physical Memory

Using Shareable EMA

A user program can have EMA data as local or shareable data. Shareable EMA allows large amounts of data to easily be shared among programs. Programs which use shareable EMA are written the same as programs using local EMA data. At load time a program's EMA data can be declared as shareable using the SH loader command. Note also that programs compiled by FTN4X cannot use shareable EMA.

A shareable EMA program requires two partitions available at the same time, one for the EMA data and one to execute the program code. Partitions can be declared as shareable EMA partitions and each is associated with a label during generation or reconfiguration (refer to the RTE-6/VM On-Line Generator Manual). EMA data residing in this partition can be shared by any number of programs declaring the same shareable EMA partition label at load time.

Once a shareable EMA program is dispatched and the shareable EMA partition is set up, that partition will not be released until all the EMA programs using that partition for data have terminated. The system keeps an active user count of the number of programs using each shareable EMA partition. As more programs enter the system and use that data area, the count is increased. As these programs terminate, the count is decreased. When the count reaches 0, the partition is released as a data area and is available for use like any other partition, unless the partition is locked (refer to the discussion below). Note that if a program terminates saving resources, the shareable EMA partition (a resource) is not released, and the active user count is not decreased.

In some cases the user may wish to leave the partition as a shareable EMA partition even when all programs accessing the area are dormant. This may be done in two ways. The first way is to reserve the partition at generation or reconfiguration time. The other method uses the system routine LKEMA to lock the shareable EMA partition as a data area. The partition will then remain a reserved data area until the active user count equals zero and the system routine ULEMA is called to unlock the shareable EMA partition or the user enters the UL system command to forcibly unlock the partition. Refer to the section on Programming with VMA and EMA for a description of the LKEMA and ULEMA library routines. The UL system command is described in the RTE-6/VM Terminal User's Reference Manual.

Resource numbers, described in Chapter 5, can be used to synchronize the use of shareable EMA data.

Shareable EMA Program Considerations

At load time the EMA data for Macro, Pascal and FORTRAN 77 programs can be declared as shareable data by specifying the "SH,label" loader command. The specified label is the label of the shareable EMA partition assigned at generation or reconfiguration time. Another program specifying "SH,label" will use the same shareable EMA partition. To insure that the same EMA data will be accessed by the two programs, a shareable EMA label file must be created.

CAUTION

A shareable EMA label file should be used when the programs to be loaded contain multiple EMA COMMON blocks. This binds each label to the correct address. Declaring the blocks in the same order in all programs does not guarantee that the labels will be bound to the same address in different programs. Also, local EMA should not be used in programs that use shareable EMA, since there is no guarantee that a local EMA variable will be bound to the same address in different programs.

A shareable EMA label file allocates the shared memory as specified in the file for all programs using the shareable EMA partition. The shareable EMA label file name must be the same as the shareable EMA partition label. This file must reside on LU 2. The first line of the file name must be the control command $SHEMA. The rest of the file consists of EMA COMMON block labels as shown below.

The format of the entries in the shareable EMA label file is:

```
label,size
```

where:

label = EMA COMMON block labels, 16 characters maximum.

size = EMA COMMON block size in words, double word value (10 digit maximum).

The loader uses these entries to order the EMA area of any program using the shareable EMA partition. Thus, as long as two EMA programs are loaded with the same shareable EMA label file, references to the same EMA variable will access the same EMA data. Refer to the RTE-6/VM Loader Reference Manual for further details on loading shareable EMA programs.

NOTE

When two programs use a SHEMA they must allocate the same amount of EMA space. Failure to do so results in an EM82 error for the program with the larger space declaration.

For example, suppose the following file ABC existed on LU 2 and a shareable EMA label ABC exists.

```
$SHEMA
YYY,4500
ZZZ,50000
XXX,10000
```

If PROG1 and PROG2 were both loaded using the loader option SH,ABC, then a reference to XXX in PROG1 would access the same data as a reference to XXX in PROG2.

CAUTION

A SHEMA file cannot exist if an FTN4X program is using EMA. Purge the SHEMA file on LU 2 or move to user cartridge before loading FTN4X EMA program.

Partition Considerations

A VMA/EMA (not shareable data) program can run in a RT, BG, or mother partition. If the VMA program and working set or the EMA program and data is too large to fit in a RT or BG partition, it must then run in a mother partition. When a VMA/EMA program needs to run in a mother partition or when a RT or BG program is assigned to a mother partition, more handling is involved than is the case with RT or BG partitions. An EMA program having shareable data runs in two partitions. The program code can run in a RT, BG, or mother partition, but its EMA data must reside in a shareable EMA partition. Certain considerations must be taken into account when using shareable EMA partitions.

Mother Partitions

When a VMA/EMA program needs to run in a mother partition, each subpartition is checked. If all subpartitions are either free or occupied by swappable programs, the subpartitions are marked as being used for a mother partition and all the programs in the subpartitions are swapped out. The subpartitions are then removed from all partition free lists. A subpartition cannot be made available by swapping if it contains a memory-locked program, a program that is performing unbuffered I/O, a scheduled program of higher priority, or if it is a shareable EMA partition and is currently being used as a data area. Note that the swapped-out programs may go back into any other partition large enough to accept them.

It is now apparent that when a mother partition is required and its subpartitions are in use, there may be a delay before the program can be dispatched into the mother partition.

If a mother partition is needed to dispatch a program and the partition is already allocated, the current occupant must be swapped out if the occupant's priority and status permit it. If the program to be swapped out is a VMA/EMA program with local data, the program's code and VMA working set or EMA data must both be swapped. Note that sufficient swap tracks must be available or the swap of the program does not take place, preventing the dispatching of a new program. The VMA working set or EMA area is swapped out in large 31K word blocks. Each block is written to the swap tracks on the disc until all of the VMA working set or EMA data is swapped. Because of the many disc accesses that may be needed to swap out a VMA/EMA program, caution should be exercised when assigning any program to a mother partition. Also, if several VMA/EMA programs must contend for the same mother partition, overall throughput will be reduced because one program must be swapped to make room for the other.

Subpartitions are not available for dispatching programs when the mother partition is in use by an active program. When a program in a mother partition terminates normally or is aborted, the subpartitions are released and again become available. The mother partition occupant is swapped only under the following conditions:

1. The occupant is swappable and another program needs the same mother partition.

2. The occupant is dormant (terminated with the save-resources option, operator-suspended or serially reusable), and a subpartition is needed for another program.

3. A program is assigned to a subpartition and the mother partition occupant is in a swappable state.

When a RT or BG program is scheduled and is not assigned to a partition, a search is made for a partition of the same type that is large enough to accommodate the program. If none can be found in the free list, dormant list, or in the allocated list (or it contains non-swappable programs), then the dormant mother partition list will be searched for one with a subpartition of the correct type and size. If a suitable subpartition can be found, the dormant program in the mother partition will be swapped out.

Shareable EMA Partitions

Shareable EMA partitions and their associated labels are defined at generation or reconfiguration time. The size of the shareable EMA partition is limited only by the amount of physical memory, less the operating system and program space. Access to one shareable EMA partition is allowed per program. This data area can be further subdivided using multiple labeled common blocks and array declarations as described in the appropriate FORTRAN Reference Manual.

Five restrictions are imposed upon shareable EMA partitions:

1. A shareable EMA partition can never have a program assigned to it, even though any type of program can execute in a shareable EMA partition.

2. If a subpartition is a shareable EMA partition, no program can be assigned to the mother partition or that subpartition.

3. If a mother partition is a shareable EMA partition, no program may be assigned to any of its subpartitions or that mother partition.

4. For a shareable EMA program or any progeny (i.e., sons, grandsons, etc.) there must exist in the system a partition big enough to execute the program code, and that partition cannot be a shareable EMA partition. If the only partition large enough is a mother partition, none of its subpartitions may be a shareable EMA partition.

5. The shareable EMA partition must be at least one page larger, than the EMA size declared in the program, to allow space for the page table.

Programming with VMA/EMA

Programming with VMA/EMA is available in FORTRAN, Pascal/1000, and Macro/1000. In every VMA/EMA program, one Extended Memory Area must be declared. This area can be subdivided using multiple labeled common blocks and array declarations.

In FORTRAN and Pascal/1000, calls to the VMA/EMA mapping subroutines are made without explicit user action (i.e., they are automatically emitted by the compiler). The VMA/EMA mapping subroutines described in Appendix B, must be called from Macro/1000 programs in order to map in data. Additional VMA/EMA subroutines are summarized in Table 4-2 and are described in this section. VMA/EMA errors which can occur due to the VMA/EMA subroutines are listed in Appendix A.

EMA programs compiled or assembled on previous RTE operating systems (i.e., RTE-IV or RTE-IVB) are not supported on the RTE-6/VM operating system. These EMA programs should be re-compiled or re-assembled before executing so that calls to the RTE-6/VM VMA/EMA mapping subroutines will be generated. If the source code of the EMA program is not available to be recompiled or reassembled, the relocatable code can be loaded on an RTE-6/VM operating system and run. The library $EMCLB must be searched at load time in order to convert the RTE-IVB EMA mapping routines to equivalent RTE-6/VM routines. Because these are software equivalent routines, the program will run slower. If your RTE-IVB EMA program calls the mapping routines directly, the program may not run.

All VMA/EMA subroutine call descriptions use the FORTRAN subroutine call format. If desired, the description of general formats included at the beginning of Chapter 2 can be consulted to convert the calls to FORTRAN functions, Pascal/1000, or Macro/1000 formats.

FORTRAN 77 character variables cannot be passed to a VMA file subroutine. Refer to the FORTRAN 77 Reference Manual for more information on character data.

Declaring Extended Memory Area (EMA)

The first step in programming with VMA/EMA is to declare an Extended Memory Area (EMA). Only one EMA can be defined per program. Note that the VMA programs are identical to EMA programs until load time.

An EMA is declared in FORTRAN by using the $EMA directive. This statement identifies the variables in labeled common blocks as EMA variables. The format of the $EMA directive in FORTRAN 4X is:

```
$EMA (blockname,MSEG)
```

where:

blockname	name of a labeled common block defined in a common statement.
MSEG	MSEG size in pages. If the default size is desired enter a zero. The MSEG default size is equal to 31 minus the program space.

The format of the $EMA directive in FORTRAN 77 is:

```
$EMA/block1/,/block2/,.../blockn/
```

where:

blocki	name of labeled common block defined in a common statement which is to be put in VMA/EMA.

The $EMA directive must be the first non-comment statement in the program. A $MSEG directive is availale in FORTRAN 77 for declaring the MSEG size. More information on declaring EMA in FORTRAN is provided in the appropriate FORTRAN Reference Manual.

An EMA can be declared in Pascal by using the HEAP and, optionally, the EMA compiler options:

$HEAP 2 $	The HEAP compiler option specifies that the heap/stack area resides in EMA. The MSEG size defaults to 31 minus the program space.
$EMA ema,mseg	The optional EMA compiler option allows the EMA size and MSEG size to be specified as an integer value.

These compiler options must appear before the program's heading. For more information on declaring EMA in Pascal, refer to the Pascal/1000 Reference Manual.

An EMA can be declared in a Macro/1000 program as a pseudo instruction:

```
label EMA size,mseg
```

where:

label — EMA label (must be defined).

size — VMA or EMA size in pages. If the default size is desired, enter a zero. The EMA default size can vary from zero to 1023 pages. The EMA size is determined at the time the program is first dispatched as the program's partition size minus program size. The VMA size can go up to 65536 pages.

mseg — MSEG size in pages. If the default size is desired, enter a zero. The MSEG default size is equal to 31 minus the program space.

In Macro/1000 program locations within an EMA array cannot be accessed using the EMA label with an offset, nor can EMA labels be referenced indirectly. External routines and segments can use EMA by declaring the EMA label as an external. For information on declaring the EMA label with the ALLOC or RELOC instructions, refer to the Macro/1000 Reference Manual.

Table 4-2. Common VMA/EMA Subroutines

INFORMATIONAL SUBROUTINES	
SUBROUTINE	DESCRIPTION
EMAST	Return general information about VMA/EMA.
VMAST	Return size of VMA/EMA.
I/O MANAGEMENT SUBROUTINES	
SUBROUTINE	DESCRIPTION
VMAIO	Perform VMA/EMA I/O data transfers to/from an LU.
EIOSZ	Determine the maximum guaranteed length of data transfer using VMAIO.
SHAREABLE EMA SUBROUTINES	
LKEMA	Lock a shareable EMA partition.
ULEMA	Unlock a shareable EMA partition.
VMA FILE SUBROUTINES	
SUBROUTINE	DESCRIPTION
CREVM	Create backing store file.
OPNVM	Open the backing store file.
PURVM	Purge the backing store file.
PSTVM	Post the working set to the backing store file.
CLSVM	Post the working set and close the backing store file.
VREAD	Read data from a data file into the VMA and EMA.
VWRIT	Write data from the VMA/EMA to a data file.

General Purpose VMA/EMA Subroutines

General purpose VMA/EMA Subroutines summarized in Table 4-2 are optional subroutines available to the user to:

* provide size information about VMA/EMA.
* manage I/O transfers.
* programmatically lock or unlock a shareable EMA partition.

EMAST Subroutine (Return Information on VMA/EMA)

EMAST is a subroutine that returns information about the VMA/EMA of the calling program.

```
CALL EMAST(NEMA,NMSEG,IMSEG[,IWS])
           ---- ----- -----  ---
```

NEMA	Total page size of VMA or EMA (not including page table).
NMSEG	Total page size of mapping segment (MSEG).
IMSEG	Starting logical page of MSEG.
IWS	Working Set page size (optional parameter). For an EMA program, this value is the same as NEMA.

COMMENTS:

An error return is made if VMA/EMA is not defined in the calling program.

Upon return:

A-Register = 0 if normal return

= -1 if error occurred

EXAMPLE: Check the various size parameters for the VMA program EMST.

```
$EMA(BIG,0)
      PROGRAM EMST
      COMMON /BIG/IARRAY(250000)
        :
      CALL EMAST (NEMA,NMSEG,IMSEG,IWS)
```

VMAST Subroutine (Return Size of VMA/EMA)

The VMAST subroutine determines which version of VMA/EMA the calling program is and returns the size of VMA/EMA.

```
CALL VMAST(IVMA,ISIZE)
           ---- -----
```

IVMA	Version of VMA/EMA of the calling program. Refer to the comments below for the possible values returned by the system.
ISIZE	VMA/EMA page size. If the program is not a VMA/EMA program, a zero is returned.

COMMENTS:

The following table shows the possible values that can be returned by the system when a call to VMAST is made.

PROG TYPE	IVMA	ISIZE
NOT VMA/EMA PROG	-2	0
NOT RTE-6/VM EMA PROG	-1	EMA SIZE
RTE-6/VM EMA PROG	0	EMA SIZE
RTE-6/VM VMA PROG	+1	VMA SIZE

The value of the EMA size is determined from word 28 of the ID segment. If the program is an RTE-6/VM EMA program, the EMA size returned is one less than the value in word 28. The VMA size is determined from word 3 of the ID extension plus one.

EMA programs compiled or assembled on previous RTE operating systems (i.e., RTE-IV or RTE-IVB) are not supported on the RTE-6/VM operating system. These EMA programs should be re-compiled or re-assembled before executing so that calls to the RTE-6/VM VMA/EMA mapping subroutines will be generated.

EXAMPLE: Use the VMAST routine to determine if the program is an RTE-6/VM VMA/EMA program and its VMA/EMA size.

```
$EMA(BIG)
      PROGRAM VMST
      COMMON/BIG/IARRAY (12500)
        :
      CALL VMAST (IVMA,ISIZE)
      IF (IVMA) 200,50,100
        :
   50 WRITE (1,*)'RTE-6/VM EMA PROGRAM'
        :
  100 WRITE (1,*)'VMA PROGRAM'
        :
  200
```

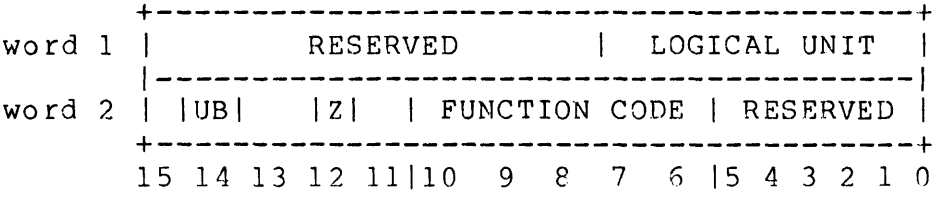
VMAIO Subroutine (Perform Large VMA/EMA Data Transfers)

The VMAIO subroutine allows the user to do large (generally up to 26 pages) I/O transfers to/from the VMA/EMA and any I/O device. This is the preferred method of performing I/O transfers to/from the I/O devices and VMA/EMA.

```
CALL VMAIO(ICODE,ICNWD,IBUFF,ILEN[,IOP1[,IOP2]])
```

ICODE I/O Request Code. Same parameter as described in the Standard I/O EXEC Requests 1 and 2.

ICNWD ICNWD is a two-word quantity with the following format:



* Logical Unit (bit 0-7) = LU of the device that data is to be transferred to or from.
* Function Code (bit 6-10) = Same as function code of the standard read/write (1/2) EXEC requests.
* The UB and Z-bit settings are described under EXEC1 and 2 calls as documented in Chapter 2.

IBUFF IBUFF is the name of the VMA/EMA array. The compiler translates this into a two-word quantity in double integer format (most significant bit first, least significant bit last). This represents the offset from the start of the buffer to be transferred.

ILEN ILEN is the length of the data transfer and cannot be greater than the length of IBUFF; positive number of words or negative number of characters. Note that the length of the transfer cannot be specified as a negative number of characters if the character count exceeds 32768 characters. Use a positive number of words should this situation arise. The maximum length is about 26K words in length. Note that the length of IBUF cannot be greater than the EMA or Working Set size

IOP1
IOP2 Optional parameters. Same parameters as described in the Standard I/O EXEC Requests 1 and 2. The buffer defined by IOP1 and IOP2 resides in the user's address space, not in the VMA area.

COMMENTS:

Since a VMA/EMA variable cannot be used as a parameter in an EXEC call, VMAIO is available to handle I/O for a VMA/EMA buffer. The VMA/EMA I/O buffer to be transferred need not be mapped into the user's logical address space prior to the request. The buffer is automatically mapped. The only requirement is that the length of the I/O transfer may not exceed the maximum length available for a VMA/EMA transfer determined by the EIOSZ subroutine (refer to the following discussion of the EIOSZ subroutine). The maximum size of the buffer is 31 minus the starting page of system common pages.

Note that any VMA/EMA mapped into logical memory prior to the call to the VMAIO subroutine is saved and is then reinstated upon return to the user's program.

If VMAIO is called from a program which is not a VMA/EMA program, the program will abort with a dynamic mapping error.

Error messages that occur due to execution of the VMAIO subroutine are contained in Appendix A.

EXAMPLE: Read from the terminal into a VMA array.

```
$EMA(BIG)
      PROGRAM VMIO
      COMMON/BIG/IARRAY(50000)
      DIMENSION ICNWD(2)
C
C   READ FROM THE TERMINAL SETTING THE
C   NO-SUSPEND BIT.
C
      ICODE=1+40000B
      ICNWD(1)=1
      ILEN=250
      CALL VMAIO (ICODE,ICNWD,IARRAY,ILEN)
      GO TO 100
        :
        :
  100 I/O SUSPEND HANDLING
        :
```

EIOSZ Subroutine (Determine Maximum Length of Transfer)

The maximum guaranteed buffer length possible for a large VMA/EMA I/O transfer can be obtained by calling the subroutine EIOSZ.

CALL EIOSZ(ISIZE) or ----- ISIZE=EIOSZ(dummy)
ISIZE The maximum length available in words is returned.

COMMENTS:

The maximum guaranteed length of the I/O transfer is 31 pages minus the starting page of system common, generally 26 pages. The maximum length is also returned in the A-Register if the call is successful. For small I/O transfers (20 pages or less), this routine does not need to be called.

On return:

A-Register = -1 if an error occurs (i.e. not VMA/EMA program).

= >0 if successful will indicate the maximum length available for transfer in words.

EXAMPLE: Determine the maximum buffer length available to transfer the VMA array using the VMAIO subroutine.

```
$EMA(BIG)
      PROGRAM ESZ
      COMMON/BIG/IARRAY (500000)
        :
      CALL EIOSZ (ISIZE)
        :
C  USE VMAIO TO TRANSFER DATA WITH
C  BUFFER LENGTH <= ISIZE
        :
```

LKEMA/ULEMA (Lock/Unlock a Shareable EMA Partition)

The subroutines LKEMA and ULEMA allow a shareable EMA partition (that was specified at load time) to be locked and unlocked by the calling program. When a shareable EMA partition is locked, this assures that the partition will be reserved as a data area which prevents execution of user programs in the partition. These subroutines must be called from a program that is using the shareable EMA partition to be locked or unlocked.

```
CALL LKEMA

    or

CALL ULEMA
```

COMMENTS:

These subroutine calls are ignored if the program does not use EMA, or if it does not use shareable EMA. If LKEMA is called to lock a shareable EMA partition which is already locked, the call is ignored. If ULEMA is called to unlock a shareable EMA partition that is already unlocked, the call is also ignored.

Normally, a shareable EMA partition is released once the number of programs actively using it drops to zero. If this partition is locked, it is not released for use by other programs until it is unlocked either through the ULEMA system library subroutine or by using the UL system command description in the RTE-6/VM Terminal User's Reference Manual.

EXAMPLE: PROG1 places data into a shareable EMA partition, locks the partition, and time schedules PROG2. PROG2 prints the data from the shareable EMA partition and unlocks the partition.

```
$EMA(BIG)
      PROGRAM PROG1
      COMMON/BIG/IARRAY (50000)
      DIMENSION INAM(3), ICNWD(2)
      DATA INAM/6HPROG2 /
C
C  ENTER DATA FROM THE TERMINAL INTO THE
C  EMA ARRAY (SHAREABLE EMA PARTITION)
C
      ICODE=1
      ICNWD(1)=1
      ILEN=25000
      CALL VMAIO (ICODE,ICNWD,IARRAY,ILEN)
C
C  LOCK THE PARTITION
C
      CALL LKEMA
C
C  SCHEDULE PROG2
C
   ICODE=12
   CALL EXEC (ICODE,INAM,3,0,-5)
   END
```

```
$EMA(BIG)
      PROGRAM PROG2
      COMMON/BIG/JARRAY(25000)
      DIMENSION ICNWD(2)
C
C  DISPLAY DATA (FROM SHAREABLE EMA PARTITION)
C  ON THE TERMINAL
C
      ICODE=2
      ICNWD(1)=1
      ILEN=25000
      CALL VMAIO (ICODE,ICNWD,JARRAY,ILEN)
C
C  UNLOCK THE PARTITION
C
      CALL ULEMA
      END
```

VMA File Subroutines

The Demand-paged Virtual Memory system transparently manipulates the working set and backing store file for the user. However, VMA file subroutines are available to allow the user to create, open, or close the virtual memory backing store file with various options. This allows the user to programmatically create or use an existing file as the backing store file in a wide variety of ways instead of the standard default ways of the virtual memory system. The backing store file must be a type 2 file with record length of 1024 words.

If the user chooses to use the standard default backing store file created by the virtual memory system, the file will be created on the first available cartridge in the user's cartridge list. The default backing store is a type 2 file in 256 block increments. The default file name is:

XXXYVM

where:

XXX = ID segment number of the VMA program.

Y = CPU number.

The default backing store file is automatically purged when the program terminates.

Using the VMA file subroutines, the user can declare an existing file as the backing store file. This provides the user with the option of having "initialized Virtual Memory" (refer to the OPNVM subroutine). In addition, the user can declare the backing store file to be read only in order to prevent inadvertant changes to the file.

All subroutine descriptions in this section use the FORTRAN subroutine call format. If desired, the format can be converted to Pascal/1000 or Macro/1000 subroutine calls using the general call formats described in Chapter 2. To call the VMA file subroutines, the program must be a VMA program. Appendix A contains a list of possible VMA errors.

CREVM Subroutine (Create a VMA Backing Store File)

The user program can create the backing store file with several options to be used by the virtual memory system by calling the CREVM subroutine.

```
CALL CREVM([NAME[,IERR[,IOPTN[,ISC[,ICR]]]]])
           ----
```

NAME File name. A three-word array containing the ASCII name of the file to be created.

IERR Error return. A one-word variable that contains a zero for successful calls.

IOPTN File options. The file options are set as follow:

- Bit 0 = 1 A non-scratch file (file NAME) is to be created and used as the backing store file. If bit 0=0, then NAME and bit 1 are ignored.
- Bit 1 = 1 File is to be opened if the create fails due to a duplicate file error.
- Bit 2 = 1 File create is to be deferred until the working set needs to be written to the file.
- Bit 3 = 1 File extents are not to be addressed or created.

ISC File security code.

ICR File cartridge reference number (CRN).

All parameters are optional. If no parameters are passed a default scratch file is created upon execution of the CREVM call, rather than being deferred.

COMMENTS:

When a file name is specified in the CREVM subroutine, the created file is opened for updating and is then closed at program completion. The CREVM subroutine creates a type 2 file in 256 block increments, with record length equal to 1024 words, creating as many extents as necessary to contain the VMA array. If the contents of the working set is to be saved in this backing store file, use the PSTVM or CLSVM subroutine described later in this section. If a file name is not specified (or bit 0=0), the default VMA scratch file is created. This default file is purged at program completion.

NORMAL RETURN--Upon normal return the IERR parameter and the A-Register are zero. However, if the deferred create option was specified it is possible that the file cannot be created. This error will show up when creation of the backing store becomes necessary.

ERROR RETURNS--When an error occurs during the subroutine call, a positive VMA error code is returned in the IERR parameter and in the A-Register.

Example: Create a file called VMDATA (on disc cartridge VM) to be used by the program TSVM1 as the backing store file when needed.

```
$EMA (BIG)
      PROGRAM TSVM1
      COMMON/BIG/IELEMB (65535)
      DIMENSION INAM (3)
      INTEGER*4  IELEMB
      DATA INAM/6HVMDATA/
C
C  CREATE VMA FILE
C
C  SET IOPTN TO BIT 0=1 and BIT 2=1
C  INDICATING NON-SCRATCH FILE AND DEFERRED CREATE
      IOPTN = 5
      ISC = 2HCS
      ICR = 2HVM
      IERR = 0
C
      CALL CREVM (INAM,IERR,IOPTN,ISC,ICR)
      IF (IERR.NE.0) GOTO 300
       :
       :         !ENTER VMA DATA IN THE BACKING STORE FILE
       :         !AND CLOSE THE FILE
 300 error message
```

OPNVM Subroutine (Open a VMA Backing Store File)

The OPNVM subroutine allows the program to specify the backing store file to be opened in a variety of ways by the virtual memory system. Refer to the OPEN FMP call for a detailed explanation of the file options.

```
CALL OPNVM(NAME[,IERR[,IOPTN[,ISC[,ICR]]]])
           ----
```

NAME	File name. Three-word array containing the ASCII name of the file to be opened.
IERR	Error return. A zero is returned to indicate a successful call.
IOPTN	File options. The file options are set as follows:
	Bit 0 = 1 File is to be opened for non-exclusive use.
	Bit 1 = 1 File is to be opened for update (refer to the comments below).
	Bit 2 = 1 File open is to be deferred until required.
	Bit 3 = 1 File extents are not to be addressed or created.
	Bit 4 = 1 Read-only access to disc file.
ISC	Security code. The contents of ISC must be equal to the security code of the file being opened, except if for read only access.
ICR	Cartridge Reference Number (CRN).

COMMENTS:

The backing store file opened by the OPNVM subroutine must be a type 2 file and must be at least 256 blocks long or else your program may abort with VM05 error. Files opened by OPNVM subroutine should be closed with the CLSVM subroutine. If the contents of the working set is to be written to the file, use the PSTVM or CLSVM subroutine described later in this section.

The backing store file is considered to be initialized only if opened in the update mode (bit 1 of the IOPTN parameter set), thus allowing the user to modify existing data in the backing store file. When this VMA array is first accessed, the corresponding data in the backing store file is swapped into the working set. If the update bit is not set, the backing store file is considered to contain uninitialized data. In this case, pages from the backing store file are not swapped into memory until a page of the working set has been swapped to the disc.

NORMAL AND ERROR RETURNS--Normal and error returns are the same as those for the CREVM subroutine.

EXAMPLE: Open the VMA backing store file called TSTDTA when required for updating.

```
$EMA (BIG)
      PROGRAM TSVM2
      COMMON/BIG/IELEMB (65535)
      DIMENSION INAM (3)
      INTEGER*4 IELEMB
      DATA INAM/6HTSTDTA/
C
C  OPEN THE VMA FILE
C
C  SET IOPTN to BIT 1=1 and BIT 2=1
      IOPTN = 6
      ISC = 2HCS
      ICR = 2HVM
      IERR = 0
C
      CALL OPNVM (INAM,IERR,IOPTN,ISC,ICR)
      IF (IERR.NE.0) GOTO 300
       :
C  MANIPULATE THE VMA ARRAY IN THE FILE TSTDTA
       :
  300 error message
```

PURVM Subroutine (Purge VMA Backing Store File)

The subroutine PURVM can be called to purge the backing store file.

```
CALL PURVM
```

EXAMPLE: Purge the VMA file created by the CREVM call.

```
$EMA (BIG)
      PROGRAM TSVM3
      COMMON/BIG/IELEMB (65535)
      DIMENSION INAM (3)
      INTEGER*4 IELEMB
         :
C   CREATE VMA FILE
         :
C   MANIPULATE FILE
         :
C   PURGE SCRATCH FILE
      CALL PURVM
         :
```

PSTVM Subroutine (Post Working Set to Disc)

The subroutine PSTVM can be called at anytime to post the entire working set (the pages of VMA data that are presently in memory) to the VMA backing store file. Note that the page table is left unchanged. If the file is opened with the read-only option, no posting will occur.

```
CALL PSTVM
```

COMMENTS:

If the user opened or created his own backing store file via the OPNVM or CREVM subroutines, the PSTVM or CLSVM subroutine should be called at the end of the program to guarantee that the virtual memory currently in memory (the working set) is posted to the disc.

CLSVM Subroutine (Close the VMA Backing Store File)

The CLSVM subroutine posts all pages of the working set in memory to the VMA backing store file on disc and executes an FMP close on the VMA backing store file. If the default VMA backing store file is opened or created, it is purged at program completion.

```
CALL CLSVM
```

COMMENTS:

If the user opened or created his own backing store file via the OPNVM or CREVM subroutines, the PSTVM or CLSVM subroutine should be called at the end of the program to guarantee that the virtual memory currently in memory (the working set) is posted to the disc. If the file is opened with the read-only option, no posting will occur.

EXAMPLE: Close the VMA file opened by the OPNVM call.

```
$EMA (BIG)
     PROGRAM TSVM4
     COMMON/BIG/IELEMB (65535)
     DIMENSION INAM (3)
     INTEGER*4 IELEMB
       :
C  OPEN VMA FILE
       :
C  UPDATE FILE
       :
C  CLOSE THE VMA FILE
     CALL CLSVM
       :
```

VREAD Subroutine (Read Data From a File to a VMA/EMA)

The VREAD subroutine allows the user to read records from a data file into a VMA/EMA array. This subroutine is similar to the FMP READF call.

CALL VREAD(IDCB,IERR,IARRAY,IDL[,ILEN[,INUM]])

IDCB Data Control Block (DCB). An array of 144+n words where n is positive or zero; previously specified in a create or open operation.

IERR Error return. A one-word variable in which a non-zero error code is returned for unsuccessful calls. Zero is returned for successful calls. The following values will be returned in IERR:

0	=	normal return	1	=	request parameter error
<0	=	FMP error	2	=	VMA/EMA mapping error

IARRAY Data Transfer Destination Start Address in VMA/EMA. This is a two-word variable representing the offset from the start of the buffer to be transferred. This offset is automatically set up by the FORTRAN compiler when the call to VREAD is made and the array is in the VMA/EMA area. This value must be positive. Refer to the following example.

IDL Data Length Requested. A one-word variable that specifies the positive number of words to be read. If the file is not type 1 or 2, the length of the request cannot exceed the size of a MSEG. This parameter is the same as the IL parameter of the READF FMP call for values greater than or equal to zero (refer to Chapter 3). For type 1 files, the IDL parameter is considered unsigned (no sign bit) to allow for a maximum data length of 65535.

ILEN Data Length Read. An optional one-word variable in which the actual number of words read is returned. Set in the same manner as the LEN parameter in a READF call.

INUM Record Number. An optional one-word variable used to specify the record number to be read (if positive) or the number of records to backspace (if negative). Used only for type 1 or 2 files. If omitted, the record at the current position is read. For further information on positioning with INUM, refer to the FMP READF call.

COMMENTS:

Type 1 files and large MSEGs will provide a very high throughput of data. If the file is not type 1 (or opened as type 1), the length of the request cannot exceed the size of the MSEG.

EXAMPLE: Read data from the file FNAME into the VMA/EMA array of the program.

```
$EMA (AREA)
      PROGRAM EXPL1
      COMMON/AREA/IARRAY (200,200) !IARRAY IS A VMA/EMA ARRAY
      DIMENSION IDCB (144),INAM(3)
      DATA INAM/6HFNAME /
      CALL OPEN (IDCB,IERR,INAM)              !OPEN FILE FNAME
      IF (IERR.LE.0) GO TO 100
          :
      CALL VREAD (IDCB,IERR,IARRAY,128)        !READ FROM FILE
      IF (IERR.NE.0) GO TO 100                    !INTO IARRAY
          :
      CALL CLOSE (IDCB,IERR)                  !CLOSE FILE FNAME
      IF (IERR.NE.0) GO TO 100
          :
      STOP
  100 WRITE (1,'("ERROR CODE = ",I5)') IERR
```

VWRIT Subroutine (Write Data From a VMA/EMA to a File)

The VWRIT subroutine allows the user to write a record of data from a VMA/EMA array into a data file. This subroutine is similar to the FMP WRITF call.

```
CALL VWRIT(IDCB,IERR,IARRAY,IDL[,INUM])
           ----
```

IDCB Data Control Block (DCB). An array of 144+n words where n is positive or zero; previously specified in a create or open operation.

IERR Error return. A one-word variable in which a non-zero error code is returned for unsuccessful calls. Zero is returned for successful calls. The following values will be returned in IERR:

0	= normal return	1	= reqest parameter error
<0	= FMP error	2	= VMA/EMA mapping error

IARRAY Data Transfer Destination Start Address in VMA/EMA. This is a two-word variable representing the offset from the start of the buffer to be transferred. This offset is automatically set up by the FORTRAN compiler when the call to VWRIT is made and the array is in the VMA/EMA area. This value must be positive. Refer to the following example.

IDL Data Length Requested. A one-word variable that specifies the positive number of words to be written. If the file is not type 1 or 2, the length of the request cannot exceed the size of a MSEG. This parameter is the same as the IL parameter of the WRITF FMP call for values greater than or equal to zero (refer to Chapter 3). For type 1 files, the IDL parameter is considered unsigned (no sign bit) to allow for a maximum data length of 65535.

INUM Record Number. An optional one-word variable word to specify the record number to be written (if positive) or the number of records to backspace (if negative). Used only for type 1 or 2 files. If omitted, the record at the current position is read.For further information on positioning with INUM, refer to the FMP READF call.

COMMENTS:

Type 1 files and large MSEGs will provide a very high throughput of data. If the file is not type 1 (or opened as type 1), the length of the request cannot exceed the size of the MSEG.

EXAMPLE: Write from the VMA/EMA array, IARRAY, to the file FNAME.

```
$EMA (AREA)
      PROGRAM EXPL2
      COMMON/AREA/IARRAY (20000)  !IARRAY IS A VMA/EMA ARRAY
      DIMENSION IDCB (144),INAM(3)
      DATA INAM/6HFNAME /
      CALL OPEN (IDCB,IERR,INAM)            !OPEN FILE FNAME
      IF (IERR.LE.0) GO TO 100
          :
      CALL VWRIT (IDCB,IERR,IARRAY,128)    !WRITE VMA IARRAY
      IF (IERR.NE.0) GO TO 100                    !TO FILE
          :
      CALL CLOSE (IDCB,IERR)               !CLOSE FILE FNAME
      IF (IERR.NE.0) GO TO 100
          :
      STOP
  100 error processing
```

Examples Using VMA File Subroutines

The following example demonstrates the use of VMA file subroutines in a program. The program uses an existing file as a backing store file and adds data from a data file to the VMA array.

```
$EMA(BIG)
      PROGRAM VMAEX
C
C     This program opens the existing file VMDATA (type 2) as
C     the backing store file, and allows the user to add data
C     from a data file (type 1) to the VMA backing store file.
C
      COMMON/BIG/IARRAY(2048),JARRAY(2500)
      DIMENSION INAM(3),IDCB(144),JNAM(3)
      DATA INAM/6HVMDATA/,JNAM/6HDATA  /
C
C     Open the backing store file in update mode.
      ISC=2HCS
      IOPTN=2
      CALL OPNVM(INAM,IERR,IOPTN,ISC)
      IF (IERR .NE. 0) GO TO 1000
C
C     Open the data file.
      CALL OPEN(IDCB,IERR,JNAM)
      IF (IERR .LE. 0) GO TO 1200
C
C     Read from the data file into the VMA array,JARRAY
C
   50 CALL VREAD(IDCB,IERR,JARRAY,128,ILEN)
      IF (IERR .EQ.-12) GO TO 100     !END OF FILE WAS REACHED
      IF (IERR .NE. 0) GO TO 50
C
C     The end of the data file was read. Close the backing store
C     file and the data file.
C
  100 CALL CLSVM
      CALL CLOSE(IDCB,IERR)
      IF (IERR. LT. 0) GO TO 1200
      STOP
C
C     Error processing
C
 1000 WRITE(1,'("VMA FILE ERROR, IERR = ",I6)')IERR
      STOP
 1200 WRITE(1,'("FMP ERROR, IERR = ",I6)')IERR
      END
```

Chapter 5
Applications Using System Routines

Introduction

RTE-6/VM operating systems are delivered with a collection of relocatable subroutines that comprise the system and math/formatter. This group of subroutines is specific to RTE-6/VM operating systems and is used to interface user programs with system services.

After the introduction of the libraries, the chapter discusses sample programs for Class I/O, resource numbers, LU locks, and parameter passing. The remainder of the chapter includes a functional arrangement of other useful RTE-6/VM library subroutines.

Calling Library Subroutines

Library subroutines are called by user programs and are linked to the caller either at generation or load time. These subroutines can be called either by disc-resident or memory-resident programs.

Subroutines referenced by disc-resident programs are appended to the end of the calling program and linked to it either by the loader (LOADR or MLLDR) or the On-Line Generator.

During generation, subroutines referenced by memory-resident programs will be placed in the memory-resident library. These subroutines must either be reentrant or privileged. Several memory-resident programs can then share one subroutine, which can save considerable space in the memory-resident area. Disc-resident programs cannot access routines in the memory-resident library, therefore, copies of these subroutines are created by the generator, placed in the disc-resident library, and appended the disc-resident programs.

If only one memory-resident program is to access a subroutine, it is advantageous to make it a Type 7 subroutine to force it to be appended to the calling program. A Type 7 subroutine is not placed in the memory-resident library and therefore need not be privileged or reentrant. This results in faster execution, since the subroutine will not incur the overhead associated with reentrant or privileged subroutines.

Reentrant Subroutine Structure

A subroutine must meet two criteria to be reentrant:

1. It must not modify any of its own instructions.

2. It must save all temporary results if it is to be called again before completing its current task.

A subroutine saves temporary results in a Temporary Data Buffer (TDB) that the operating system ensures is unique to each program. For example, assume PROGA is executing a reentrant subroutine that is interrupted by PROGB. If PROGB then begins execution of the same subroutine, the system saves PROGA's variables in a TDB in System Available Memory (SAM) until PROGA resumes execution, at which time it restores the proper TDB.

Each time a reentrant subroutine begins executing, the address and length of its temporary data block are transferred to the operating system through the entry point $LIBR in order to save the data. At the end of execution, the reentrant subroutine again calls RTE-6/VM through entry point $LIBX to restore any previous temporary data.

The reentrant subroutine structure is used for subroutines with an execution time exceeding one milli-second. However, for shorter execution times, the overhead time the system uses in saving and restoring temporary data makes reentrant structure unreasonable. Faster subroutines can be structured as privileged.

NOTE

A library (Type 6) subroutine can only call another library subroutine or Table Area I or Table Area II entry points.

The format and calling sequence for reentrant subroutines is as follows:

```
        NAM   xxxxx,6
        EXT   $LIBR,$LIBX
ENTRY   NOP                   Entry point of subroutine
        JSB   $LIBR           Tell system to go reentrant
        DEF   TDB             Address of temporary data
          .
          .                   Subroutine instructions go here
          .
EXIT    JSB   $LIBX           Tell system reentrant run is finished
        DEF   TDB             Address of temporary data
        DEC   N               Return adjustment
                                 (Return point=N+ENTRY)

TDB     NOP                   System-supplied link to previous TDB
        DEC   K               Total length of current TDB in words
        NOP                   System-returned address to calling
                                 program

        BSS   K-3
          -
          -
          -                   Temporary data (K-3 words)
          -
```

Privileged Subroutine Structure

Privileged subroutines execute with the interrupt system turned off. This feature allows many memory-resident programs to use a single privileged subroutine without incurring reentrant overhead. As a result, privileged subroutines need not save temporary data buffers but must execute very rapidly (<1 millisecond) to minimize the time that the interrupt system is disabled.

Since privileged subroutines disable the interrupt system, EXEC calls are illegal within a privileged subroutine. If one is attempted, the calling program will be aborted with an EX error.

The format and calling sequence for privileged subroutines is as follows:

```
        NAM  xxxx,6
        EXT  $LIBR,$LIBX
ENTRY   NOP                 Entry point to the routine
        JSB  $LIBR          Call the system to disable the
                              interrupt system and memory
                              protect fence
        NOP                 Denotes privileged format
         .
         .                  Subroutine instructions
         .
EXIT    JSB  $LIBX          Call the system to return to calling
                              program, and to enable interrupts
                              and memory protect fence
EXIT1   DEF  ENTRY          Return address
```

It is also possible to go privileged in a block of in-line code, as follows:

```
         -
         -
        JSB  $LIBR          Go privileged
        NOP                 Denotes privileged format
         -                  First instruction
         -
         -
         -
        JSB  $LIBX          Leave privileged status
        DEF  *+1            Both DEF's are required
        DEF  *+1
```

For greater flexibility, subroutines can be treated as reentrant or privileged when they are generated into the system or as ordinary routines if they are relocated with LOADR or MLLDR. Two library routines, .ZPRV for privileged subroutines, and .ZRTN for reentrant subroutines, cause subroutine code to be modified when it is relocated by the generator.

For subroutines that call .ZRTN, the generator will modify the code as follows:

Original Code	Modified at Generator Relocation
ENTRY NOP	ENTRY NOP
JSB .ZRTN	JSB $LIBR
DEF EXIT	DEF TDB
.	Body of subroutine
.	.
.	.
EXIT ENTRY,I	EXIT JSB $LIBX
DEF TDB	DEF TDB
DEC 0	DEC 0

For subroutines that call .ZPRV, the generator will modify the code as follows:

Original Code	Modified at Generator Relocation
ENTRY NOP	ENTRY NOP
JSB .ZPRV	JSB $LIBR
DEF EXIT	DEF TDB
.	Body of subroutine
.	.
.	.
EXIT JMP ENTRY,I	EXIT JSB $LIBX
DEF ENTRY	DEF ENTRY

Memory-Resident Library

The memory-resident library area in RTE-6/VM contains only Type 6 subroutines that are referenced by memory-resident programs and Type 14 subroutines forced into the memory-resident library at generation time.

Reentrant and privileged subroutines may be placed in the memory-resident library during generation by either of the following methods:

1. If the routine is declared as an external (called) by a memory-resident (Type 1) program, or is called by another memory-resident library subroutine, the subroutine will be automatically placed in the memory-resident library by the generator.

2. The routine can be changed to a Type 14 subroutine during the Parameter Input phase of generation (it also could have been assembled as a Type 14 subroutine).

NOTE

After the relocation of the memory-resident library and all memory-resident programs, all Type 6 routines are converted to Type 7 routines (making them available to disc-resident programs).

Not all subroutines referenced by memory-resident programs are necessarily loaded into the memory-resident library. By declaring the subroutine to be Type 7, the user can ensure that the subroutine will be loaded with the program. Then if .ZRNT and .ZPRV are used instead of $LIBR and $LIBX, the subroutine will execute faster since the system does not need to do the reentrant or privileged processing prior to executing the subroutine.

Utility Subroutine Structure

Utility subroutines are subroutines that cannot be shared by several programs because of internal design or I/O operations. A copy of a utility subroutine is appended to every program that calls it. The PAUSE subroutine is a typical example of a utility subroutine.

When the RTE system is generated, all library subroutines other than Type 8 subroutines are converted to Type 7 utility subroutines following the relocation of memory-resident programs. All required utility subroutines are then relocated immediately following each user program that references them during program relocation.

Format of Routines

The subroutines in each section are presented in the following format:

NOTE

If a parameter is underlined in a subroutine call description, the value is a variable returned by the system. The parameter value is not supplied by the user.

Name	The name of the routine in the NAM record.
Purpose	The use of the routine.
Entry Points	The entry points to the routine.
Assembly	The Macro/1000 assembly language calling sequence for each entry point. "A" and "B" indicate the A- and B-Registers.
FORTRAN 4X FORTRAN 77	A statement of whether the routine is callable or NOT callable in FORTRAN 4X or FORTRAN 77. Where the routine cannot be called by either version, the statement header is given as FORTRAN with no version identifiers.

Pascal — A statement of whether the routine is callable or NOT callable in Pascal.

Parameters — An explanation of the parameters form and value.

Result — The type of result and the registers (if any) where it is returned.

Errors — A summary of the error conditions reported by the subroutine. Errors generated by external references are not described. Refer to Appendix A for a fuller discussion of error messages.

Program Type — The three types of library subroutines are Utility (U), Reentrant (R), and Privileged (P).

External References — Other subroutines that are called by the subroutine. All external references except EXEC, $OPSY, REIO, IFBRK, .ZPRV, and .ZRNT are entry points in a library. EXEC and $OPSY are system entry points. IFBRK and REIO are system library entry points.

These symbols receive special handling by the RTE-6/VM generators and loaders. In RTE, both JSB .ZPRV and JSB .ZRNT are changed to RSS unless the routine is generated into the resident library. If the routine is in the resident library, the generator modifies its code as follows:

```
ENTRY NOP           -> ENTRY NOP
      JSB .ZPRV     ->       JSB $LIBR
      DEF EXIT      ->       NOP
       :                      :
EXIT  JMP ENTRY,I   -> EXIT  JSB $LIBX
      DEF ENTRY     ->       DEF ENTRY
```

```
ENTRY NOP           -> ENTRY NOP
      JSB .ZRNT     ->       JSB $LIBR
      DEF EXIT      ->       DEF TDB
       :                      :
EXIT  JMP ENTRY,I   -> EXIT  JSB $LIBX
      DEF TDB       ->       DEF TDB
      DEC 0         ->       DEC 0
```

$LIBR and $LIBX are system entry points that allow multiple RTE-6/VM programs to share code.

Notes — Additional information for using the routine.

Routines Callable From FORTRAN

Using FORTRAN 4X, routines are callable as a function or subroutine; examples are ABS(x) and RMPAR (IBUF), respectively. The routines must also be in the appropriate calling sequence as well as either the proper function or subroutine format to be callable from FORTRAN 4X; otherwise, they are not callable.

Using FORTRAN 77, some routines are callable under the same conditions as Pascal. Refer to the FORTRAN 77 Reference Manual for a discussion of compatibilities between the two languages.

Routines Callable From Pascal

Using Pascal, routines are callable if the calling sequence is either in the standard calling sequence format or one of the special calling sequences supported by the compiler. For additional information, refer to the Pascal/1000 Reference Manual.

Routines that return results in the A-Register or in the A- and B-Registers are callable from Pascal as functions.

Routines that have names containing characters that are not allowed in Pascal identifiers, such as the leading dot in .RTOI, must have EXTERNAL declarations using the ALIAS compiler option. For example,

```
FUNCTION power
   $ALIAS '.RTOI',DIRECT,ERROREXIT$
   (x=REAL
    i=INT)
   ;REAL;EXTERNAL;
```

All of the routines in this manual operate on one-word addresses only. VAR parameters in HEAP 2 programs should be declared with the $HEAPPARMS OFF$ compiler option in the EXTERNAL declaration.

The calling sequences supported by the compiler are:

1. Standard:

```
    JSB routine
    DEF *+n+1
n(DEF)
```

2. Direct:

```
    JSB routine
n(DEF)
```

3. Standard, Errorexit:

```
    JSB routine
    DEF *+n+1
n(DEF)
    JSB ERRO
```

4. Direct, Errorexit:

```
    JSB routine
n(DEF)
    JSB ERRO
```

A routine is not callable from Pascal if it has any other calling sequence. For example, Pascal does not support passing parameters in registers.

Microcoded Subroutines

Fast FORTRAN Processor

The HP 2100 and 21MX computers have, as an option, a Fast FORTRAN Processor (FFP). The HP 12907 FFP is optional for the HP 2100 computers and the HP 12977 FFP is optional for the HP 21MX computers. The FFP Firmware feature provides for faster execution of the following routines:

.GOTO ..MAP .ENTR .ENTP DBLE

SNGL .XMPY .XDIV .DFER .XFER

.XADD .XSUB .SETP (DOS-III only)

The following additional Relocatable Subroutine entry points are available only in HP 12977 FFP:

.PWR2 .XPAK .FLUN .XCOM .PACK ..DCM DDINT

No change from the calling sequence defined in this manual is required to use these routines in FFP, if installed. The user should be aware that after the first execution of a subroutine call, "JSB .GOTO" for example, the main memory location containing the JSB is modified to hold a branch to the ROM address where the .GOTO microcode begins.

Floating Point Library

A second microcode option on the HP 2100 computer, HP 12901 Floating Point, provides firmware for faster execution of the following routines:

.FAD .FSB .FMP .FDV FLOAT IFIX

These routines are implemented in the same fashion as the FFP routines. (The HP 21MX computers include the floating point firmware as part of the basic instruction set.)

Syntax Conventions

In the sections that follow, certain conventions are used to describe System Library Subroutine calls.

* Parameters that are underlined have values returned by the system (e.g., the value is not supplied by the user). An example of this is:

```
CALL RNRQ(IP1,IP2,IP3)
                  ---
```

* Parameters that are double underlined have values that are system-supplied in some cases and user-supplied in other cases. The comments associated with the call description should be consulted for details concerning their use. An example of this is:

```
CALL RNRQ(IP1,IP2,IP3,IP4)
                  ===
```

* Parameters enclosed in square brackets are optional. An example of this is:

```
CALL RNRQ(IP1[,IP2])
```

* Parameters with no qualifiers, i.e., square brackets, angle brackets, or underlines, are required and their value is supplied by the user.

Class I/O Applications

The range of possible areas where Class I/O could be used to improve applications program performance is wide and varied. The examples illustrated in this section are intended only to demonstrate some of the considerations and procedures used in designing specific applications.

Each example uses the preferred method of allocating class numbers via the CLRQ library subroutines. This routine is described in detail below and is followed by two application examples.

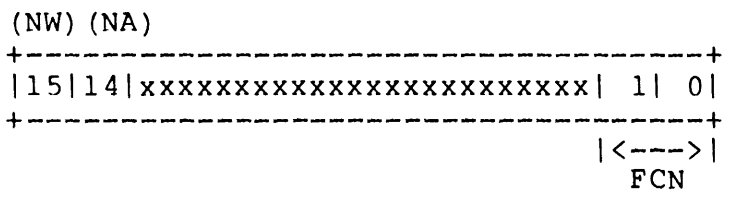
CLRQ — Class I/O Management

The routine CLRQ allows the assignment of class ownership so that in the event of a program terminating or aborting without cleaning up the class numbers and class buffers assigned to it, the system will be able to deallocate these resources. This routine also allows programmatic flushing of pending class buffers on an LU or flushing of all class buffers (pending or completed) with deallocation of the class resource itself.

CALL CLRQ (IFUNC,ICLAS[,IOP])
IFUNC - Class Management control function (refer to comments below). ICLAS - Class number. IOP1 - Optional parameter used to describe a program name or LU (refer to the comments below).

COMMENTS:

CONTROL FUNCTION (IFUNC)---The format of the control function is as follows:



The function of the control function fields are described below:

* "No-Wait" Bit (bit 15)--When set the calling program does not suspend if no class numbers are available when the CLRQ request is made. The A-Register will contain a -1 if no class numbers are available, otherwise, the A-Register will contain a zero which means the request completed without error.

* "No-Abort" Bit (bit 14)--When set this bit operates the same as the no-abort bit in the ICODE parameter of the EXEC calls. This allows the user to maintain program activity if an error is made in the call sequence to CLRQ or some other programming activity. When set the registers will contain an ASCII error message. The A-Register will contain the first two ASCII characters and the B-Register will contain the second two ASCII characters of the four character message.

FCN=1 Class ownership is assigned. If IOP1 contains the name of a program, the program is assigned ownership of the class specified in ICLAS. If IOP1 is zero no ownership is assigned. If IOP1 is defaulted, which in this case means omitting the parameter from the call, the calling program is assigned ownership. If ICLAS is zero, a new class number is allocated by the call. When a program is the "owner" of a class, the system then knows when that class can be deallocated. When the program becomes dormant, then the system will deallocate the class and its associated buffers.

FCN=2 Flush class requests and deallocate the class specified in ICLAS. All non-active pending requests will be deallocated. Abort requests will be issued by the system for all active I/O requests, in which case the buffer will be deallocated at the completion of abort processing. All previously completed requests will be immediately deallocated. The class table entry will be flagged so that no new requests will be issued on the class. An I/O error (IO00) will be returned to programs that do issue a request on the class after the class table entry is flagged. When the pending class request count in the class table entry reaches zero, the system will deallocate the class. Note that the IOP1 is not used.

FCN=3 Flush class requests on LU designated by IOP1. The system looks at the class table entry specified in ICLAS. Non-active requests on ICLAS that are pending on the LU specified in IOP1 are deallocated. If a request is active, an abort request is issued by the system. The buffer will be deallocated when the active EXEC request completes. The class number is not deallocated nor are the completed class buffers affected.

ICLAS ICLAS is the class number that can be owned by a program. The class number format is the same as in the EXEC 17, 18, 19, and 20 requests. This parameter works in conjunction with FCN and IOP1. For example, when this parameter is zero and FCN=1 then a new class number will be assigned to the calling FCN program and returned in ICLAS when the call completes.

IOP1 An optional parameter that works with FCN and ICLAS in a number of ways. Refer to the above descriptions for details.

GENERAL FLOW

The system will check all terminating and aborting programs for class ownership. If ownership exists, all completed class request buffers will be deallocated. If the program terminates without terminating its I/O requests, such as a program that does a Class Read and then terminates, the pending class requests will be allowed to be completed normally. If I/O is to be aborted, all non-active pending requests are flushed, and the drivers will be issued an abort request for all active requests. In the latter case, the buffer and the class will be automatically deallocated by the system when abort processing has completed. Refer to Appendix A for possible CL errors that can be returned due to a CLRQ call.

EXAMPLE 1: Allocate two class numbers, assigning one to the calling program and the second to the program called PROG2. PROG2 must have an ID segment or an error (SC05) will result. The no-abort bit is set in the function parameter to prevent the program from being aborted.

```
      PROGRAM ALLOC
      IMPLICIT INTEGER (A-Z)
      DIMENSION IOP1 (3)

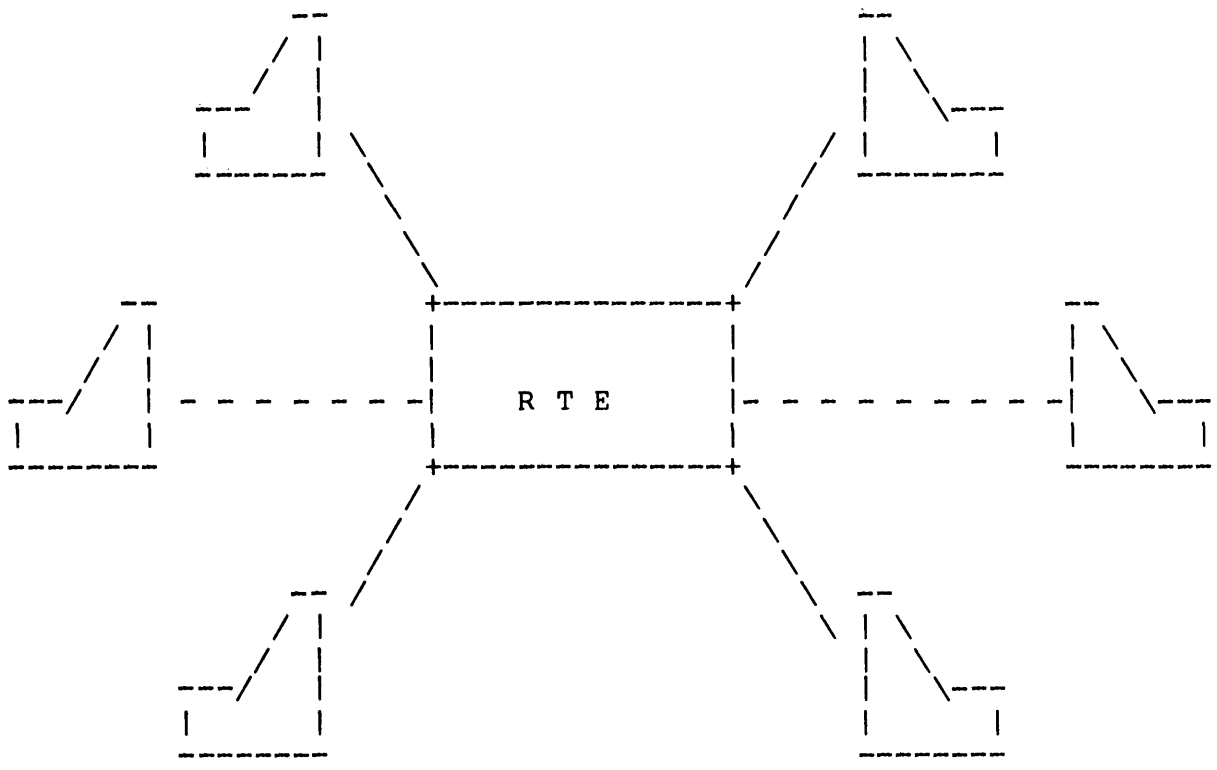
      ICLAS1 = 0
      IFUNC  = 1
C     ALLOCATE FIRST CLASS NUMBER TO THE CALLING PROGRAM
      CALL CLRQ (IFUNC + 40000B, ICLAS1)

C     CHECK ERROR
      GOTO 100
C     NOW ALLOCATE THE 2ND CLASS NUMBER, ASSIGNING IT TO P2
   50 ICLAS2 = 0
      IOP1(1) = 2HPR
      IOP1(2) = 2HO6
      IOP1(3) = 2H2
      CALL CLRQ (IFUNC + 40000B, ICLAS2,IOP1)

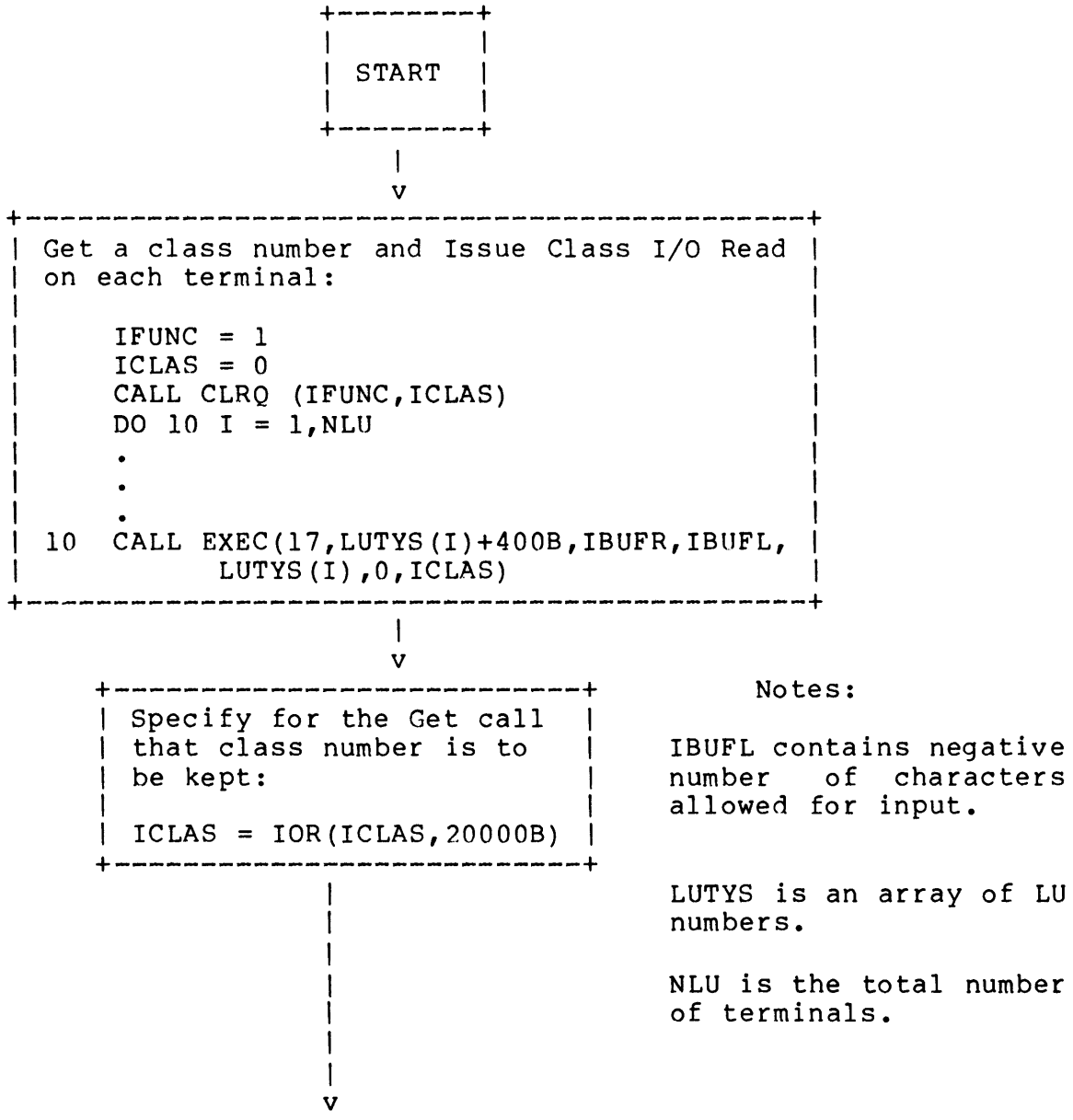
C     CHECK ERROR
      GOTO 100
       :
       :
  100 CONTINUE
C     ERROR PROCESSING.................................
       :
      END
```

EXAMPLE 2: MULTIPLE TERMINALS WITH A SINGLE APPLICATIONS PROGRAM

In the following example, any one of many users could be providing input to the program:



Assume an order-entry situation in which there are several operators but only one program. If standard I/O was used, it would be possible to read from only one terminal at a time. However, by using Class I/O, the program permits all operators to enter data seemingly at once. RTE handles all queueing so that the program operates on a single transaction at a time, thus simplifying the programming while giving the appearance of simultaneous processing on all transactions. The flowchart for such an application is illustrated in Figure 5-1. Note that although operators and terminal devices are shown, the input could be received from any one of a series of identical devices.



<continued on next page>

Figure 5-1. Class I/O Multiple Terminal Input Example

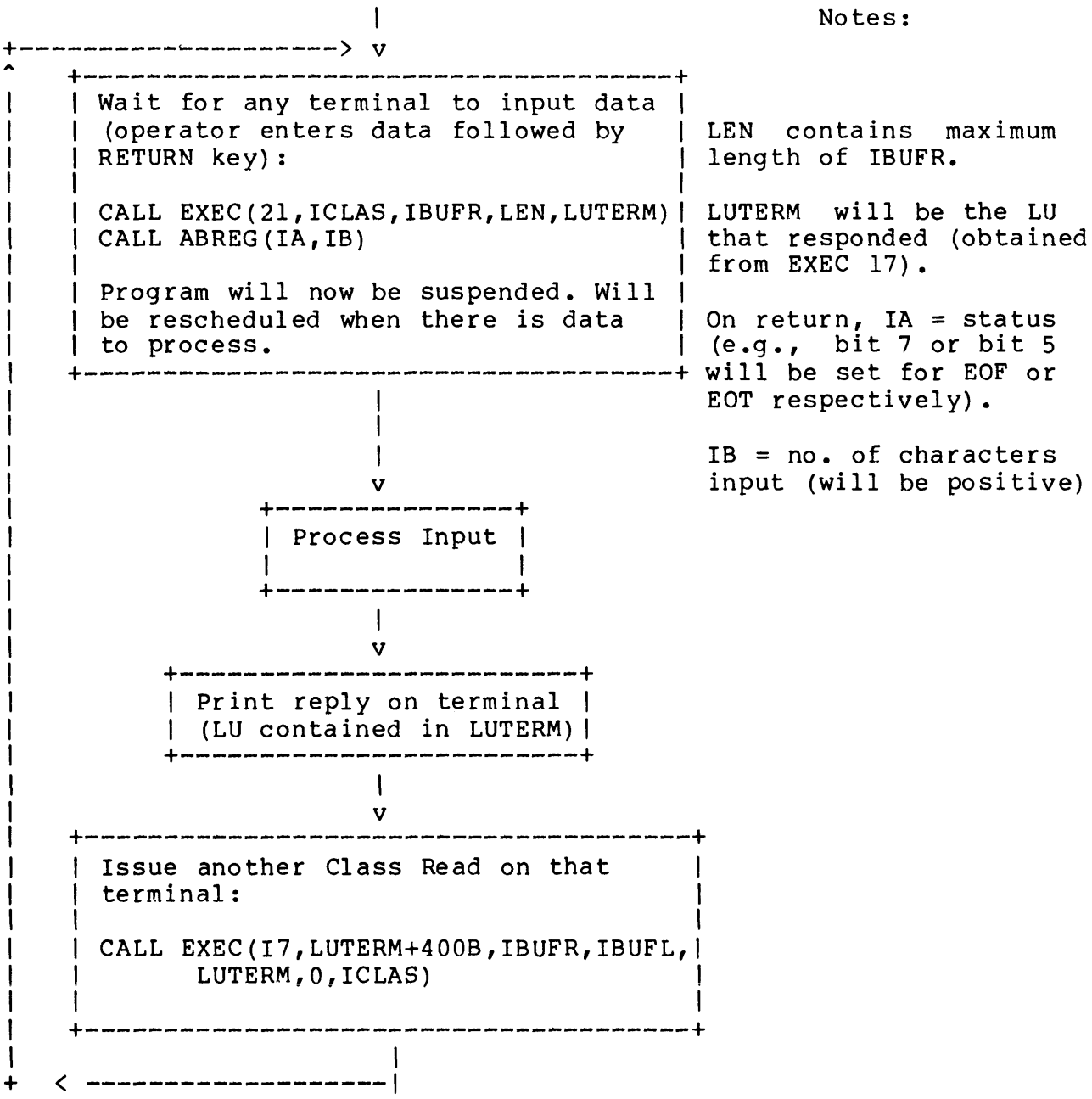


Figure 5-1. Class I/O Multiple Terminal Input Example

In some applications, it may be necessary to maintain contextual information for each operator; for example, a code indicating the type of input expected next, or the operator's name to be used in friendly dialog, etc. This information can be kept in a two-dimensional array that is indexed by the terminal LU number.

EXAMPLE 3: MAILBOX COMMUNICATION BETWEEN PROGRAMS

Program-to-program communication involves a "mailbox" scheme to pass data buffers back and forth in the most expeditious manner. Instead of implementing one large program to process all user inputs, it is often more efficient to separate these into subtasks that are processed by separate programs. In the example below, the program given in the previous example is still used as the "main control," but it now sends user inputs to the appropriate processor by using mailbox I/O. This separation allows the the various processors to be given different priorities, with the highest priority being assigned to those items that are most urgently needed. An added benefit is that the separation reduces the partition size requirements.

Assume that the box labeled "Process Input" in Figure 5-1 actually involved several programs, one each for a number of general categories:

a. Order entry.

b. Inventory quantity look-up.

c. Report generation.

d. Display of status or recent history of several critical real-time activities.

The program illustrated in Figure 5-1 might then serve only as a keyboard entry controller that checks input for legality and calls on other programs to process operator commands. Many operators could now enter commands, with the applications software relying on RTE to queue the commands according to the priority of the category.

The real-time display program might have the highest priority, perhaps followed by order entry, inventory quantity look-up, and report generation last.

Other orderings are possible, depending upon the application. Some management summary reports might be considered most important, or categories may be ordered so that those involving the least processing may have the highest priority to minimize waiting time for users with "short jobs."

The significant point to note is that RTE's priority-driven scheduling functions can be used to process commands according to priority. This is done through the simple means of separating the processes of those commands into separate programs that run at different priority levels, and coordinating the processing via Class I/O.

Figure 5-2 provides a revised version of the sample program given in Figure 5-1. In this new version, class numbers must be allocated for each of the process subprograms using the library routine CLRQ, and these subprograms must be scheduled. This is performed in the initialization section of the original program as follows:

```
      DO 20 I=1,NSUBP
      JCLAS=0
      IFUNC=1
      IOP1=<processing program name>
      CALL CLRQ (IFUNC,JCLAS,IOP1)
      CALL EXEC(18,0,IBUFR,0,0,0,JCLAS)
      JCLAS=IOR (JCLAS,20000B)
      CALL EXEC(21,JCLAS,IBUFR,0)
      CALL EXEC(10,<processing program name>,JCLAS)
  20  ISUBCL(I)=JCLAS
```

NOTES:

Every Class I/O Write, Read, Write/Read and Control call issued must always be matched with a corresponding Class Get call issued at some point in the calling sequence. The time sequence is not important (Class Gets can be issued before class calls) but there must be a Class Get for every class call. Failure to do so will tie up system resources (the class number and the system buffer memory) that other programs may need.

When a program is finished with a class number, it should explicitly release it with a Class Get call in which class number bits 13 and 14 are cleared and bit 15 is set. The system does not release a class number when the allocating program terminates unless the class number is assigned using the CLRQ library routine.

The "Process Input" box previously illustrated in Figure 5-1 can then be expanded as illustrated in Figure 5-2 below:

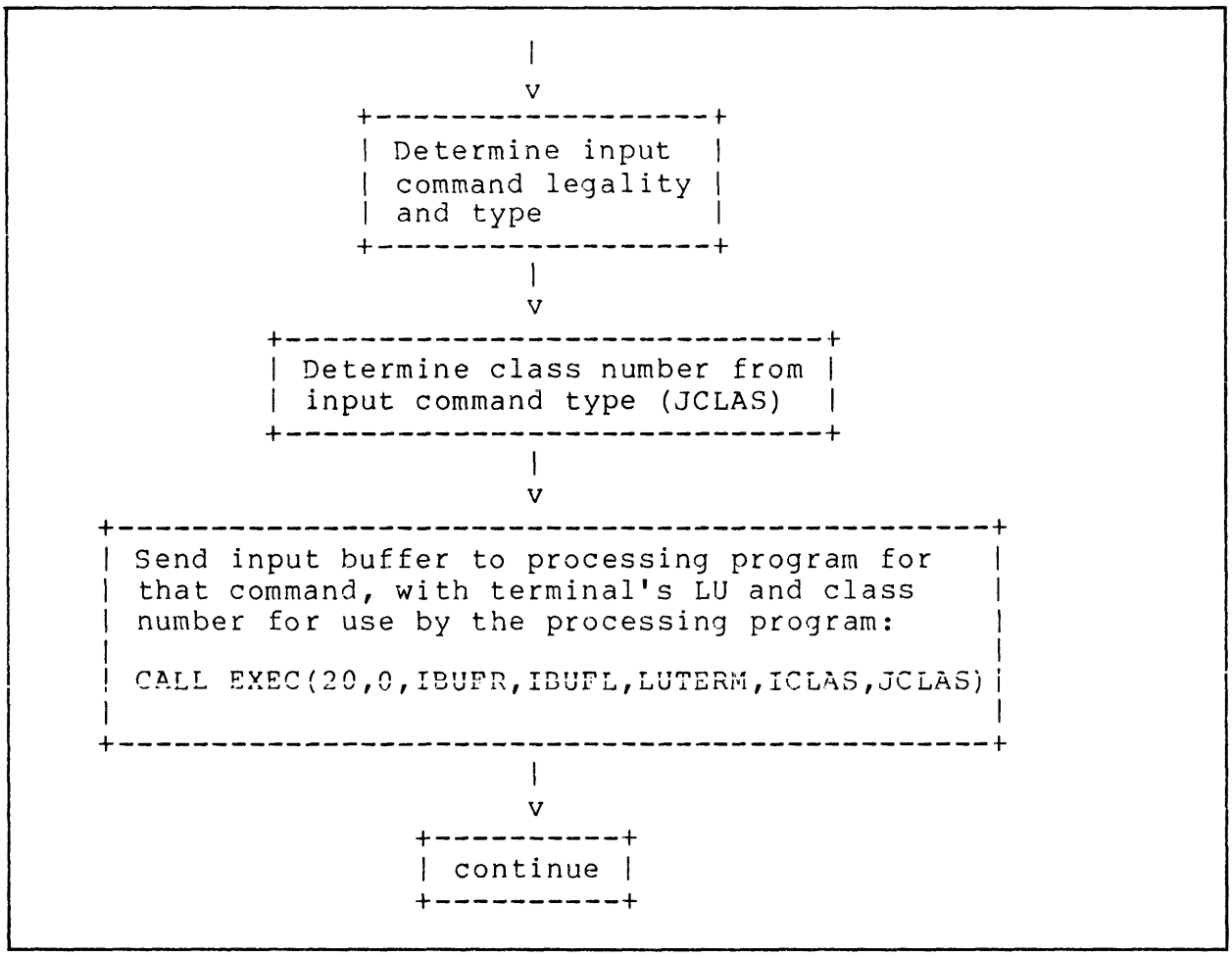


Figure 5-2. Dispatching Input to Subtasks for Processing

Because no devices are involved in mailbox I/O, the ICNWD (second parameter) of the request is zero. For this case, it is usually desirable to let the processing program print an acknowledgement or error return and then issue another Class Read on the terminal. The class number to use for this purpose is placed in the second optional parameter. For this reason, in the original example (Figure 5-1), the last two boxes for "printing a reply" and "issuing another Class Read" are deleted and included in the processing programs.

The processing programs obtain the class number to use for the above procedure by calling the RMPAR subroutine, as follows:

```
      CALL RMPAR(IPRAM(1))
      MYCLAS = IPRAM(1)
           .
           .
           .
      (Initialization code may go here)
      Waits for processing input
           .
           .
           .
  100 CALL EXEC(21,MYCLAS,IBUFR,MAXLEN,LUTERM,ITRMCL)
           .
      (Process input)
           .
      WRITE(LUTERM,1100)
 1100 FORMAT (<acknowledgement or error message>)
           .
           .
           .
  200 CALL EXEC(17,LUTERM+400B,IBUF,IBUFL,0,ITRMCL)
      GO TO 100
```

The processing programs now issue another Class Read to ITRMCL and return to line 100 to reissue the Class Get on MYCLAS, suspending themselves until other transactions are available for the programs to process.

Resource Numbers and Logical Unit Lock Applications

Like class numbers, the number of Resource Numbers available to the on-site RTE system is determined during system generation. Resource numbers provide the capability of synchronizing programs that access the same resource. The resource might be a device (locking a LU requires a resource number), a table in memory, a file or even another program or subroutine.

The use of resource numbers is only required when:

a. Two or more programs use the same device, or change the contents of a memory location of disc file.

b. One or more programs make decisions based upon the contents of a data item that can be modified by at least one other program.

To relate the resource number mechanism to applications considerations, assume the following "problem" conditions.

```
PROGRAM A                       PROGRAM B
COMMON J                        COMMON J
IF (J.EQ.2) J=J+1               IF (J.EQ.2) J=J+3
      :                               :
```

Assume that Programs A and B are both scheduled memory-resident programs and that J, which they share through system COMMON, is initially 2. Further assume that Program A executes the IF statement but before it can execute J=J+1, Program B gets scheduled (with B having the highest priority).

Program B sets J to J+3 (making it 5), perhaps performing other tasks, and then terminates.

Program A then increments J, making it 6. Notice that Program A running alone would leave J=3. Program B running alone would leave J=5. Programs A and B running together might leave J=3, 5 or 6.

Now assume J is a table of tasks to be executed and there are several programs scanning the table. Also assume the tasks are sufficiently I/O bound so that the applications software has several identical programs, each of which may select any task. Without synchronization via resource numbers, two or more of these programs might select the same task.

Such "race conditions" can be defined as any code that will execute unexpectedly, depending upon when other programs execute relative to the code. These conditions are an elusive form of software bug, causing unusual errors that can seldom be successfully repeated. Consequently, these errors are much harder to locate and identify.

Standard program priority cannot be relied upon to solve the described problems. Under the dynamics of real-time applications, there are too many other considerations under which a lower priority program occasionally may run when a higher priority program is scheduled.

A high priority program may have to be swapped because a still higher priority disc-resident program has been scheduled, and it either has been assigned to the same partition, or the partition is the smallest that the highest priority program will fit into. Meanwhile, the lower priority program may be running in another parition while the other programs are being swapped.

The proper way to avoid race conditions is to assign a resource number to all data accesses that are updated by more than one program, or updated by one program and read by others. However, it is extremely important to note and remember three items:

1. The association between a resource number (RN) and a shared data area is created through the user's software design. RTE's only role is to make RNs available for allocation, locking, clearing and releasing, and the system will suspend (state 3) any program that attempts to lock an RN that is already locked. RTE will reschedule the program only when the RN is cleared.

2. All programs that access the same resource must cooperate with each other in controlling "simultaneous" access. An RN may be saved anywhere that the cooperating programs can find it (SSGA and COMMON are typical) and the cooperating programs must lock the RN locally before accessing the associated resource and clear the RN when finished with it.

3. RTE automatically clears all RNs locked locally whenever the locking program is aborted or terminates (unless it terminates saving resources).

Resource numbers are managed by the RNRQ library subroutine. LUs can be locked by the LURQ library subroutines. The calling sequences for the subroutines are described below, followed by application examples.

RNRQ — Resource Management

The subroutine RNRQ allows cooperating programs a method of efficiently utilizing resources through a resource numbering scheme. A detailed discussion of resource management considerations is provided in this section.

```
CALL RNRQ (ICON, IRN, ISTAT)
           -----
```

ICON - Control option. Defines how the resource number is to be used (refer to COMMENTS below).

IRN - Resource number.

ISTAT - Status word. The status word returns are as follows:

- 0 - normal deallocate return
- 1 - RN is clear (unlocked)
- 2 - RN is locked locally to caller
- 3 - RN is locked globally
- 4 - no RN available now
- 6 - RN locked locally to other program
- 7 - RN was locked globally when request was made

COMMENTS:

Figure 5-3 illustrates the format of the control word required in the calling sequence.

15	14	5	4	3	2	1	0
WAIT OPTION		ALLOCATE OPTION			SET OPTION		
NO WAIT	NO ABORT	CLEAR	GLOBAL	LOCAL	CLEAR	GLOBAL	LOCAL

Figure 5-3. Control Word Format (ICON)

If more than one bit is set in the control word, the following order of execution is used:

1. local allocate (skip step 2 if done)
2. global allocate
3. deallocate (exit if done)
4. local set (skip step 5 if done)
5. global set
6. clear

The system has a set quantity of resource numbers (RNs) that are specified during generation. If a resource number is not available when a program requests one, the program is suspended (state 3) until one is free, unless the no wait bit is set. If the no wait bit is set, the RN location is set to zero. If the RN allocation is successful, the value returned in IRN is set by the system. It has no meaning to the user but must be specified (through IRN) when a lock is requested or the RN is cleared or deallocated.

The no abort bit is used to alter the error return point of the call as shown in the following example:

```
                                .
                                .
                          ICON = ICON + 40000B
                          CALL RNRQ (ICON,...)
error return        -----> GO TO 100
normal return point ------>     .
                                .
                                .
                          100 error processing
                                .
```

The above special error return is established by setting bit 14 to 1 in the request code word (ICON). This causes the system to execute the GO TO statement following CALL RNRQ if there is an error, or skip the GO TO statement if there is no error. If the error return is taken, error information will be available in the A- and B-Registers. If the alternate return label is used, the FORTRAN directive, $ALIAS RNRQ, NOABORT, must appear before the program statement. (Refer to the appropriate FORTRAN Reference Manual.)

```
                          $ALIAS RNRQ, NOABORT
                                .
                                .
                                .
                          ICON = ICON + 40000B
                          CALL RNRQ (ICON,...*100)
normal return point ------>
                                .
                                .
                                .
error return        ------> 100 error processing
```

RNRQ ALLOCATE OPTIONS:

LOCAL - Allocate an RN to the calling program. The number is returned in the IRN parameter. The number is automatically released on termination of the calling program, and only the calling program can deallocate the number.

GLOBAL - Allocate an RN globally. The number is released by a request from any program.

CLEAR - Deallocate the specified number.

RNRQ SET OPTIONS:

LOCAL - Lock the specified RN to the calling program. The RN is specified in the IRN parameter. The local lock is automatically released on termination of the calling program. Only the calling program can clear the number.

GLOBAL - Lock the specified RN globally. The RN is specified in the IRN parameter and the calling program can globally lock this number more than once. The number is released by a request from any program.

CLEAR - Unlock the specified RN.

If the RN is already locked to someone else, the operating system suspends the calling program (unless the no wait bit is set) until it clears the RN. If more than one program attempts to lock an RN, the program with the highest priority is given precedence. A single call can both lock and clear an RN.

If a program makes this call with the clear bit set, along with either the global or local bits set, the program will wait (in the general wait list) until the RN is cleared by another program and then continue with the RN clear.

An entry point is provided for drivers or privileged subroutines of Type 3 programs that clear a global (and only global) RN:

```
LDA     RN
JSB     $CGRN
return  point
```

LURQ — Logical Unit Lock

The subroutine LURQ allows a program to exclusively dominate (lock) an input/output device.

```
CALL LURQ (ICON, LURAY, NUM)
```

ICON - Control option. An octal number that specifies the locking or unlocking action to be performed (refer to the COMMENTS below).

LURAY - LU array. An array of LU numbers to be locked or unlocked.

NUM - The number of LUs to be locked or unlocked.

COMMENTS:

This request temporarily assigns a logical unit to the calling program. It prevents a higher priority program from interrupting a program's use of the device until the device is unlocked by the program that locked it.

The LURQ routine request allows up to 31 programs to exclusively dominate (lock) input/output devices. Any other program attempting to use or lock a locked LU will be suspended (state 3) until the original program unlocks the LU or terminates.

The functions of the control option (ICON) are summarized below:

* ICON = 000000B - unlock LUs specified in LURAY.

* ICON = 100000B - unlock all LUs the program currently has locked.

* ICON = 000001B - lock (with wait) the specified LUs.

* ICON = 100001B - lock (without wait) the specified LUs.

NO ABORT BIT (bit 14) - The no-abort bit is used to alter the error return point of this call as shown in the following example:

```
                                    .
                                ICON = ICON + 40000B
                                CALL LURQ (ICON,..)
error return            -----> GO TO 100
normal return point     ------>    .
                                   .
                                   .
                                100 error processing
                                   .
```

Setting bit 14 in ICON (ICON = ICON + 40000B) establishes the above special error return. This causes the system to execute the GO TO statement following the CALL LURQ if there is an error, or to skip the GO TO statement if there is no error. If the alternate return label is used, the FORTRAN directive, $ALIAS, LURQ, NOABORT, must appear before the program statement. (Refer to the appropriate FORTRAN Reference Manual.) An error will occur if the specified LU is not in the user's SST, or if the specified control word is illegal. If the error return is taken, error information will be available in the A- and B-Register.

8-BIT LU BIT (bit 13) - The 8-bit LU bit is used to lock an LU greater than 63. If set to 0, then the LU is between 0 and 63. If set to 1, then the LU number is greater than 63.

NO LU SWITCH BIT (bit 12) - The no LU switch bit indicates whether the specified LU is a system or session LU. Setting this bit indicates a system LU.

DISC ALSO BIT (bit 11) - The disc also bit is used to allow LU locks on discs. Failure to set this bit when specifyng a disc LU lock requests results in the LU error (LU02).

UNLOCK - To unlock all owned LUs, the LURAY array is not used but still must be coded; the program will not abort.

Any LUs the program has locked will be unlocked when the program:

* Performs a standard termination.

* Performs a serial reusability termination.

* Aborts.

Note that LUs will not be unlocked when the program performs a `save resources' termination.

This subroutine calls the program management subroutine (RNRQ) for a resource number (RN) allocation; that is, the system locks an RN locally to the calling program. Therefore, before the LURQ subroutine can be used, a resource number must have been defined during generation. Only the first 31 RNs can be used for LU locks.

If the no-wait option is coded, the A-Register will contain the following information on return:

0 - LU lock successful

-1 - no RN available at this time

1 - one or more of the LUs is already locked to some other program.

Note that the calling program may not have LUs locked at the time of the call unless the no-wait option is used. All LUs locked by the calling program are locked to the same RN.

A user program cannot put an LU lock on the system console. A call to LURQ(1,1,1) will not lock the system console or return an error.

EXAMPLE 1. TWO PROGRAMS UPDATING A DISC FILE

In this example the file may be either an FMP file or an area in the system track pool on LU 2 or LU 3. In the first case, the file must be opened non-exclusively (shared). Note that FMP files are normally opened for exclusive use and therefore are not shareable. No RNs are necessary to control exclusively opened files. In the second case, the disc tracks must be allocated globally. In either case, the RN must be kept in some area common to all programs (COMMON, SSGA, or in the file itself).

It is poor practice to assume the RNs will always be allocated in the same order; changes in initialization sequences or different RTE generations may change the RNs allocated. When RTE is booted up, an initialization program should be run automatically that will allocate all required RNs and store them where required.

You might possibly choose to use one RN to control access to all data bases. Although this practice consumes the least nunber of RNs, it is inefficient when several programs need to update different files.

Increasing the number of RNs so that each controls a smaller number of files or area of memory, increases the probability that the RN will be clear when the associated resource is required. More RNs therefore reduce the probability of a delay. The number of RNs allowed is limited to 255.

The application itself may limit the minimum area of control, depending upon the circumstances. Typically, one RN per file is the limit. However, one RN should control the set if several files are updated together.

Figures 5-4 and 5-5 illustrate the use of resource numbers when two programs want to update the same file.

```
      PROGRAM RNPR1
      DIMENSION INAM(3),IDCB(144),IBUF(128)
      DATA INAM/6HSHFILE/
C
C GLOBALLY ALLOCATE AND LOCALLY LOCK THE RN
C
      ICON=21B+40000B
      CALL RNRQ(ICON,IRN,ISTAT)
      GO TO 900
C
C SCHEDULE RNPR2 PASSING THE RN
C
   10 ICODE=10
      CALL EXEC(ICODE,6HRNPR2 ,IRN)
C
C OPEN THE FILE NONEXCLUSIVE
C
      IOPTN=1
      CALL OPEN(IDCB,IERR,INAM,IOPTN)
      IF(IERR.LT.0) GO TO 900
C
C UPDATE THE FILE
C
      :
      :
C
C CLOSE THE FILE
C
      CALL CLOSE(IDCB,IERR)
      IF(IERR.LT.0) GO TO 900
      :
  900 ERROR PROCESSING
      :
```

Figure 5-4. Resource Number Example 1

```
      PROGRAM RNPR2
      DIMENSION INAM(3),IDCB(144),IBUF(100),IP(5)
      DATA INAM/6HSHFILE/
C
C PICK UP THE RN
C
      CALL RMPAR(IP)
      IRN=IP(1)
C
C ATTEMPT TO LOCK THE RN
C
      ICON=1
      CALL RNRQ(ICON,IRN,ISTAT)
C
C OPEN THE FILE NONEXCLUSIVE FOR UPDATE
C
      IOPTN=1
      CALL OPEN(IDCB,IERR,INAM,IOPTN)
      IF(IERR.LT.0) GO TO 900
C
C UPDATE THE FILE
C
      :
      :
C
C CLOSE THE FILE
C
      CALL CLOSE(IDCB,IERR)
      IF(IERR.LT.0) GO TO 900
C
C DEALLOCATE THE RN
C
      ICON=40B
      CALL RNRQ(ICON,IRN,ISTAT)
      :
  900 ERROR PROCESSING
      :
```

Figure 5-5. Resource Number Example 2

EXAMPLE 2. DEVICE CONTROL

Programs using a device that many other programs also use (e.g., line printer) should usually lock it first. The File Management System provides users with this exclusive control and therefore LU locking is not required. Whenever any other program attempts to access the LU, the calling program will be suspended (state 3) until the locking program unlocks the LU, terminates, or aborts. Note that in this case, cooperation among programs is not required because RTE performs the LU/RN association.

However, when two or more programs employ LU locking, a condition known as "Deadly Embrace" can sometimes occur.

The "Deadly Embrace" condition can occur when programs attempt to lock more than one resource in separate calls. For example, assume the following situation:

a. Programs PROGA and PROGB are running. PROGA locks the line printer and then begins to output to it, causing PROGA to be I/O suspended (state 2).

b. PROGB runs, locks the magnetic tape unit and outputs to it, causing PROGB to be I/O suspended (state 2).

c. Now assume that PROGA is rescheduled and attempts to use or lock the magnetic tape unit. Because it is already locked by PROGB, PROGA is suspended (state 3).

d. If PROGB attempts to use or lock the line printer, then it also will be suspended (state 3).

e. PROGA and PROGB each now require a resource the other currently "owns"; neither can proceed and will stay "locked-up" together forever unless an operator intervenes.

Figures 5-6 through 5-8 illustrate a typical "Deadly Embrace" condition. Programs LOCKA and LOCKB use the same shareable EMA partition. Program LOCKA allocates and locks LOCK1, and then waits one minute while the operator schedules the LOCKB program. Program LOCKB allocates and locks LOCK2 and then waits one minute.

When program LOCKA runs again, it attempts to lock LOCK2 and is suspended (state 3). Program LOCKB attempts to lock LOCK1 and is also suspended (state 3).

```
$EMA(LOCK)
      PROGRAM LOCKA
      COMMON /LOCK/LOCK1,LOCK2
C
C  Allocate first resource number  (IRN1)
C
      ICODE=11B
      CALL RNRQ(ICODE,IRN1,ISTAT)
      LOCK1=IRN1
      WRITE(1,'("LOCKA:STATUS LOCK #1",I5)')ISTAT
      CALL EXEC(12,0,3,0,-1)     !Time schedule LOCKA
C
C  Allocate second resource number (IRN2)
C
      ICODE2=1
      IRN2=LOCK2
      CALL RNRQ(ICODE2,IRN2,ISTAT)
      WRITE(1,'("LOCKA:STATUS LOCK #2 ",I5)')ISTAT
      END
```

Figure 5-6. "Deadly Embrace" Example 1

```
$EMA(LOCK)
      PROGRAM LOCKB(3,90)
      COMMON/LOCK/ LOCK1,LOCK2
C
C  Allocate the first resource number (IRN2)
C
      ICODE=11B
      CALL RNRQ(ICODE,IRN2,ISTAT)
      LOCK2=IRN2
      WRITE(1,'("LOCKB:STATUS LOCK #1 =",I5)')ISTAT
      CALL EXEC(12,0,3,0,-1)   !Time schedule LOCKB
C
C  Allocate the second resource number (IRN1)
C
      ICODE2=1
      IRN1=LOCK1
      CALL RNRQ(ICODE2,IRN1,ISTAT)
      WRITE(1,'("LOCKB:STATUS LOCK #2 =",I5)')ISTAT
      END
```

Figure 5-7. "Deadly Embrace" Example 2

```
12:52:29:370
-------------------------------------------------------------------
 PRGRM T PRIOR PT  SZ DO.SC.IO.WT.ME.DS.OP.  .PRG CNTR.  .NEXT TIME.
-------------------------------------------------------------------
.
  WHZAT 1 00041  0  . .  1, . . . . . . . . . . . . P:72543
  LOCKA 3E00099 16   5 . . . .  3,RN  056,LKPRG=LOCKB
                                                     P:36263
** BLOCK **
  LOCKB 3E00090 18   5 . . . .  3,RN  057,LKPRG=LOCKA
                                                     P:36264
** BLOCK **
********** DEAD LOCK **
*** SEE ABOVE FOR REPORT ON LOCKA
-------------------------------------------------------------------
ALL LU'S OK
ALL EQT'S OK
-------------------------------------------------------------------
12:52:30:680
```

Figure 5-8. "Deadly Embrace" Example 3

In the case of devices, this condition can be avoided by locking all devices that may be required at once in the same call. The program will be suspended until all devices are available.

In the case of resource numbers, the condition can generally be avoided by increasing the "area of control" of the resource numbers so that a program requiring simultaneous and exclusive access to two files (for instance) merely locks on RN, rather than one for each file. If an applications problem does not allow this solution, then the user should attempt to lock all RNs required without suspension (bit 15 of ICODE is set).

If a lock cannot be granted, you may wish to attempt the following steps:

1. Do not update any of the related files; post whatever has already been processed (only for those files to which exclusive access has already been obtained).

2. Release all RNs that are locked and re-attempt to lock the last RN, this time with suspension.

3. When the lock is granted, re-lock all the previous RNs and continue. Note that RTE will allow a program to locally lock an RN that it has already locked locally.

In summary, if a program must lock more than one resource and finds one or more of these resources already in use, the program should "back off", release all RNs it has already locked (if any), wait for the resource it wanted to become available, and then re-attempt to lock all RNs it needs. The program must not fully or partially update any files unless it has all the RNs it needs (that control access to the file and any related files that must be updated) locked simultaneously.

Parameter Passage Applications

Values can be passed to a program when you execute it via the RU command or schedule it with an EXEC call. Up to five integer values (or pairs of ASCII characters) may be included as parameters in the RU command or the scheduling EXEC call.

The program can then use the library subroutine RMPAR to retrieve these values and store them in an array in the program. Parameters can be returned to FMGR or the father program by making a call to the library routine PRTN or PRTM.

A string of characters can also be passed to the program by the RU command. The library routine GETST can be called to retrieve the parameter string (any characters after the second comma).

The calling sequence of RMPAR, PRTN, PRTM, and GETST subroutines are described in this section.

RMPAR - Retrieve Parameters

The subroutine RMPAR is used to retrieve up to five parameters passed to a program by the operating system.

CALL RMPAR (IPARM)
IPARM - Parameter buffer. A 5-word integer array that contains up to 5 parameters to be passed to the program.

COMMENTS:

RMPAR is a general purpose request used to recover parameters passed to the calling program. These parameters can originate from a number of sources: operator commands (RU, XQ, ON, GO), program scheduling requests (EXEC 6, 7, 9, 10, 23, 24), or information passed by some drivers after an unbuffered read, write, or control (EXEC 1, 2, 3) request. For recovery of scheduling parameters, a call to RMPAR should be the first executable instruction following a program suspend call. For example:

```
PROGRAM PASS
DIMENSION IPARM(5)
CALL RMPAR (IPARM)                    !PICK UP PARAMETERS
 .
 .
 .
CALL EXEC(7)
CALL RMPAR (IPARM)
```

In most cases, the system attempts to default the first parameter to the LU of an interactive terminal to which the calling program can report errors. This default requires the calling program to use LU 1 if the first parameter is 0; otherwise, the actual first parameter is used.

If at least one but less than five parameters are passed from the operator runstring or from the father scheduling program and a RMPAR request is made, the unsupplied parameter positions return zero.

If no parameters are passed, the first parameter defaults to 0 in the following cases:

- RU or ON entered from a non-session prompt at the system console

- EXEC 9, 10, 23, or 24 calls

- All GO commands

- MESSS calls that do not have the LU parameter specified

- MESSS calls, or RU or ON entered after the S=xx COMMAND? prompt and preceded by at least one space.

If no parameters are passed, the first parameter defaults to the LU of the terminal from which the calling program was run in the following cases:

- RU or ON entered after a S=xx COMMAND? prompt and not preceded by any spaces

- RU or XQ entered from CI or FMGR

If the first parameter is 0, the terminal LU is passed in the following cases:

- ON or RU entered after a S=xx COMMAND?

- MESSS calls with the LU specified

In all other cases, 0 is passed when the value of the first parameter is 0.

For example:

```
CI> XQ,PROGA,1,,2   or   CALL EXEC(9, PROGA, 1, 0, 2)
```

causes PROGA to become scheduled with 3 parameters. When PROGA executes the request:

```
INTEGER ARRAY(5)
CALL RMPAR(ARRAY)
```

```
Then:    ARRAY(1)=1
         ARRAY(2)=0
         ARRAY(3)=2
         ARRAY(4)=0
         ARRAY(5)=0
```

ASCII strings passed by the XQ, RU, ON, and GO commands (or a MESSS call) are parsed according to the rules documented with the PARSE subroutine. Only the first two characters of an ASCII parameter are passed by RMPAR.

A- and B-Registers are meaningless upon return from RMPAR.

PRTN, PRTM — Parameter Return

PRTN or PRTM routine is used by the calling program to pass parameters back to its father (the program that scheduled it) or FMGR. The father can recover the returned parameters by calling the RMPAR routine.

```
CALL PRTN (IPRAM)
    or
CALL PRTM (IPRAM)
```

IPRAM - Parameter buffer. A 5-word array (for PRTN) or a 4-word array (for PRTM) that contains the parameters to be returned to the father program.

COMMENTS:

The PRTN routine passes five parameters and clears the wait flag. Because the wait flag is cleared, the calling program should terminate immediately after the call. For example;

```
DIMENSION IPRAM (5)
  :
CALL PRTN (IPRAM)
CALL EXEC (6)
```

The PRTM routine passes four parameters and does not clear the wait flag; an immediate termination (EXEC 6) is not necessary. When the parameters are recovered with RMPAR, the first parameter will be meaningless.

When FMGR is the father, the returned parameters are in the global parameters 1P - 5P.

The Macro/1000 calling sequences for the PRTN and PRTM routine are shown below:

```
        EXT  EXEC, PRTN
         :
        place values in IPRAM
         .
        JSB  PRTN
        DEF  *+2
        DEF  IPRAM
        JSB  EXEC
        DEF  *+2
        DEF  SIX
         :
IPRAM   BSS  5
SIX     DEC  6
```

```
        EXT  PRTM
         :
        place values in IPRAM
         .
        JSB  PRTM
        DEF  *+2
        DEF  IPRAM
         :
IPRAM   BSS  4
```

GETST — Recover Parameter String

The GETST subroutine recovers the parameter string from a program's command string storage area. The parameter string is defined as all the characters following the second comma in the command string, the third comma if the first two characters in the first parameter are NO (as in ON, PROG, NOW, PARM1, PARM2), or the first comma if the implied RUN is used. The parameter string is returned with all leading and trailing blanks removed.

```
CALL GETST (IBUF, ILEN, ILOG)
            ----  ----
```

IBUF - The user defined buffer large enough to hold the parameter string.

ILEN - The length of IBUF, expressed as a positive number of words or a negative number of characters. Recommended length is a maximum of 40 words, or 80 characters.

ILOG - Transmission log or actual number of words or characters returned (always positive). If ILEN is positive, ILOG contains the number of words in the string; if ILEN is negative, ILOG is equal to the number of characters in the string.

COMMENTS:

The Macro/1000 calling sequence for GETST is as follows:

```
        EXT  GETST
         :
        JSB  GETST
        DEF  RTN
        DEF  IBUF
        DEF  ILEN
        DEF  ILOG
RTN      :
         :
IBUF    BSS  n
ILEN    DEC  n
ILOG    NOP
         :
```

Upon return, ILOG contains a positive integer giving the number of words (or characters) actually transmitted into IBUF. ILOG will never return greater than ILEN. The A- and B-Registers are meaningless after a return from GETST. Note that if RMPAR is used, it must be called before GETST.

When an odd number of characters is specified, an extra space is transmitted in the right half of the last word.

This subroutine performs a function similar to an EXEC 14 call.

NOTE

> Any string that is retrieved with GETST must be structured so that two leading commas exist in the string. GETST discards the information preceding the two commas and returns the string following them.

The program pair and its corresponding output show the difference between EXEC 14 and GETST.

EXAMPLE 1:

This sample program demonstrates an EXEC 14 call.

```
        PROGRAM EXAMP
        IMPLICIT INTEGER (A-Z)
        DIMENSION BUFR1 (40)

C       GET THE RUNSTRING

        CALL EXEC (14,1,BUFR1,40)
        CALL ABREG (A,LEN)

        WRITE (1,1) (BUFR1(I),I=1,LEN)
1       FORMAT ("The runstring is:  ", 40A2)
        END
```

OUTPUT:

```
The runstring is: RU,EXAMP,P,P1,P2,P3,P4,P5,P6.
```

EXAMPLE 2:

This sample program illustrates a GETST call.

```
       PROGRAM EXAMP
       IMPLICIT INTEGER (A-Z)
       DIMENSION BUFR2 (40)
C      GET THE PARAMETER STRING WITH GETST

       CALL GETST (BUFR2,40,ILOG)

       WRITE (1,2) (BUFR2(I),I=1,ILOG)
2      FORMAT ("Parameter part is:  ",40A2)
       END
```

OUTPUT:

```
Parameter part is: P1,P2,P3,P4,P5,P6.
```

I/O Requests and Related Routines

.STIO — Configures Driver

This routine is used to configure a driver for the select code currently in use.

Not callable in FORTRAN or Pascal
See Macro/1000 calling sequence below.

The Macro/1000 calling sequence is as follows:

```
      LDA SC        A=SELECT CODE (LOW 6 BITS)
      JSB .STIO     INVOKE SUBROUTINE
      DEF RTN
      DEF IO.1[,I] POINTERS TO LOCATIONS TO CONFIGURE
      DEF IO.2[,I]
         - - -
      DEF IO.N[,I]
      RTN < --- >   RETURN POINT
    UPON RETURN, A IS UNCHANGED, B IS ??
```

On return, the A-Register is unchanged, the B-Register content is unknown.

BINRY — Data Transfer

The BINRY subroutine is called from a FORTRAN program to transfer information to or from a disc device. BINRY has two entry points; BREAD for read operations and BWRIT for write operations.

```
CALL BWRIT (IBUFF, ILEN, IDISC, ITRAK, ISECT, IOFST)

CALL BREAD (IBUFF, ILEN, IDISC, ITRAK, ISECT, IOFST)
            -----
```

IBUFF - Data buffer. Contains data to be written for a write operation or data returned from a read operation (must be a non-EMA buffer).

ILEN - Data length. The number of words to be read or written.

IDISC - Disc LU. Logical unit number of disc device involved in the data transfer.

ITRAK - Disc track number.

ISECT - Disc sector number.

IOFST - Sector offset. Offset (in words) within the sector.

If IOFST = 0, the transfer starts at the sector boundary.

If IOFST = n, the transfer starts n words past the beginning of the sector.

COMMENTS:

Because data transfer between a user's program and a disc device are buffered through the driver module on a sector basis, certain sector offset considerations must be allowed. These sector offset considerations are summarized below:

1. Offset=n (transfer begins within a sector), and less than a sector is written, or the data transfer ends on a sector boundary. The entire first sector is initially read into the driver's internal buffer, the data is modified according to the BWRIT statement, and the entire sector is then rewritten on the disc with no data loss. No special precautions are required in this instance.

2. Offset=0 (transfer begins on a sector boundary), and less than a sector is written. The remaining data in the sector will be lost unless the entire existing sector on the disc is first read into a user's buffer, modified to reflect the desired changes, and then rewritten on the disc as a full sector.

3. Offset=0 or n, and a sector boundary is crossed in the data transfer. The remaining data in the final sector will be lost unless the entire final sector (of the data transfer) on the disc is read into a user's buffer, modified to reflect the desired changes, and then rewritten on the disc as a full sector.

It is the user's responsibility to initiate the "read-before-write" operation in case 2 and 3 above, the system automatically handles case 1.

The Macro/1000 calling sequence is as follows:

```
      EXT BWRIT (or BREAD)
       .
       .
      JSB BWRIT (or BREAD)
      DEF RTN
      DEF IBUF
      DEF ILEN
      DEF IDISC
      DEF ITRAK
      DEF ISECT
      DEF IOFST
RTN    .
       .
```

EQLU — Finds Logical Unit Number

The EQLU subroutine finds the logical unit number of an interrupting device if given the address of word 4 of the device's Equipment Table entry.

```
CALL EQLU (LU)
           --
```

LU - Logical unit number of interrupting device (same as A-Register return).

COMMENTS:

The EQLU routine expects the address of EQT word 4 of the interrupting device to be in the B-Register. This is done by another program/subroutine (using LDB EQT4) or by the driver associated with the interrupting device.

The EQLU routine will function correctly only if the LU number to be returned is less than or equal to 99.

The Macro/1000 calling sequence is as follows:

```
          EXT  EQLU
           :
          JSB  EQLU
          DEF  RTN
          DEF  LU
     RTN   :
```

On return:

A-Register = 0 if an LU referring to the EQT was not found.
= LU if LU was found.

B-Register = ASCII "00" (if LU not found) or the ASCII LU number.

LU = Same as A-Register.

IFTTY — Logical Unit Interactive?

The IFTTY subroutine determines whether a logical unit is interactive or not.

```
INT = IFTTY (LU)
---
```

INT - Set to -1 if LU is interactive; set to 0 if LU is not interactive (same as A-Register return).

LU - Logical unit number of device being tested.

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
          EXT  IFTTY
           .
           .
          JSB  IFTTY
          DEF  RTN
          DEF  LU
     RTN   .
           .
     LU   DEC  n      Logical unit being tested.
```

On return, the following registers are set as indicated:

A-Register = -1 if LU is interactive, 0 if it is not interactive.

B-Register = Upper byte is the driver type (word 5 of Equipment Table entry, bits 8-13). Lower byte is the subchannel number.

LOGLU - Logical Unit Return

The LOGLU subroutine returns the logical unit numbers of the terminal from which the currently executing program was scheduled.

```
LU = LOGLU (LUSYS)
--          -----
```

LU - Logical unit number of device from which the calling program was scheduled (same as A-Register); 1 (if in session) or positive log-LU (if not in session).

LUSYS - The system LU of the session terminal (in session) or the negative log-LU (not in session).

COMMENTS:

The LOGLU routine will return the LU numbers of the console from which the currently executing program was scheduled. This LU number is passed down from the Father program to the Son program when one program schedules another program for execution. If the program was scheduled by interrupt or from the time list, the scheduling LU will be LU 1, the system console.

The Macro/1000 calling sequence is as follows:

```
          EXT  LOGLU
           .
           .
          JSB  LOGLU
          DEF  RTN
          DEF  LUSYS
     RTN   .
           .
```

On return:

A-Register = LU number of device from which program was scheduled.

B-Register = ASCII LU number

LUTRU — Returns True System Logical Unit

The LUTRU subroutine returns the true system logical unit number associated with a session or batch LU.

```
CALL LUTRU (ITEST, LU)
                   --
   or

LU = LUTRU (ITEST)
--
```

LU — True system LU is returned here.

ITEST — The session logical unit number to be checked.

COMMENTS:

If the calling program is not a session or batch program, LU is set equal to ITEST.

If the calling program is a session program and ITEST is not defined for the caller's session, LU is set to -1.

If the calling program is a batch program and ITEST is not defined in the Batch Switch Table, LU is set equal to ITEST.

The Macro/1000 calling sequence is as follows:

```
      EXT LUTRU
       .
       .
      JSB LUTRU
      DEF RTN
      DEF ITEST
      DEF LU
RTN   .
      .
```

REIO, XREIO — Reentrant I/O

The REIO subroutine permits user programs to perform reentrant I/O and disc-resident programs to be swappable while performing I/O. REIO is a utility type library subroutine that is appended to each program that calls it (Type 7).

XREIO, extended REIO, allows access of up to 255 logical units through an extended (double word) ICNWD. All other parameters and functions of XREIO are identical to REIO. The format of ICNWD is the same as XLUEX, a subroutine documented in Chapter 2.

```
CALL REIO (ICODE, ICNWD, IBUFF, ILEN)
```

ICODE - Request code. 1 = read; 2 = write.

ICNWD - Control word. Specifies the LU (must be non-disc) involved in the I/O operation. Can also specify driver-dependent information.

IBUFF - Data buffer. Contains the data to be written in a write operation or data returned from a read operation.

ILEN - Data length. Positive number of words (or negative number of characters) to be read or written.

COMMENTS:

ICODE, IBUFF, and ILEN are identical to those used in the EXEC 1 and EXEC 2 calls. ICNWD is similiar except that the Z-bit is not used (no control buffer passage) and the LU cannot specify a disc device.

REIO will always perform the requested I/O operation. However, it will perform the operation on a reentrant basis only if the buffer is less than 130 words (to save System Available Memory), and the buffer address is at least three words beyond the beginning of the program. Note that FORTRAN puts equivalenced arrays at the beginning of the program. Therefore, a dummy array (of at least three words) should be equivalenced prior to equivalencing the buffer to be used in the REIO call.

Error processing and the returns for the A- and B-Registers are the same as for the EXEC 1 and 2 calls.

The Macro/1000 calling sequence is as follows:

```
      EXT REIO (or XREIO)
       .
       .
      JSB REIO (or XREIO)
      DEF RTN
      DEF ICODE
      DEF ICNWD
      DEF IBUFF
      DEF ILEN
RTN   .
      .
```

TRMLU - Finds Logical Unit Number

The TRMLU subroutine finds the logical unit number of an interrupting device if given the address of word 4 of its Equipment Table entry, and checks that it is an interactive device.

CALL TRMLU (LU) ---
LU - Logical unit number of interrupting device. (Same as A-Register return).

COMMENTS:

The TRMLU routine expects the address of EQT word 4 of the interrupting device to be in the B-Register. This is done by another program/subroutine (using LDB EQT4) or by the driver associated with the interrupting device.

The TRMLU routine will function correctly only if the LU number to be returned is less than or equal to 99.

The Macro/1000 calling sequence is as follows:

```
          EXT  TRMLU
           :
          JSB  TRMLU
          DEF  RTN
          DEF  LU
     RTN   :
```

On return:

A-Register = 0 if LU not found.
= LU number if found.

B-Register = ASCII "00" if LU not found, ASCII LU number if found.

LU = Same as A-Register. (optional).

Session Related Routines

GTERR — Returns SCB Error

The GTERR subroutine returns the SCB error mnemonic from the current Session Control Block (SCB).

```
CALL GTERR (INMIC [,IERR])
            -----   ----
```

INMIC - 4-word buffer that error mnemonic is returned to.

IERR - Error return. 0 indicates that the retrieval was successful; -1 indicates that the calling program was not in session (optional).

COMMENTS:

The error mnemonic is an 8-ASCII-character message that represents the last error encountered for the current session. The error mnemonic is posted by the RTE-6/VM Operating System and some HP supported subsystems. The error mnemonic can also be updated by a user application program by calling the library PTERR routine.

Note the HELP command (see RTE-6/VM Terminal User's Reference Manual) uses the 8-character mnemonic as the implicit keyword to search the HELP file and returns expanded information on the last error posted in the current session.

The error mnemonic is stored in words 5 through 8 of the SCB.

The Macro/1000 calling sequence is as follows:

```
      EXT GTERR
       .
       .
      JSB GTERR
      DEF RTN
      DEF INMIC
      DEF IERR    (optional)
RTN    .
       .
```

GTSCB — Returns Contents of Current SCB

The GTSCB subroutine returns the contents of the current SCB.

```
CALL GTSCB (IBUF,ILEN,IERR[,ADSCB])
            ----      ----
```

IBUF - Buffer to which the contents of the SCB are returned, beginning with word 3.

ILEN - Length of IBUF.

IERR - If positive, indicates the number of words in the SCB (not counting words 0, 1, and 2). If -1, indicates the calling program was not in session. If -2, indicates the SCB address passed to GTSCB was not a valid address. If a negative number (besides -1 or -2), indicates that the buffer size passed to GTSCB (IBUF) was too small; the negative number indicates the actual SCB size (not counting words 0, 1, and 2).

ADSCB - Address of SCB (optional).

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
      EXT GTSCB
       .
       .
      JSB GTSCB
      DEF RTN
      DEF IBUF
      DEF ILEN
      DEF IERR
      DEF ADSCB  (optional
RTN    .
       .
```

ICAPS — Returns Current Capability Level

The ICAPS subroutine returns the current session's capability level.

```
ICPSE = ICAPS (IDUMY)
```

IDUMY - Dummy variable.

ICPSE - Session capability; 0 if calling program was not in session.

COMMENTS:

The following FORTRAN program segment illustrates the ICAPS subroutine.

```
      PROGRAM SESCAP

      ICPSE=ICAPS (IDUMY)
      IF (ICPSE .EQ. 0) GO TO 99
C     ***USE SESSION CAPABILITY***
      .
      .
      .
      STOP
99    WRITE (PROGRAM NOT IN SESSION)
      END
```

The Macro/1000 calling sequence is as follows:

```
      EXT ICAPS
       .
       .
      JSB ICAPS
      DEF *+1
      - return -
       .
       .
```

LUSES — Scans Session Control Blocks

The LUSES subroutine scans the list of Session Control Blocks (SCBs) looking for a SCB defined for the session identifier passed in the call.

```
ISCB = LUSES (IDBNT)
----
```

ISCB - Address of SST length word in SCB if found, 0 if not found (same as A-Register return).

IDBNT - Session identifier (word 3 of SCB).

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
          EXT  LUSES
           .
           .
          JSB  LUSES
          DEF  RTN
          DEF  IDENT
     RTN   .
           .
```

On return:

A-Register = 0 if SCB not found.

= SCB address if found.

PTERR — Updates Error in Session Control Block

The PTERR subroutine updates the error mnemonic in the current Session Control Block (SCB).

```
CALL PTERR (INMIC [,IERR])
                  ----
```

INMIC - 4-word buffer that contains the error mnemonic to be posted to the SCB.

IERR - Error return. 0 indicates a successful posting operation; -1 indicates that the calling program was not in session (optional).

COMMENTS:

The PTERR routine places an 8-ASCII-character (4-words) message in the current SCB. This message can be retrieved by calling the library GTERR routine. The PTERR/GTERR calls can be used to implement error and data communication schemes between programs running in the current session.

The Macro/1000 calling sequence is as follows:

```
          EXT  PTERR
           .
           .
          JSB  PTERR
          DEF  RTN
          DEF  INMIC
          DEF  IERR  (optional)
     RTN   .
           .
```

SESSN — Calling Program in Session?

The SESSN subroutine determines if the calling program is in session.

CALL SESSN (ID)
ID -ID segment address of program that is to be checked.

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
                EXT  SESSN
                 .
                 .
                JSB  SESSN
                DEF  RTN
                DEF  ID
        RTN      .
                 .
```

On return from SESSN:

E-Register = 0 if calling program was in session.
1 if calling program was not in session.

B-Register = ID segment session word (address of SST length word in SCB) if calling program was in session.

The ID segment address of the calling program can be obtained by using the library IDGET routine.

System Calls and Entry Points

OPSYS — Returns $OPSY Value

This routine returns the $OPSY value for the currently executing operating system.

```
IRTE = OPSYS( )
```

IRTE - $OPSY value for the currently executing operating system.

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
      EXT OPSYS
       :
      JSB OPSYS
      DEF *+1
      - return -
       :
```

On return result in register A:

```
      JSB .OPSY
      A = -7   (RTE-MI)
      A = -15  (RTE-MII)
      A = -5   (RTE-MIII)
      A = -3   (RTE-II)
      A = -1   (RTE-III)
      A = -9   (RTE-IV, RTE-IVB)
      A = -17  (RTE-6/VM)
      A = -13  (RTE-4E)
      A = -29  (RTE-XL)
      A = -31  (RTE-L)
      A = -37  (RTE-A)
      A = -45  (RTE-A.1)
```

NOTES: This routine is equivalent to

```
        EXT $OPSY
        XLA $OPSY
```

SEGLD Overlay Load With Link

SEGLD loads an overlay of the calling program from disc into an overlay area in memory provided by the program, and transfers control to the overlay's entry point. It is an alternative to EXEC 8 which allows use of SEGRT and Symbolic Debug.

```
CALL SEGLD(NAME,IERR)
```

NAME - a three-word array containing the five-character overlay name:

- NAME(1) = 1st two characters
- NAME(2) = 2nd two characters
- NAME(3) = last character in upper 8 bits (the lower byte is not significant)

IERR - an error return; its value is 5 if the overlay cannot be loaded.

COMMENTS:

SEGLD loads overlays via an executive request. Control returns to the calling routine if the segment cannot be loaded (IERR will be equal to 5) or if the overlay calls SEGRT (IERR will equal zero).

SEGLD With Loader

A segmented program loaded with Loader has different characteristics than one loaded with Link. The SEGLD (loader) subroutine supports optional parameters; SEGLD (Link) does not. An explanation of SEGLD (loader) follows:

```
CALL SEGLD(NAME,IERR[,PRAM1[,PRAM2[,PRAM3[,PRAM4[,PRAM5]]]]])
```

NAME - a three-word array containing the five character segment name:

- NAME(1) = 1st two characters
- NAME(2) = 2nd two characters
- NAME(3) = last character in upper 8 bits (the lower byte is not significant)

IERR - an error return; its value is 5 if the segment cannot be loaded.

PRAM1 through PRAM5 - up to five optional parameters that the segment may recover via RMPAR.

COMMENTS:

SEGLD (loader) segments via an EXEC 8 request. Control returns to the calling routine if the segment cannot be loaded (IERR will be equal to 5) or if the segment calls SEGRT (IERR will equal zero). The advantage of SEGLD over EXEC 8 is that segments can return to the main program if SEGLD is used; EXEC 8 does not allow a return to the main program as its standard operation.

PRAM1 through PRAM5 can be passed to the segment to be loaded. The segment should use RMPAR to retrieve these parameters.

The Macro/1000 calling sequence is as follows:

```
        EXT SEGLD
         .
         .
        JSB SEGLD
        DEF RTN
        DEF NAME
        DEF IERR
        DEF PRAM1  (optional)
        DEF PRAM2  (optional)
        DEF PRAM3  (optional)
        DEF PRAM4  (optional)
        DEF PRAM5  (optional)
RTN      .
         .
```

SEGRT — Returns Instructions After SEGLD Request

SEGRT allows a segment to return to the instruction following the SEGLD call in the main program.

```
CALL SEGRT
```

COMMENTS:

SEGRT allows any segment that was called from the main by a SEGLD request to return to the instruction following the SEGLD request in the main program.

There are restrictions on the use of SEGRT:

1. SEGRT can only be used if the segment was loaded by a SEGLD request.

2. The segment must have been loaded from the main, not another segment.

3. SEGRT can only return to the main, not another segment.

The Macro/1000 calling sequence is as follows:

```
EXT  SEGRT
 :
JSB  SEGRT
DEF  *+1
- return -
 .
 .
```

EXAMPLE

Segmentation in Pascal is implemented via language constructs such as compiler options $SEGMENT$, and EXTERNAL declarations to Pascal libraries. A call to Pas.SegmentLoad is made from the main, then a call to the desired procedure or function in the segment must be made. Control is not automatically passed to the segment. There is, however, an automatic return to the Pascal main at the end of each procedure. A sample Pascal program follows:

```
PROGRAM segmt;
 TYPE seg_name = PACKED ARRAY [1..5] of CHAR;

 PROCEDURE seg_load $ALIAS 'Pas.SegmentLoad'$
       (name : seg_name);
       EXTERNAL;
 BEGIN
     seg_load ('segls');
     proc1;
     seg_load ('seg2s');
     proc2;
 END.


 $SEGMENT$
 PROGRAM segls;
 {Types and Vars - exact same as in main - use $INCLUDE$}
   PROCEDURE proc1;
     BEGIN
     END;
   PROCEDURE proc12;              { Not used in this example }
     BEGIN                        { when overlay brought in }
     END;
 .  { INDICATES END OF SEGMENT SEG1S }

 $SEGMENT$
 PROGRAM seg2s;
 {Types and Vars declaration same as main.}
   PROCEDURE proc2;
     BEGIN
     END
 .  { INDICATES THE END OF SEGMENT SEG2S }
```

SETTM - Set System Time

A program may change the system time by calling the SETTM system subroutine:

```
error = SETTM(HOUR,MINUTE,SECOND,MONTH,DAY,YEAR)
```

```
error - 0  no error, time is changed,
        -1 illegal parameter value

        HOUR    0-23
        MINUTE  0-59
        SECOND  0-59

        MONTH   1-12
        DAY     1-31
        YEAR    1978-1999
```

COMMENTS:

Each of the parameters including the return parameter, error, are single-word integers. All parameters are required. The hour must be given as 0 through 23, so 12:01 AM would be represented as hour=0, minute=1 and 11:05 PM would be hour=23, minute=5. The year may not be less than 1978.

The system time may be changed even if there are time-scheduled programs in the system. Programs which were scheduled absolutely (to run at a particular time) will run at that time. For example, if the current time is 1:00 PM, and a program is scheduled to run at 2:00 PM, and the system time is changed to 3:00 PM, the program will execute at 2:00 PM the following day. Programs that were scheduled to run relatively (after a certain amount of time) are put in the time list to run at an absolute time. The absolute time is based on the time the request was made. For example, a program that is to run 1 hour relative to 1:00 PM will be set to run at 2:00 PM. If SETTM is used to change the time to 3:00 PM before the 2:00 PM time occurs, the program will run at 2:00 PM the following day.

The Macro/1000 calling sequence is as follows:

```
        EXT SETTM
         .
         .
        JSB SETTM
        DEF RTN
        DEF HOUR
        DEF MINUTE
        DEF SECOND
        DEF MONTH
        DEF DAY
        DEF YEAR
RTN      .
         .
```

TATMP — Maps Track Assignment Table

The TATMP subroutine maps the Track Assignment Table (TAT) into the driver partition area in the user's map. The Track Assignment Table (TAT) resides in physical memory and is mapped in the driver area whenever it has to be used. This map will be preserved when the program is swapped. However, when an I/O request is made or a call to one of the EMA routines, (.EMAP, .EMIO, MMAP) is made, the mapping of the driver partition will be destroyed. A call to TATMP must be made if the Track Assignment Table has to be used after an I/O request or a call to the EMA routines is made.

```
CALL TATMP
```

The Macro/1000 calling sequence is as follows:

```
        EXT TATMP
         .
         .
        JSB TATMP
        DEF RTN
RTN      .
         .
```

Program Status Routines

COR.A, COR.B — Returns Address

The COR.A subroutine returns the address of the first word of available memory (high address +1) for a main program or program segment given the address of the main program or segment associated with the ID-segment specified.

The COR.B subroutine returns the first word of available memory for a main program given the address of its ID segment. For segmented programs, the address returned is equal to:

main high address + largest segment high address +1.

Not callable in FORTRAN or Pascal
See Macro/1000 calling sequence below

COMMENTS

For non-segmented programs, the addresses returned by COR.A and COR.B are the same.

On return from COR.A, the A-Register contains the high address +1 of the main program or segment associated with the ID segment specified. The B-Register is not used.

On successful returns from COR.B, the B-Register will contain the high address +1 of the main program associated with the specified ID segment. If the program is segmented, the B-Register will contain the high address +1 of the largest segment. The A-Register will contain 0.

On unsuccessful returns from COR.B, the A-Register will contain -1, and the B-Register will be meaningless. COR.B makes an error return if it is passed the address of a short ID segment.

The Macro/1000 calling sequence is as follows:

```
 :                    :
EXT  COR.A           EXT  COR.B
 :                    :
LDA  IDSEG           LDA  IDSEG
JSB  COR.A           JSB  COR.B
-return              -return-
 :                    :
```

IDSEG - ID segment address; must be long ID segment address for COR.B.

IDGET — Retrieve ID Segment

The IDGET subroutine retrieves the ID Segment address of a specified program.

```
IDSEG = IDGET (INAM)
-----
```

IDSEG - ID segment address. Contains the returned ID segment address of the specified program; set to 0 if the program does not exist.

INAM - Program name. 3-word array used to contain the 5-character ASCII name of the program to which the ID segment address is being requested.

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
          :
         EXT    IDGET
          :
         JSB    IDGET
         DEF    *+2
         DEF    INAM
          :
  INAM   ASC 3,PROGY
          .
```

On return, the following registers are set as indicated:

A-Register = ID segment address, or 0 if not found

E-Register = 0 if program found, or 1 if not found

B-Register = 0

IFBRK — Test Break Flag

The IFBRK subroutine tests the BREAK FLAG and clears it if it is set. The BREAK FLAG is set with the BR command described in the RTE-6/VM Terminal User's Reference Manual.

```
IF (IFBRK (IDMY)) n, m
```

IDMY - A dummy variable used to inform the FORTRAN compiler that an external function is being called.

n - Statement number that control branches to if the break flag was set; the flag will be cleared.

m - Statement number that control branches to if the break flag is not set.

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
      .
EXT   IFBRK
      .
      .
JSB   IFBRK
DEF   *+1
- return -
      .
      .
```

On return, the A-Register will contain -1 if the BREAK FLAG was set or will contain 0 if it was not set. The flag will be cleared if it was set.

LIMEN — Manages Memory

LIMEM allows a program to use and manage the memory between the end of its code and the end of memory available to the program. LIMEM finds and returns the limits of available memory.

For non-segmented programs:

```
CALL LIMEM (CODE [,FWAM [,WORDS]])
```

CODE	is positive or zero to lock memory, or negative to release memory.
FWAM	is the returned address of the first word of available memory after the program code.
WORDS	is the returned number of words of memory available after FWAM.

For segmented programs:

```
CALL LIMEM (CODE [,FWAM [,WORDS [,CURNT [,CWRDS]]]])
```

CODE	if negative, releases memory and returns to original FWAM. If positive or zero, CODE gets memory.
FWAM	is the returned first word of available memory after the largest segment (or after the main if the program is not segmented).
WORDS	is the returned number of words available after the largest segment (or after the main if the program is not segmented).
CURNT	is the returned first word available after the current segment (or after the main if no segments have been loaded).
CWRDS	is the returned number of words available after the current segment (or after the main if no segments have been loaded).

COMMENTS:

Do not use the LIMEM subroutine when writing programs in Pascal. The LIMEM subroutine interferes with Pascal's use of the Heap/Stack area.

The Macro/1000 calling sequence is as follows:

```
          EXT LIMEM
           .
           .
          JSB LIMEM
          DEF RTN
          DEF CODE
          DEF FWAM
          DEF WORDS
          DEF CURNT (optional)
          DEF CWRDS (optional)
     RTN   .
           .
```

Conversion Routines

$CVT3(CNUMD,CNUMO), $CVT1(KCVT) — Convert Binary to ASCII

These subroutines convert a positive integer binary number to ASCII.

```
CALL CNUMD ( n ,IBUF)
CALL CNUMO ( n ,IBUF)
I= KCVT (n)
```

n	- Actual binary number that is to be converted to ASCII; must be positive.
IBUF	- Three-word array that the ASCII representation (6 characters) of n is returned to. For CNUMD, a decimal representation is returned; for CNUMO, an octal representation is returned. Leading zeros are suppressed.
I	- Least significant two digits of the ASCII decimal representation of n.

COMMENTS:

The Macro/1000 calling sequence is as follows:

```
  :
EXT $CVT3 (or $CVT1)
  :
LDA n
  .
CLE (for octal results; E=0)
or
CCE (for decimal results; E=1)
  .
JSB $CVT3 (or $CVT1)
- return -
  :
```

Upon return the register contents will be as follows:

E-Register = 1

A-Register = $CVT3 - address of 3-word ASCII result.
$CVT1 - two least-significant characters of converted number.

B-Register = unchanged.

Miscellaneous

.DRCT — Resolves Indirect Address

The .DRCT subroutine resolves an indirect address within the map of the calling program.

```
$ALIAS    .DRCT =  'DRCT',   DIRECT
  .
  .
CALL      DRCT     (ADDR)
```

ADDR - Address to be resolved

COMMENTS:

The .DRCT subroutine returns with the A-Register set to the direct address of ADDR, the B-Register unaltered, and the E-Register lost. This routine is usually used when ADDR is external.

The Macro/1000 calling sequence is as follows:

```
          EXT .DRCT
           .
           .
          JSB .DRCT
          DEF ADDR
          - return -
           .
           .
```

FTIME — Returns Date and Time

The FTIME subroutine returns the date and time as an ASCII string.

CALL FTIME (IBUF) ----
IBUF - 15-word array that receives the ASCII string.

COMMENTS:

The format of the returned string is illustrated by the following example:

```
12:42 PM FRI., 23 OCT., 1981
```

The month will be returned as a three-character abbreviation followed by a period or (June, July) as four characters.

The Macro/1000 calling sequence is as follows:

```
        EXT FTIME
         .
         .
        JSB FTIME
        DEF RTN
        DEF IBUF
RTN      .
         .
```

INPRS — Inverse Parse

The INPRS subroutine converts a buffer of parsed data (produced by the PARSE subroutine) back to its original form.

```
CALL INPRS (IRBUF, NUPAR)
```

IRBUF - Parameter buffer. Contains the parameters to be converted back to an ASCII string; previously produced by the PARSE routine.

NUPAR - Number of parameters contained in IRBUF; obtained from word 33 of IRBUF returned by PARSE (NUPAR = IRBUF(33)).

COMMENTS:

IRBUF and NUPAR are obtained from a previous call to the PARSE routine or are formatted by the user as if they were; the function of INPRS is to reverse the action performed by PARSE.

The results of the INPRS operation are stored in IRBUF. The length of the resultant ASCII string will be eight times the number of parameters (8 X NUPAR).

The Macro/1000 calling sequence is as follows:

```
        EXT INPRS
         .
         .
        JSB INPRS
        DEF RTN
        DEF IRBUF
        DEF NUPAR
RTN      .
         .
```

MESSS — Programmatic Access

The MESSS subroutine provides programmatic access (limited by the user capability if under session control) to all system commands.

```
IA = MESSS (IBUF,INUM [,LU])
```

IA - 0 if no message is returned from the system; negative character count if message is returned (same as return of A-Register).

IBUF - Command buffer. Contains the ASCII command to be passed to the RTE-6/VM Operating System.

INUM - Character count. Number of characters (bytes) contained in the ASCII command.

LU - Replacement LU. If the command in IBUF is a RU or an ON, and the first parameter in the parameter string is zero or absent, then LU will be inserted as the first parameter.

COMMENTS:

If the operating system returns information, it is placed in IBUF and the character count (negative) is placed in IA (and the A-Register). The command buffer (IBUF) should be at least 14 words long to allow for the largest possible system return.

If the command was a RU or ON command, the father ID-segment-word 33 will be propagated to the son's ID-segment-word 33.

The Macro/1000 calling sequence is as follows:

```
        EXT  MESSS
         .
         .
        JSB  MESSS
        DEF  RTN
        DEF  IBUF
        DEF  INUM
        DEF  LU    (optional)
RTN      .
         .
         .
IBUF    BSS
INUM    DEC
LU      DEC
```

On return the A-Register will be 0 if no information is returned, or will be the negative character count if information is returned.

PARSE ($PARS) — Parse String

The PARSE subroutine (or $PARS) allows a program to parse a string into separate parameters.

```
CALL PARSE (IBUFA,ICON,IRBUF)
            -----
```

IBUFA - Source buffer. Contains the ASCII string to be parsed.

ICON - Character count. Number of characters in the ASCII string.

IRBUF - Receiving buffer. A 33-word buffer that contains the results of the parse operation.

COMMENTS:

The results of the parse operation are stored in IRBUF using four words of IRBUF to represent each parameter found in IBUFA. The function of these four words is summarized below:

WORD	ENTRY	
1	FLAG WORD	0 = NULL 1 = NUMERIC 2 = ASCII
2	VALUE(1)	0 If NULL; Value if Numeric; first two characters if ASCII.
3	VALUE(2)	0 If Null or numeric else the 3rd and 4th characters (ASCII).
4	VALUE(3)	0 If NULL or numeric else the 5th and 6th characters (ASCII).

The Macro/1000 calling sequence is as follows:

```
        EXT $PARS
         .
         .
        LDA IBUFA
        LDB ICON
        JSB $PARS
        DEF IRBUF
        - return -
```

The following example illustrates the PARSE and INPRS subroutines.

```
        PROGRAM IMP
        IMPLICIT INTEGER (A-Z)
        INTEGER GET (40)
        INTEGER PBUF(33),PBUF2(33)

C     GET THE PARAMETER STRING
        CALL GETST (GET, -80, TLOG)

C     PRINT IT
        CALL EXEC (2, 1, GET, -TLOG)

C     PARSE IT
        CALL PARSE(GET,TLOG,PBUF)

C     COPY THE RESULT TO A SECOND BUFFER
        DO 50,I=1,33
        PBUF2(I)=PBUF(I)
  50    CONTINUE

C     INVERSE PARSE THE SECOND BUFFER
        CALL INPRS(PBUF2,PBUF2(33))

C     PRINT THE TWO BUFFERS IN OCTAL
        DO 60 I-1,33
           WRITE (1,100) PBUF(I), PBUF2(I)
 100       FORMAT(2010)

  60    CONTINUE

C     PRINT THE RESULT
C     SYNTACTICALLY EQUIVALENT TO ORIGINAL
        CALL EXEC(2,1,PBUF2,PBUF2(33)*-8))

        END
```

SYCON — Writes Message to System Console

The SYCON subroutine writes a message to the system console (system LU 1).

```
CALL SYCON (IBUF, ILEN)
```

IBUF - Buffer that contains the message to be written.

ILEN - Length of IBUF, positive value indicates the number of words and a negative value indicates the number of characters.

COMMENTS:

This routine bypasses the Session Switch Table (SST) and writes directly to system LU 1.

The Macro/1000 calling sequence is as follows:

```
        EXT SYCON
         .
         .
        JSB SYCON
        DEF RTN
        DEF IBUF
        DEF ILEN
RTN      .
         .
```

TMVAL — Reformats System Time

The TMVAL subroutine reformats the system millisecond time format (double-word negative integer) into an array of time parameters.

```
CALL TMVAL (ITIM, ITMR)
           ----
```

ITIM - Two-word negative time value in tens of milliseconds. This double-word integer can be obtained from the system entry point $TIME (real-time clock) or the time value stored in a program's ID segment.

ITMR - Time array. 5-word array to which the system returns the reformatted time. The array is set up as:

```
ITMR(1) = tens of milliseconds
ITMR(2) = seconds
ITMR(3) = minutes
ITMR(4) = hours
ITMR(5) = day of year (julian)  -  not related to call
          values.
```

COMMENTS:

The next scheduled execution time of a program currently in the time list can be obtained from the program's ID segment and then formatted into an array of time parameters by the TMVAL routine. The following program segment illustrates this programming sequence.

```
DIMENSION   INAM(3),ITMR(5),IARRAY(2)
DATA INAM/2HPR,2H06,2H1/
 :
IDSEG=IDGET (INAM)
ITAD=IDSEG+18
 :
IARRAY(1)=IGET(ITAD)
IARRAY(2)=IGET(ITAD+1)
 :
CALL TMVAL (IARRAY,ITMR)
```

The Macro/1000 calling sequence is as follows:

```
        EXT TMVAL
         .
         .
        JSB TMVAL
        DEF RTN
        DEF ITIM
        DEF ITMR
RTN      .
         .
```

XQPRG - Load and Execute a Program

A call to XQPRG will execute a program. XQPRG checks to see whether the program occupies an ID segment, sets up an ID segment if necessary, and executes the program. If XQPRG sets up an ID segment, the ID segment is released after the program terminates.

```
CALL XQPRG (IDCB,ICODE,NAME,IFIVE,IBUF,IL,
                                   IPRTN,IERR[,ISECU[,ICR]])
                                   ----- ----
```

IDCB - the Data Control Block; 144-word array for use by XQPRG.

ICODE - the EXEC request code; 1-word variable used to schedule the program (9, 10, 23, or 24).

NAME - the program name; 3-word array; contains 5-character program name or 6-character program file name.

IFIVE - the scheduling parameters; 5-word array to be passed to program.

IBUF - the user buffer; array to be passed to program.

IL - the user buffer length; 1-word variable; if positive, contains length in words; if negative, contains length in characters.

IPRTN - the return parameters; 5-word array passed back from program.

IERR - for error return; 1-word variable in which error code is returned; see below for details.

ISECU - the security code; optional 1-word variable; must be specified to execute program in file created with a negative security code.

ICR - the cartridge reference; optional 1-word variable; if zero, the first file found with the specified name will be executed; if set only the file on the specified cartridge will be executed. If used, ISECU must be specified also.

COMMENTS:

ICODE is similar to the EXEC schedule request code. It is:

9 for immediate schedule with wait.
This is the same as the RU operator command if the program is dormant. If the program is already executing, an error is returned.

10 for immediate schedule without wait.
This is the same as the XQ operator command if the program is dormant. If the program is already executing, an error is returned.

23 for queue schedule with wait.
Same as the RU operator command.

24 for queue schedule without wait.
Same as the XQ operator command.

IERR error codes returned are:

0 = Successful execution, no scheduling errors

1 = Duplicate program name (IPTRN(1) = 0)

2 = No ID segments available (IPRTN(1) = 14)

3 = Program not found (IPRTN(1) = -6 or -32)

4 = File open error other than program not found (IPRTN(1) = FMP error number)

5 = File close error (IPRTN(1) = FMP error number)

6 = RP error other than duplicate program name or no ID segments available (IPRTN(1) = FMP error number)

7 = Program busy (IPRTN(1) = 0)

8 = Program was scheduled, but then aborted (IPRTN(1) = 100000B)

9 = Scheduling error other than "program not found" (IPRTN(1) = ASCII "SC" code ('04,'10').)

10 = Illegal ICODE parameter (should be 9, 10, 23, or 24).

11 = Not enough parameters

The Macro/1000 calling sequence is as follows:

```
        EXT XQPRG
         .
         .
        JSB XQPRG
        DEF RTN
        DEF IDCB
        DEF ICODE
        DEF NAME
        DEF IFIVE
        DEF IBUF
        DEF IL
        DEF IPRTN
        DEF IERR
        DEF ISECU
        DEF ICR
RTN     .
        .
```

EXAMPLE:

```
   DIMENSION  IDCB(144), NAME(3), IBUF(6), IDUM(5)

   DATA  NAME/6H PARENT/
   DATA  IBUF/12H HELLO THERE/
   IL=6
   ICODE=10

   CALL XQPRG(IDCB,ICODE,NAME,IDUM,IBUF,IL,IDUMY,IERR)

   IF (IERR .NE. 0)  GOTO 99

99 ERROR MESSAGE
```

Appendix A
Error Messages

Appendix A includes the following error messages:

EXEC Call Errors

FMP Call Errors

VMA/EMA Call Errors

Executive Error Messages

When RTE-6/VM discovers an Executive error, it normally aborts the program, releases any disc tracks assigned to the program, issues an error message to the system console (and session terminal if in session environment) and proceeds to execute the next program in the scheduled list.

The user may specify the no-abort or no-suspend option for a program for some Executive error conditions. See Chapter 2 for a detailed discussion of this option.

The error messages described below are those that may occur while accessing the Executive. They are grouped by type. Table A-1 contains a summary of all possible errors associated with EXEC calls.

Memory Protect Violations

The RTE-6/VM operating system is protected by a hardware memory protect. Consequently, any user program that illegally tries to modify or jump to the operating system will cause a memory protect interrupt. The operating system intercepts the interrupt and determines its legality. If the memory protect is illegal, the program is aborted and the following message is displayed on the system console:

```
MP INST = xxxxxx      (offending octal instruction code)
ABE pppppp qqqqqq r   (contents of A, B and E registers at abort)
XYO pppppp qqqqqq r   (contents of X, Y and O registers at abort)
LEAF NODE = nnn       (nnn = leaf node number of the currently
MP yyyyy  zzzzz                enabled path, only for MLS programs)
yyyyy ABORTED      (yyyyy=program name; zzzzz=violation address)
```

Dynamic Mapping Violations

A dynamic mapping violation occurs when an illegal read or write occurs to a protected page of memory. This may happen when a user program tries to write beyond its own address space to non-existent memory or to some other program's memory. In this case, the program is aborted and the following message is issued:

```
DM VIOL = wwwww       (contents of DMS violation register)
DM INST = xxxxxx      (offending octal instruction code)
ABE pppppp qqqqqq r   (contents of A, B and E-Registers at abort)
XYO pppppp qqqqqq r   (contents of X, Y and O-Registers at abort)
LEAF NODE = nnn       (nnn = leaf node number of the currently
DM yyyyy  zzzzz                enabled path, only for MLS programs)
yyyyy ABORTED      (yyyyy=program name; zzzzz=violation address)
```

Dispatching Errors

It is possible for programs to be scheduled and discover at a later time that there is no partition large enough to dispatch the program. This could occur if a parity error downed a partition while the program is currently swapped out to the disc and that partition was the largest of its type (i.e., BG, RT, or EMA). If this occurs, the program will be aborted with a DP error. A DP error can also occur on a program swap. The format of the error message is:

```
ABE pppppp qqqqqq r     (contents of A,B, and E registers at abort)
XYO pppppp qqqqqq r     (contents of X,Y, and O registers at abort)
DP   yyyyy  zzzzz     (yyyyy=program name; zzzzz=violation address)
yyyyy ABORTED
```

EX Errors

It is possible to execute in the privileged mode, that is, with the interrupt system off. Therefore, the user may not make EXEC calls in this mode because the memory protect, which is the access vehicle to EXEC, is off. An attempt to make an EXEC call with the interrupt system off causes the calling program to be aborted and the following message to be issued:

```
ABE pppppp qqqqqq r    (contents of A,B and E registers at abort)
XYO pppppp qqqqqq r    (contents of X,Y and O registers at abort)
LEAF NODE = nnn        (nnn = leaf node number of the currently
EX   yyyyy  zzzzz             enabled path, only for MLS programs)
yyyyy ABORTED       (yyyyy=program name; zzzzz=violation address)
```

Unexpected DM and MP Errors

The operating system handles all DM and MP violations. Some of these violations are legal; others are not. In any case, the operating system associates these violations with program activity. A DM or MP violation occurring when no program is active is an unexpected violation. Since no program is present there is no program to abort. In such a case, one of the following messages will be issued:

```
DM VIOL = wwwwww      (contents of DMS violation register)
DM INST = xxxxxx      (offending octal instruction code)
ABE pppppp qqqqqq r   (contents of A, B and E registers at abort)
XYO PPPPPP qqqqqq r   (contents of X, Y and O registers at abort)
DM <INT>      0

     or

MP INST = xxxxxx      (offending octal instruction code)
ABE pppppp qqqqqq r   (contents of A, B and E registers at abort)
XYO pppppp qqqqqq r   (contents of X, Y and O registers at abort)
MP <INT>      0
```

Both of the above messages specify <INT> as the program name to signal the user that an unexpected memory protect or dynamic mapping violation error has occurred. Either is a serious violation of the operating system integrity. Usually, it indicates that user-written software (driver, privileged subroutine, etc.) has damaged the operating system integrity or has inadequately performed required (driver) system housekeeping. However, it could also mean that the CPU has failed and that the operating system detected the failure in time to prevent a system crash.

If this error occurs, it is recommended that all users on the system save whatever they were doing (i.e., finish up editing, etc.) and reboot the system. If only HP modules are present in the operating system, CPU failure is a highly likely cause of the error and CPU diagnostics should be run prior to rebooting.

TI, RE and RQ Errors

The following errors have the same format as the MP and DM error returns except that the register contents are not reported:

Error	Meaning
TI	Batch program exceeds allowed time.
RE	Reentrant subroutine attempted recursion.
RQ	Illegal request code (is not between 1 and 26) An RQ00 error means that the address of a returned parameter is below the memory protect fence.

Disc Parity Error (Track Error)

Upon detecting a disc parity error, the program that encountered the error will be aborted and the following message will be issued:

```
     TR nnnnn EQTxxx,Uyy S
or   TR nnnnn EQTxxx,Uyy U
```

where:

nnnnn = Track number of track containing the error.

xxx = EQT of the disc.

yy = Subchannel of the disc.

S = System request encountered the error.

U = User request encountered the error.

For CS80 discs, additional error information may be provided if program CSERR has been configured into the system at generation time. Refer to the DVM33 Disc Driver Manual for details.

Parity Errors

Upon detecting a "hard" parity error (i.e., one that is reproducible). RTE will abort the program that encountered the parity error, bring down the program's partition, and issue the following message:

```
PE  PG#    nnnnn BAD
ABE aaaaaa bbbbbb e
XYO xxxxxx yyyyyy o
PE  ppppp  mmmmmm
ppppp ABORTED
```

where:

nnnnn = physical page number where the parity error was detected (page number counting starts at zero).

ABE = contents of the A, B, and E-Registers respectively when the parity error was detected.

XYO = contents of the X, Y, and O-Registers respectively when the parity error was detected.

ppppp = program name.

mmmmmm = logical memory address of parity error.

If the program was disc-resident, the following message will be issued in addition to those listed above:

```
PART'N xx DOWN
PART'N yy DOWN
```

where:

xx = the partition the program was running in.

yy = the mother partition program if any are affected

Alternately, if xx is a mother partition, then yy is a subpartition that contained the parity error. In either case, partition xx and yy will no longer be available for running user programs until the system is next booted up.

Upon detecting a "soft" parity error (i.e., one that is not reproducible), RTE will abort the user program. However, the program partition is not brought down as in the case for a hard parity.

The following message is then issued:

```
SOFT PE  PG# nnnnn
ABE  aaaaaa  bbbbbb e
XYO  xxxxxx  yyyyyy 0
PE   ppppp   mmmmmm
ppppp ABORTED
```

The parameters are the same as those for a "hard" parity described above.

If RTE is not able to locate the physical page number of the parity error, the following message is issued:

```
PE @mmmmmm
DMS STAT=zzzzzz
```

where:

mmmmmm = Logical address of parity error.

zzzzzz = DMS status register.

A soft parity error occurs if RTE-6/VM cannot reproduce a parity error while reading the logical parity error address in the system map, Port A map, or the user map. When a second soft parity error occurs in the same memory location, a hard parity is generated and the partition is set to undefined status (it becomes unavailable to any user).

A parity error occurring within the operating system itself, a driver or system table area causes the system to execute a HLT 102005. The A- and B-Register will contain the following:

A-Register = physical page number where the parity error was detected (page number counting starts at 0).

B-Register = logical memory address of the parity error.

A parity error occurring in a DCPC transfer when the operating system is executing in the System Map causes the system to execute a HLT 103005, where the A and B-Registers are as above.

Note: If CPU switch AISI is set to "HALT" instead of "INT/IGNORE", the CPU halts when a parity error occurs, and the parity error is not processed by RTE.

Other EXEC Errors

The general format for the following errors is

type name address

where:

type = a four-character error code (DR, SC, IO, RN, LU; see below)

name = the program that made the call.

address = the location of the call (equal to the exit point if the error is detected after the program suspends).

Disc Allocation Error Messages

DR01 = Not enough parameters

DR02 = Number of tracks zero, illegal logical unit, or number of tracks to release is zero or negative.

DR03 = Attempt to release track assigned to another program.

Schedule Call Error Codes

SC00 = Batch program attempted to suspend (EXEC (7)).

SC01 = Missing parameter.

SC02 = Illegal parameter.

SC03 = Program cannot be scheduled.

SC03 INT = Occurs when an external interrupt attempts to schedule a program that is already scheduled. RTE-6/VM ignores the interrupt and returns to the point of interruption.

SC04 = program name specified is not a subordinate (or "son") of the program issuing the completion call.

SC05 = Program given is not defined. The format for the SC05 error is:

```
PROG/SEGMENT nnnnn
SC05 name    ADDRESS
```

where nnnnn is the program/segment not found for the schedule request and name is the calling program name.

SC06 = No resolution code in Execution Time EXEC Call (not 1, 2, 3, or 4).

SC07 = Prohibited memory lock attempted.

SC08 = The program just scheduled is assigned to partition smaller than the program itself or to an undefined partition. Unassign the program or reassign the program to a partition that is as large or larger than the program.

SC09 = The program just scheduled is too large for any partition of the same type. For example, trying to schedule a 23K background program when the largest background partition is only 21K.

SC10 = Not enough system available memory for string passage.

SC11 = Attempt to schedule (or time schedule) a program already in the time list for another session.

SC12 = The program tried to do an EXEC 8 call to load a multi-level segment program.

SC13 = The main program and the segments are not on the same disc cartridge.

SC14 = Track ownership in the Track Assignment Table (TAT) does not correspond to the ID segment for program.

I/O Call Error Codes

IO00 = Illegal class number. Outside table, not allocated, or bad security code.

IO01 = Not enough parameters were specified.

IO02 = Illegal logical unit number was specified.

IO03 = Illegal EQT referenced by LU in I/O call (Select code=0).

IO04 = Illegal user buffer. Extends beyond RT/BG area or not enough system available memory to buffer the request.

IO05 = Illegal disc track or sector.

IO06 = Reference to a protected track; or using LG tracks before assigning them (refer to LG, Chapter 3).

IO07 = Driver has rejected the call.

IO09 = Overflow of LG area.

IO10 = Class GET call issued while one call already outstanding.

IO11 = Type 4 program made an unbuffered I/O request to a driver that did not do its own mapping.

I012 = Logical unit not defined for this session. The format for I012 error is:

SES LU = XX
I012 name ADDRESS

where XX is the session LU not present in the Session Switch Table (SST) and name is the calling program name.

I013 = LU was either locked to another program, or pointed to an EQT which was locked to another program.

I014 = An I/O request was issued with the no-suspend option.

I015 = Buffer size of a type 6 program is greater than what will fit in the user map. Note that the buffer and the driver area up to the SSGA, including the base page, must be mappable (see the Standard I/O EXEC Calls section).

I016 = CPU backplane failure or I/O extender timing failure.

I020 = Read attempted on write-only spool file.

I021 = Read attempted past end-of-file (EOF).

I022 = Second attempt to read JCL card from batch input file by other than FMGR.

I023 = Write attempted on read-only spool file.

I024 = Write attempted beyond end-of-file (EOF), usually spool file overflow.

I025 = Attempted to access spool LU that is not currently set up.

I026 = The I/O request was made to t spool file that has been terminated by the GASP KS command.

I0xx errors, where 33<xx<60, are reserved for HP subsystem use.

I/O Error Message Format

The following error message format is used to report I/O errors:

```
  NR
IOET L xxx E yyy Szz qqq
  PE
  TO
```

where:

xxx = the device's logical unit number

yyy = the device's EQT number

zz = the subchannel for the device

qqq = device status returned by driver

NR = device not ready

ET = end of tape

PE = transmission parity error

TO = the device has timed out

NOTE

If a driver is down when an I/O request is made, the device status cannot be assumed to be valid. Therefore, the device status information of the error diagnostic is replaced with three asterisks, indicating that the device was already down when the request was issued.

Program Management Error Codes

RN00 = No option bits set in call.

RN01 = No resource numbers in system.

RN02 = Resource Number not defined.

RN03 = Unauthorized attempt to clear a LOCAL Resource Number.

Logical Unit Lock Error Codes

LU01 = Program has one or more logical units locked and is trying to lock another with wait.

LU02 = Illegal logical unit reference (greater than maximum number).

LU03 = Not enough parameters furnished in the call. Logical unit reference less than one. Logical unit not locked to caller.

LU04 = Lock attempted on logical unit not defined for this session.

CLRQ

CL01 = Illegal class number or no class numbers defined in the system.

CL02 = Parameter or calling sequence error.

Executive Halt Errors

There are several HLT instructions included in the RTE operating system that indicate a serious violation of the integrity of the operating system. Usually, these errors indicate that the CPU or one of its subsystems (DCPC, Memory Protect, etc.) has failed. However, they could indicate that user-written software (driver, privileged subroutine, etc.) has damaged the operating system integrity or has inadequately performed required (driver) system housekeeping. If these HLT's occur, it is recommended that the user check out his hardware with the appropriate diagnostics.

HLT 0 Located in Table Area I.

HLT 2 Memory wrap-around halt. Located in location 2 of the system map.

HLT 3 Memory wrap-around halt. Located in location 3 of the system map.

HLT 4 Powerfail occurred and one of the following is true:

* Powerfail/auto-restart subsystem not installed.

* CPU was halted when powerfail occurred.

* Powerfail code did not execute completely.

Other system HLT's exist for which there is some corrective action:

HLT 5 Parity error in system map. See Parity Error discussion in this section.

HLT 5,C Parity error in a DCPC transfer when operating system was executing in the system map. See Parity Error discussion in this section.

HLT 6 System tried to remove a partition from a list and the partition was not found there. System lost track of its memory (confusion halt). The operating system may be corrupt or a bug may exist. Reboot the system and if the problem persists, call your HP representative.

HLT 10B At startup, the system discovered that there was no partition large enough to execute FMGR or D.RTR.

HLT 20B Uninstalled Memory Halt. The system will halt if memory has been defined but is not installed. The uninstalled page number will be displayed in the A- and B-Registers. Reboot and reconfigure the memory in the system. Alternately, check the memory address jumpers on the memory just installed.

HLT 21B RTE-6/VM operating system or VMA firmware failed selftest.
or
(105355B)

A summary of EXEC call error messages is provided in Table A-1.

55B,
56B,
57B While dispatching a program the operating system encountered an unexplainable condition. The operating system may be corrupt or a bug may exist. Reboot the system and if the problem persists, call your HP representative.

Error Routing

All EXEC call error messages are reported to the "session" terminal as well as the system console. The term "session" is used to mean the terminal from which the program making the invalid request was invoked. This means that the routing of errors to the correct console is not dependent on the Session Monitor.

Table A-1, at the end of this section, summarizes the EXEC error messages.

Before the message is issued to the session terminal, several system level status checks are performed. Each of the following tests must pass or the echo to the session terminal is not performed.

* The session terminal must not be down.

* Enough System Available Memory is available for the error message.

* The session terminal must not have reached its buffer limit. If the buffer limit has been reached, only one error message will be allowed until:

  1) any previous system error message is completed or,

  2) The buffer limits are no longer exceeded.

Equipment error messages will always be sent to the session terminal (in addition to the system console) if any of the following conditions are true:

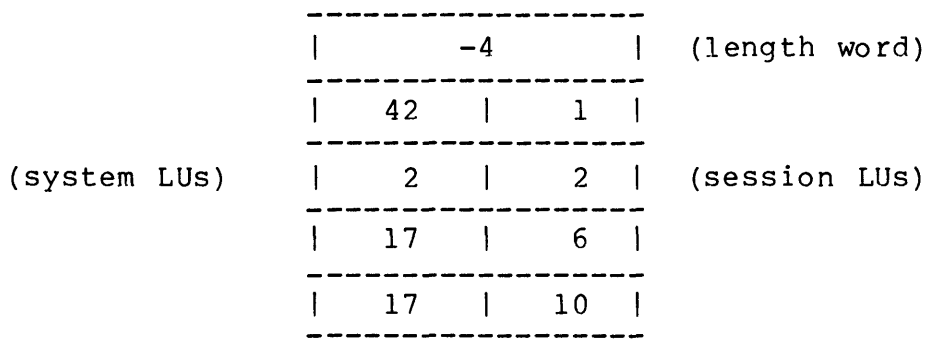
* The error occurred on a non-buffered request.

* The error occurred on the initiation of a request.

In either case, the ID segment (word 33) of the requesting program provides the information required to issue the error message to the "session" terminal.

If the error (parity error) occurred during a DMA transfer, the request is checked to determine the caller's identity. If a user program can be identified, the ID segment once again identifies the destination of the error message.

If the requesting program is under session control, RTE-6/VM scans the program's Session Switch Table for every system LU to be put down. For every match found (system LU=high byte of switch table), the system issues the error diagnostic to the session terminal, but uses the session LU as defined in the SST. For example, say system LU 17 goes down (needs to be write enabled) and the user has the following SST:



The system console will receive the following diagnostic:

```
IONR L  17 E  10 S  0  004
```

The session terminal (system LU 42 in this example) will receive the following diagnostics:

```
IONR L*   6 E  10 S 0 004
IONR L*  10 E  10 S 0 004
```

The three octal numbers at the end of the message represent the device status returned by the driver. Refer to the Device Status Table in Chapter 3.

The asterisk before the LU indicates that the LU reported is the session related value.

NOTE

If the above device was buffered, the calling program would still be scheduled and capable of executing. The next access of the down device would cause:

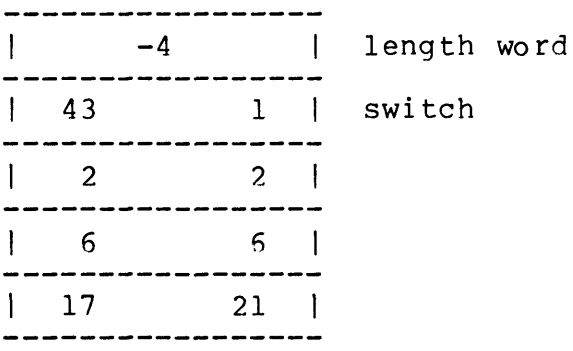
* The program to be suspended.

* The following diagnostic messages to be issued to the session console.

```
IONR  L*   6   E  10    S0  ***
IONR  L*  10   E  10    S0  ***
```

The system console does not receive any additional message.

If another session, represented by the following SST,



attempts to access the same down device, the session terminal (system LU 43 in this example) will receive the following diagnostic:

```
IONR   L*   21   E  10   S0   ***
```

Table A-1. EXEC Call Error Summary

ERROR MEANING	READ 1	WRITE 2	CONTROL 3	PROGRAM TRACK ALLOCATE 4	PROGRAM TRACK RELEASE 5	PROGRAM COMPLETION 6	PROGRAM SUSPEND 7	PROG. SEG. LOAD 8	PROG. SCHED. W/WAIT 9	PROG. SCHED. WO/WAIT 10	TIME REQUEST 11
DR01 Not Enough Parameters 1. Less than 4 parameters. 2. Less than 1 parameter. 3. Number = -1. 4. Less than 3 (not -1).				1	2						
DR02 Illegal Track Number or Logical Unit Number 1. Track number = 0. 2. Logical Unit not 2 or 3. 3. Deallocate 0 or less Tracks.				1	2 3						
DR03 Attempt to release Track assigned to another program.					X						
IO00 Illegal Class Number 1. Outside Table. 2. Not allocated. 3. Bad Security Code.											
IO01 Not Enough Parameters. 1. Zero parameters. 2. Less than 3 parameters. 3. Less than 5/disc. 4. Less than 2 parameters. 5. Class word missing.	1 2 3	1 2 3	1								
IO02 Illegal Logical Unit. 1. 0 or maximum. 2. Class request on disc LU. 3. Less than 5 parameters and X-bit set.	1 3	1 3	1								
IO03 Illegal EQT command by LU in I/O call; delete code = 0.	X	X	X								
IO04 Illegal User Buffer. 1. Extends beyond RT/BG area. 2. Not enough system memory to buffer the request.	1	1									
IO05 Illegal Disc Track or Sector 1. Track number maximum. 2. Sector number 0 or maximum.	1 2	1 2									
IO06 Attempted to WRITE to LU2/3 and track not assigned to user or globally, or not to next load-and-go sector. Illegal WRITE to a FMP track. Attempted to use copy of loader to make permanent load or delete.		X									
IO07 Driver has rejected request and request is not buffered.	X	X	X								
IO08 Disc transfer implies track switch (LU2/3).	X	X									
IO09 Overflow of LG area.		X									
IO10 Class GET and one call already outstanding.											
IO11 Illegal User Map request for System Driver area.	X	X	X								
IO12 LU not defined for this session.	X	X	X								

8300-01

Table A-1. EXEC Call Error Summary (Continued)

PROG. SCHED. TIME 12	I/O SATAUS 13	STRING PASSAGE 14	GLOBAL TRACK ALLOCATE 15	GLOBAL TRACK RELEASE 16	CLASS I/O READ 17	CLASS I/O WRITE 18	CLASS I/O CONTROL 19	CLASS I/O WRITE/ READ 20	CLASS I/O GET 21	PROG. SWAPPING CONTROL 22	PROG. SCHED. QUEUE W/WAIT 23	PROG. SCHED. QUEUE WO/WAIT 24	RNRQ LURQ
			1	3 4									
			1	2 3									
					1 2 3	1 2 3	1 2 3	1 2 3	1 2 3				
	1 4				1 2 5	1 2 5	1 5	1 2 5					
	1				1 2 3	1 2 3	1 2 3	1 2 3					
	X				X	X	X	X	X				
					2	1 2	2	1 2	1				
									x				
	x				x	x	x	x					

Table A-1. EXEC Call Error Summary (Continued)

ERROR MEANING	READ 1	WRITE 2	CONTROL 3	PROGRAM TRACK ALLOCATE 4	PROGRAM TRACK RELEASE 5	PROGRAM COMPLETION 6	PROGRAM SUSPEND 7	PROG. SEG. LOAD 8	PROG. SCHED. W/WAIT 9	PROG. SCHED. WO/WAIT 10	TIME REQUEST 11
LU01 Program has one or more logical units locked and is trying to LOCK another with WAIT.											
LU02 Illegal logical unit reference (greater than maximum number)											
LU03 Not enough parameters furnished in the call. Illegal logical unit reference (less than one). Logical unit not locked to caller.											
LU04 LU not defined for this session.											
RQ00 Return buffer below memory protect fence.	X	X		X							X
RQ EXEC call contains an illegal request code. 1. Return address indicates less than one or more than seven parameters. 2. Parameter address indirect through A- or B-Register. 3. Request code not defined or not loaded.	X	X	X	X	X	X	X	X	X	X	X
RN00 No option bits set.											
RN01 No resource numbers in system.											
RN02 Resource number not in Table (undefined).											
RN03 Unauthorized attempt to clear a LOCAL Resource Number.											
SC00 Batch program cannot suspend.							X				
SC01 Missing Parameter. 1. Segment name missing. 2. Not 4 or 7 parameters in Time Call. 3. Not 4 parameters in String Passage Call or partition status call.								1			
SC02 Illegal Parameter. 1. Option word is missing or not 0, 1, 2, or 3. 2. Read/write word in String Passage Call is not 1 or 2.						1					
SC03 Program cannot be scheduled. 1. Not a segment. 2. Is a segment.								1	2	2	
SC04 Attempted to control a program that is not a "Son."						X					
SC05 Program given is not defined. 1. No segment. 2. No program. 3. "Son" not found.						3		1	2	2	
SC06 Resolution not 1, 2, 3, or 4.											
SC07 Prohibited core memory lock attempted.											
SC08 Partition is too small. 1. Assigned partition too small for program. 2. Segment too large for partition. (Segment not relocated with main.)								2	1	1	
SC08 Assigned partition is too small for program.									X	X	
SC09 Program too large for any partition of same type.									X	X	
SC10 Not enough system available memory for string passage.									X	X	
SC11 Attempted to schedule a program already schedule from another session.									X	X	X

Table A-1. EXEC Call Error Summary (Continued)

PROG. SCHED. TIME 12	I/O SATAUS 13	STRING PASSAGE 14	GLOBAL TRACK ALLOCATE 15	GLOBAL TRACK RELEASE 16	CLASS I/O READ 17	CLASS I/O WRITE 18	CLASS I/O CONTROL 19	CLASS I/O WRITE/ READ 20	CLASS I/O GET 21	PROG. SWAPPING CONTROL 22	PROG. SCHED. QUEUE W/WAIT 23	PROG. SCHED. QUEUE WO/WAIT 24	RNRQ	LURQ
														X
														X
														X
														X
	X		X				X	X	X	X				
X	X	X	X	X	X	X	X	X	X	X	X	X		
													X	
													X	
													X	
													X	
2		3												
		2								1				
											2	2		
2											2	2		
										X				
													X	
													X	
											X	X		
		X												

FMP Error Codes

FMP error codes are equivalent to negative FMGR error codes.

NOTE: The CI file system can produce additional FMP error codes. Refer to the CI Users Manual for a list of these error messages.

FMGR-105

D.RTR DIRECTORY TRACK BUFFER TOO SMALL

The directory buffer in the D.BUF routine was defined to be less than 512 words when loaded with D.RTR. Change the size of the directory buffer by increasing the constant defined by D.LEN in the source code, then reassemble D.BUF and regenerate the system with the new relocatable module.

FMGR-104

REQUEST EXTENT IS MISSING

FMGR-103

DIRECTORY IS CORRUPT

This error normally occurs when a disc is mounted before it is initialized. During a directory lock done by MC, DC, IN, PK, CR, or PU the directory is scanned for internal consistency. When this error occurs, copy the file to another disc or store the ones that you need.

FMGR-102

ILLEGAL D.RTR CALL SEQUENCE

A lock was not requested before a D.RTR call, the file in the call was not opened exclusively. Possibly an operator error, such as removing a cartridge without first dismounting it.

FMGR-101

ILLEGAL PARAMETER IN D.RTR CALL

Possibly an operator error. Check the previous entries for illegal or misplaced parameters. This error can also occur during a request to create a scratch file, if a file with the same name already exists and is open to another program. If D.RTR cannot purge the existing scratch file, this error is returned. See the system manager.

FMGR-099

DIRECTORY MANAGER EXEC REQUEST WAS ABORTED

An EXEC request made by D.RTR was aborted. Make sure that all discs being accessed are up. Notify system manager.

FMGR-052

SPOOL SHUT DOWN. SPOOL FILE SETUP FAILED

Spool shut down is in progress. A write (WR) or read/write (BO) spool file cannot be set up at this time. Start up spooling using the GASP SU command, and try to set up the spool file again.

FMGR-049

COPY VERIFY FAILED

The CO command detected an error during the verification of the file just copied. The 10G global contains the name of the file that failed the verification. Retry the copy by specifying the name in the 10G global as the starting file in the CO command.

FMGR-048

SPOOL NOT INITIALIZED OR SMP CANNOT BE SCHEDULED

Spooling has not been initialized (run GASP to do so), the SMP program cannot be found, or there are no partitions large enough to run it. The default for SMP is real-time (type 2) sized to 6 pages.

FMGR-047

NO SESSION LU AVAILABLE FOR SPOOL FILE

If the session LU to be used for the spool file is not specified during set up, SMP allocates a session LU less than 64 that is not already used in the session switch table. Use the :SL,lu,- command to release a session LU in the spare part of your session switch table.

FMGR-046

GREATER THAN 255 EXTENTS

You cannot create a file extent numbered greater than 255; make the main file size larger. If generated during an SM command, the message is not put in the message file, but is truncated at the last valid message.

FMGR-041

NO ROOM IN SST

There are no spare entries left in the session switch table. Spare entries can be recovered by using the :SL,lu,- command, where lu is a session logical unit number that is not needed.

FMGR-040

LU NOT FOUND IN SST

You are trying to access an LU that is not in your session switch table. Use the SL command to add the LU to your SST.

FMGR-039

SPOOL LU NOT MAPPED TO THE SPOOL DRIVER

A spool LU must point to a spool EQT. Switch all spool LUs to point to spool EQTs and try the spool file set up again.

FMGR-038

ILLEGAL SCRATCH FILE NUMBER

The range for scratch file numbers is 0 through 99. Try again with a number in the correct range.

FMGR-037

ATTEMPT TO PURGE AN ACTIVE TYPE 6 FILE

Type 6 files that have been RP'd cannot be purged. Issue an OF,prog command for the program if no one is running it, and try again.

FMGR-036

LOCK ERROR ON DEVICE

A call to OPENF failed to lock a device. The device may be already locked, or there may be no resource numbers available.

FMGR-035

ALREADY 63 DISCS MOUNTED TO SYSTEM

There are already 63 discs mounted. A disc will have to be dismounted before a new one may be mounted.

FMGR-034

DISC ALREADY MOUNTED

A disc with this ID is already mounted in the cartridge directory. Either dismount the duplicate disc or mount a different one.

FMGR-033

NOT ENOUGH ROOM ON CARTRIDGE

The cartridge does not have enough space on it. Try using another cartridge, decreasing the file size, or packing the disc.

FMGR-032

CARTRIDGE NOT FOUND

The specified cartridge cannot be found in the cartridge list.

FMGR-030

VALUE TOO LARGE FOR PARAMETER

1. The value supplied in the parameter is beyond the defined range.

2. If a parameter supplied in a calling sequence receives information returned from the routine, the parameter must be large enough. If the parameter supplied is a single word but the value to be returned is a double word, this error is reported.

FMGR-026

QUEUE FULL OR MAX PENDING SPOOLS EXCEEDED

The spool queue is full or the maximum number of spools pending has been exceeded. Re-run the job when the space becomes available.

FMGR-025

NO SPLCON ROOM

The SPLCON control-record area is full. This error may occur when the spool system is competing with programs using their own spooling file and running outside of batch. Re-run the job when SPLCON control-record entry space is available.

FMGR-024

NO MORE BATCH SWITCHES

The LU switch table is full; the size of the switch table specified at system generation is inadequate. Notify the system manager.

FMGR-023

NO AVAILABLE SPOOL FILES

All spool files are currently being used. Re-run the job when a spool file becomes available.

FMGR-022

NO AVAILABLE SPOOL LU'S

All spool logical units are currently being used. Re-run the job after a spool LU becomes available.

FMGR-021

ILLEGAL DESTINATION LU

The specified LU was not allocated by GASP. Try again using an LU allocated by GASP.

FMGR-020

ILLEGAL ACCESS LU

1. The logical unit number specified in the LU or cs command was not a positive logical unit number. Re-enter the corrected command.

2. There is an LU entry in the cartridge list that does not point to a disc device. After the disc was mounted, the LU command switched the device. Switch the LU back to its disc definition. If desired, dismount the disc. The LU can then be switched to a non-disc device.

FMGR-019

ILLEGAL ACCESS ON A SYSTEM DISC

The system manager is the only user with the capability to write on a system disc.

FMGR-018

ILLEGAL LU

The LU is either not assigned to the system, or is not defined in your session switch table (SST).

FMGR-017

ILLEGAL READ/WRITE ON TYPE 0 FILE

1. Some type 0 files do not support all read, write, or position requests. For example, a spool file may be open for read-only access; attempting to write to the file will fail. Check the file parameters or the namr.

2. There is not enough space on the cartridge to write to a spool file.

FMGR-016

ILLEGAL TYPE 0 OR SIZE=0

One of the following occurred: the wrong file type was specified; a type 0 file received a purge or create request; or the size specified was zero. Check the size and type parameters.

FMGR-015

ILLEGAL NAME

The file name does not conform to syntax rules; a common cause is the presence of an illegal character.

FMGR-014

DIRECTORY FULL

There is no more room in the file directory. Purge unused files and pack the disc, or try another cartridge.

FMGR-013

DISC LOCKED

A program has locked the disc. Initialize the cartridge if it was not initialized, or wait until it becomes unlocked.

FMGR-012

EOF OR SOF ERROR

A read, write, or position attempt was made to access a file beyond its boundaries. Check the record position parameters.

FMGR-011

DCB NOT OPEN

An unopened DCB was accessed before a create or open call, or the DCB was already closed when another close was attempted.

FMGR-010

NOT ENOUGH PARAMETERS

One or more of the required parameters were omitted. Enter all necessary parameters.

FMGR-009

ATTEMPT TO USE APOSN OR FORCE TO 1 A TYPE 0 FILE

A type 0 file cannot be positioned with APOSN or be forced to a type 1 file. Check the file type.

FMGR-008

FILE OPEN OR LOCK REJECTED

A file may not be opened if it is opened exclusively to another program, opened non-exclusively to 7 programs, or if its cartridge is locked. The CL or DL commands tell which program has the disc locked. If the cartridge is being packed, check to see if spooling is shut down.

FMGR-007

ILLEGAL SECURITY CODE OR ILLEGAL WRITE ON LU 2 OR 3

1. You specified a security different from the actual one, or you specified none and one is required.

2. You tried to write on LU 2 or 3, but only the system manager can write to files on these LUs.

FMGR-006

FILE NOT FOUND

The file cannot be found. Perhaps the file name or cartridge reference you typed was different from the actual one.

FMGR-005

RECORD LENGTH ILLEGAL

The record length specified is not valid. For a read or position attempt, the record may not have been written; on a write attempt, the record length for the data is illegal. The file may be corrupt.

FMGR-004

RECORD SIZE OF TYPE 2 FILE IS 0 OR UNDEFINED

A type 2 file create attempt was made without a record size or with a size of 0.

FMGR-003

BACKSPACE ILLEGAL

The device or type 0 file cannot be backspaced. Check its device type.

FMGR-002

DUPLICATE FILE NAME

A file already exists with the name specified. Repeat the command with a new name or purge the existing file.

FMGR-001

DISC ERROR

The disc is down. Try again and then report the problem to the system manager.

VMA/EMA Errors

VMA/EMA errors may cause program abortion. These errors have the same format as the MP and DM error returns, and look like this:

VMxx or EMxx

xx = either an FMP error number (xx is less than 80) or a VMA/EMA error (xx is greater than 80).

01

The disc is down. Try again, then report the problem to the system manager.

02

A VMA file already exists with the name specified. Change file name to a new name or purge the existing file.

05

1. You created a backing store file with less than 256 blocks of memory.

2. File extent cannot be created when read-only access has been specified for the VMA file. X-reg = the requested page ID that caused the problem.

06

You tried to access a VMA file that cannot be found. Check the file name or the cartridge reference.

07

1. An attempt was made to access a VMA file without specifying the security code or the wrong security code.

2. An attempt was made to write on LU 2 or 3; only the system manager can write to files on LU 2 or 3.

08

You attempted to open a VMA file that was already opened exclusively or was already opened to seven programs, or the cartridge containing the file is locked. Use the CL or DL command to locate the lock. If the cartridge is being packed, check to see if spooling is shut down.

12

File extent cannot be created when read only access has been specified to the VMA file. The X-Register contains the requested page that caused the problem.

13

The cartridge specified for the FMA file is locked. Initialize the cartridge if it was not initialized, otherwise keep trying.

14

There is no more room in the file directory. Purge any unused files and pack the disc if possible. Otherwise, try another cartridge. Probable cause is a directory full error when an attempt was made to extend VMA file. (VMA file may have up to 255 extents, thus another directory track may be necessary).

15

The file name does not conform to the syntax rules. Correct the name passed to the CREVM routine.

19

You attempted to write on a system disc; the system manager is the only user that has this capability.

20

An array is specified with incorrect subscripts. Check array declarations and array subscripts.

21

MSEG is not specified correctly in the $VMA or $EMA directive. Specify MSEG = (2*N)-1, where N is the maximum, number of VMA/EMA arrays used in a VIS instruction.

22

The program is not an VMA/EMA program.

32

You attempted to access a cartridge that cannot be found in the cartridge list. Check your cartridge reference.

33

You made an attempt to access a cartridge which does not have enough room. Try another cartridge, decrease the file size, or pack the disc.

40

LU was not found in Session Switch Table (SST).

46

Greater than 255 file extents on the VMA file. Make the VMA file size larger.

80

VMA/EMA system is corrupt. Self checks on the data structure did not pass a sanity check, so the VMA/EMA system aborted. Probable cause is by a program exceeding a non-EMA/VMA array boundary.

81

Program is not a VMA/EMA program, or bad request to VMAIO, EMAIO, or XLUEX. Re-load the program using the OP,VM or VM,number loader commands for a VMA program, or the OP,EM or EM, number loader commands for an EMA program.

82

The requested page is beyond the maximum page specified for VMA/EMA or the VMA disc file is not big enough. A VMA/EMA array boundary has been exceeded in your program. The X-reg equals the requested page number; the Y-reg is the logical address to map in the requested page; the abort address is the address of the instruction causing the problem.

83

All pages locked. This will occur when your working set is not large enough to support the size of MSEG that you require in your program. Increase your working set size with the WS,number command.

84

File type is not 2 or record length is not 1024 words. The file specified in the OPNVM or CREVM call was not type 2 with record length of 1024 words.

85

The VMA scratch file cannot be purged. This error should occur only when the scratch file to be used by VMA is in use by another program. See the system manager to correct the program.

86

Your program tried to access VMA after the VMA file has been closed. Revise your program.

87

MSEG is too small. This is an error from the .ESEG routine when the number of map registers specified is too large for the MSEG in the program space. The program must be revised to use less MSEG area or make the MSEG bigger. Also issued by .ESEG when zero pages are requested to be mapped.

88

Cannot re-specify the VMA file. Revise the program to call OPNVM or CREVM no more than once if the first call is successful.

89

Transfer too big for VMAIO, EMAIO, or XLUEX.

VM07

1. An attempt was made to access a VMA file without specifying the security code or with the wrong security code.

2. An attempt was made to write on LU 2 or 3; only the system manager can write to files on LU 2 or 3.

VM06

You tried to access a VMA file that cannot be found. Check the file name or the cartridge reference.

VM05

File extent cannot be created when read-only access has been specified for the VMA file. X-reg = the requested page ID that caused the problem.

VM02

A VMA file already exists with the name specified. Change file name to a new name or purge the existing file.

VM01

The disc is down. Try again, then report the problem to the system manager.

Appendix B
VMA/EMA Mapping Management Subroutines

This Appendix discusses available VMA mapping subroutines. These subroutines are used by HP compilers to access VMA data. The VMA/EMA mapping subroutines handle the mapping of VMA/EMA data into the program's logical address space. FORTRAN and Pascal programs do not need to make explicit calls to the mapping subroutines when accessing VMA/EMA data; the code calling the mapping subroutines is emitted automatically by the compilers. However, Macro/1000 programs must make calls to the mapping subroutines when accessing VMA/EMA data.

Table B-1 summarizes the VMA/EMA mapping subroutines described in this appendix. Firmware versions of the mapping subroutines exist for use on the HP 1000 E- and F-Series computers.

Note that the firmware versions of these subroutines should not be called from a program running in privileged mode (interrupt system off). To do so could cause the calling program to abort or destroy the integrity of the system.

As discussed in Chapter 4, in order to access VMA/EMA data it must be mapped into the program's logical address space. The physical pages in memory containing the VMA data are mapped by the VMA/EMA firmware routines into the two page virtual memory mapping segment (VSEG) of the program's logical address space. The VSEG is located in the last two pages of the user's logical map as shown in Figure B-1. The software mapping routines also use two pages to map data into logical memory as shown in Figure B-1. The two-page VSEG is used to map in contiguous pages of data from the disc, but will not necessary point to contiguous pages in physical memory.

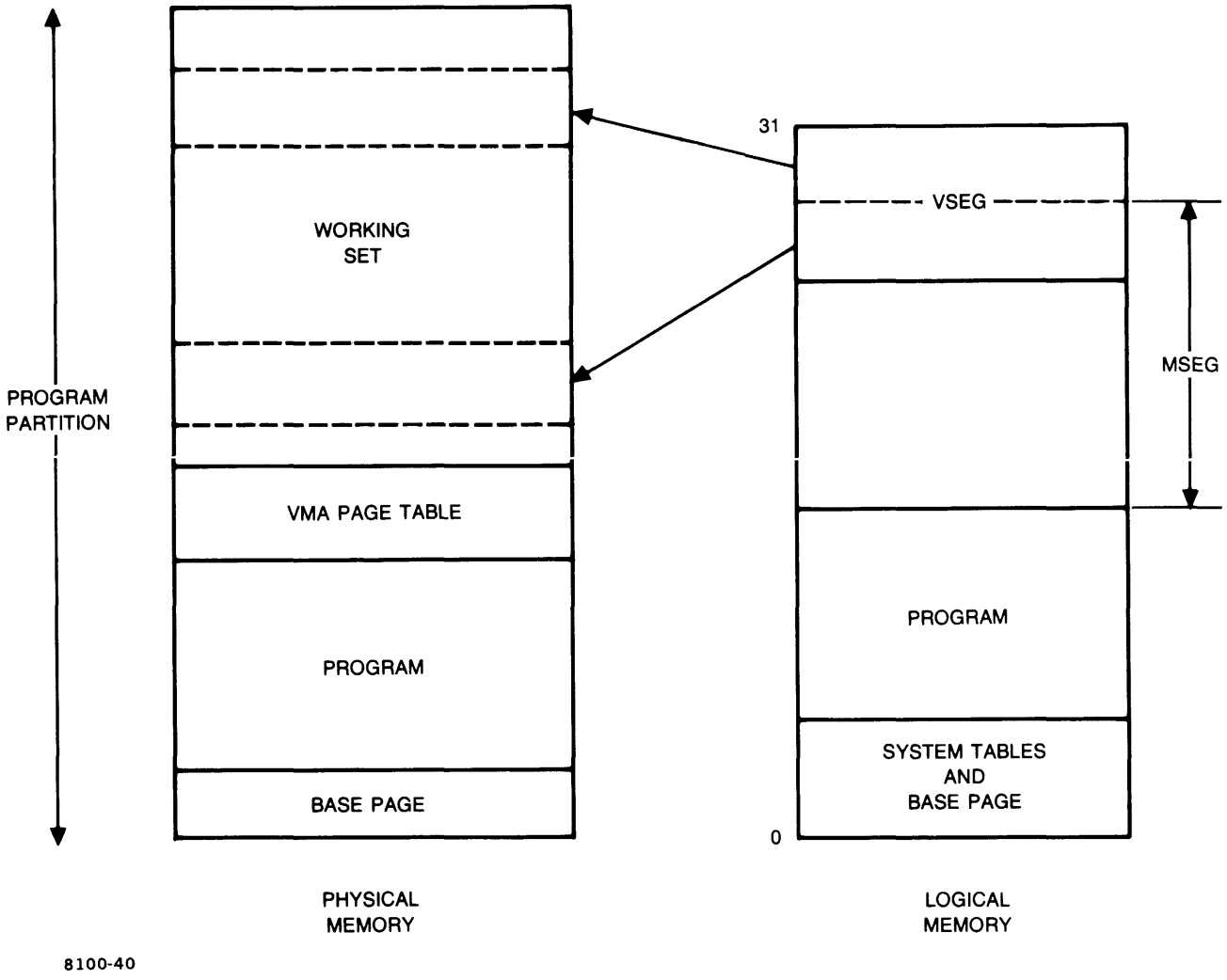


Figure B-1. VMA/EMA and Memory Structure

Table B-1. VMA/EMA Mapping Management Subroutines

SUBROUTINE	DESCRIPTION
.IMAP	Resolves address of an array element with one-word integer subscripts and maps it into logical memory.
.IRES	Resolves address of an array element with one-word integer subscripts (does not map).
.JMAP	Resolves address of an array element with double integer subscripts and maps it into logical memory.
.JRES	Resolves address of an array element with double integer subscripts (does not map).
MMAP	Maps consecutive pages of VMA/EMA into logical memory.
.ESEG	Maps several pages of VMA/EMA (not necessarily contiguous) into logical memory.
.LBP	Converts a virtual address to a logical address.
.LBPR	Converts a virtual address to a logical address.
.LPX	Converts a virtual address and an offset to a logical address.
.LPXR	Converts a virtual address and an offset to a logical address.
.EMIO	Maps in up to MSEG size buffer that can then be used for I/O. VMAIO and VREAD/VWRIT are the preferred routines for handling VMA/EMA I/O.

.IMAP Subroutine

The .IMAP subroutine resolves an address of an array element with a one-word integer subscript in a VMA or EMA array and maps the element into logical memory. .IMAP returns the logical address of the referenced element.

The .IMAP subroutine maps two pages of physical memory in the logical address space of the program into the VSEG. Refer to Chapter 4 for a description of MSEG and VSEG.

.IMAP maps in the page containing the element and the following page (if the following page is in the EMA or working set areas). This allows for multiword array elements. However, with the firmware version of .IMAP, if the element is in the last page of EMA or the working set, that physical page will be mapped through the second page of the VSEG.

The Macro/1000 calling sequence is:

```
      EXT .IMAP
      JSB .IMAP
      DEF TABLE       Address of table containing array parameters.
      DEF An          Address of nth subscript value.
      DEF An-1        Address of (n-1) subscript value.
        .
        .
        .
      DEF A2          Address of 2nd subscript value.
      DEF A1          Address of 1st subscript value.
  RTN normal return
```

NORMAL RETURN--On a normal return, the B-Register contains the logical address of the element referenced. The A-Register is the page number in VMA or EMA of the element.

Subroutine .IMAP aborts with an error (VM82) if the element address for an VMA or EMA variable does not fall within the VMA/EMA bounds. Other errors which may cause the program to abort are described in Appendix A.

TABLE--A table of array parameters containing the number of dimensions in the array; the number of elements in every dimension (upper bound-lower bound + 1); and the number of words per element.

For VMA/EMA arrays, a two-word offset value is required at the end of the table. The use of this offset enables several arrays to be defined in the same VMA/EMA. The offset is a double precision integer value with the high bits (bits 31-16) in offset word 1 and the low bits (bits 15-0) in word 2.

The lower bound of each dimension of the array is zero.

The number of words per element must be between 1 and 1024.

The content and structure of TABLE is as follows:

```
TABLE  DEC          Number of Dimensions
       DEC D(n-1)   Number of elements in the (n-1) dimension.
       DEC D(n-2)     .
        .             .
        .             .
        .             .
       DEC D(1)     Number of elements in the first dimension.
       DEC          Number of words per element.
       BSS 2        Offset word in double integer format (most
                     significant word first, least significant
                     word last).
```

The .IMAP subroutine assumes the array is stored in column-major order.

If the virtual address is a negative value (bit 31=1), then the lower word (bits 15-0) contains a logical address and the B-Register will be returned by .IMAP as the logical address with indirection resolved.

.IRES Subroutine

The .IRES subroutine resolves an address of an array element with a one-word integer subscript in a VMA or EMA array. The .IRES subroutine is similar to the .IMAP subroutine except the element is not mapped into logical memory.

The Macro/1000 calling sequence is:

```
      EXT .IRES
      JSB .IRES
      DEF TABLE       Address of table containing array parameters.
      DEF An          Address of nth subscript value.
      DEF An-1        Address of (n-1) subscript value.
       :
      DEF A2          Address of 2nd subscript value.
      DEF A1          Address of 1st subscript value.
  RTN normal return
```

NORMAL RETURN---On a normal return, the A- and B-Registers contain the offset of the array element into the EMA or working set in double integer format (most significant word in the A-Register, least significant word in the B-Register).

The TABLE for .IRES has the same contents and structure as the TABLE for .IMAP.

If the virtual address is a negative value (bit 31=1) then the lower word (bits 15-0) contains a logical address and the B-Register will be returned by .IRES as the logical address with indirection resolved.

Any error detected causes the program to be aborted. Possible errors are described in Appendix A.

.JMAP Subroutine

The .JMAP subroutine resolves an address of an array element with a double-integer subscript in either a VMA or EMA array and maps the element into logical memory. .JMAP sets up the last two user logical map registers (VSEG) and the B-Register to access the required array element. The .JMAP subroutine is similar in function to the .IMAP subroutine. Any error detected causes the program to abort. Possible errors described in Appendix A.

The Macro/1000 calling sequence is:

```
        EXT .JMAP
        JSB .JMAP
        DEF TABLE           Address of the array description table.
        DEF An              Address of nth subscript value.
        DEF A(n-1)          Address of (n-1) subscript value.
         :
        DEF A1              Address of 1st subscript value.
    RTN normal return
```

NORMAL RETURN--On a normal return, the VMA or EMA array element resides in physical memory, the last two user map registers (VSEG) point to that element, and the B-Register contains the logical address of the element. The A-Register contains the page number of the element in physical memory.

TABLE--The table of array parameters is the same as .IMAP, except it has the following structure:

```
TABLE DEC         Number of dimensions.
      DEC D(n-1)  Number of elements in (n-1) dimension (High Bits)
      DEC D(n-1)  Number of elements in (n-1) dimension (Low Bits)
       :                    :
      DEC D(1)    Number of elements in first dimension (High Bits)
      DEC D(1)    Number of elements in first dimension (Low Bits)
      DEC         Number of words per element.
      BSS 2       Offset word in double integer format.
```

If the VMA array element does not exist in physical memory and the working set is full, the VMA/EMA firmware is called to map a page out of the working set and move into physical memory the desired virtual memory page (and the following page). The data in memory can then be managed via the VMA/EMA subroutines.

If the virtual address is a negative value (bit 31=1), then the lower word (bits 15-0) contains a logical address and the B-Register will be returned by .JMAP as the logical address with indirection resolved.

.JRES Subroutine

The .JRES subroutine resolves an address of an array element with a double-integer subscript in either a VMA or EMA array. The .JRES subroutine is similar to the .JMAP subroutine except the element is not mapped into logical memory.

The Macro/1000 calling sequence is:

```
     EXT .JRES
     JSB .JRES
     DEF TABLE        Address of table containing array parameters.
     DEF An           Address of nth subscript value.
     DEF An-1         Address of (n-1) subscript value.
      .                        .
      .                        .
      .                        .
     DEF A2           Address of 2nd subscript value.
     DEF A1           Address of 1st subscript value.
 RTN normal return
```

NORMAL RETURN---On a normal return, the A- and B-Registers contain the offset of the array element into the VMA or EMA in double integer format (most significant word in the A-Register, least significant word in the B-Register).

The TABLE for .JRES has the same contents and structure as the TABLE for .JMAP.

If the virtual address is a negative value (bit 31=1), then the lower word (bits 15-0) contains a logical address and the B-Register will be returned by .JRES as the logical address with indirection resolved.

MMAP Subroutine

MMAP is a subroutine that maps a sequence of VMA/EMA pages into the mapping segment area of the logical address space of a program. It is callable from both Macro/1000, FORTRAN, and Pascal/1000 programs. MMAP starts mapping at the starting logical page of the mapping segment area (bits 11-15 in the second word of the ID extension). The maximum number of pages that may be requested with MMAP is 31 minus the starting logical page of MSEG. If the working set size is less than this value, then the working set size becomes the maximum number of pages that can be mapped.

MMAP will map one extra page than the number of pages requested. This is done in order to prevent dynamic mapping system (DMS) errors in case a multiple word element or a buffer for in I/O transfer crosses the end of the last requested page to be mapped. If the extra page mapped in by MMAP is the last+1 page of VMA/EMA, then the page mapped in is read/write protected. If the number of pages to be mapped is zero, MMAP will map in one page.

The user cannot assume that previous mapped in data is still in logical memory if calls to other mapping routines have been made after the call to MMAP.

The RTE FORTRAN calling sequence is:

```
CALL MMAP(IPGS,NPGS)
```

where:

IPGS starting page to map (where the first page in EMA or working set is page 0.

NPGS Number of pages to be mapped.

The Macro/1000 calling sequence is:

```
    EXT MMAP
    JSB MMAP
    DEF RTN
    DEF IPGS
    DEF NPGS
RTN return point
```

Upon return:

A-Register = 0 if normal return

= -1 if an error occurred.

MMAP returns an error under any of the following conditions:

1. IPGS or NPGS is negative.

2. NPGS is greater than the maximum number of mappable pages (described above).

3. All NPGS to be mapped do not fall within VMA/EMA bounds.

4. EMA was not declared in the calling program.

5. Last page of requested NPGS if past the end of VMA/EMA.

The Pascal/1000 calling sequence (for MMAP) can be derived from the EXEC Procedure Call Format previously described in Chapter 2.

.ESEG Subroutine

The .ESEG subroutine will map several pages of VMA/EMA (not necessarily contiguous) into the mapping segment area of the logical address space of a program.

The Macro/1000 calling sequence is:

```
      EXT .ESEG
       .
       .
       .
      LDB <number>    Number of map registers to modify.
      JSB .ESEG
      DEF *+2         Error return point (not used).
      DEF PBUFR       Table of pages to map.
  RTN error return    (Not used.)
      normal return
```

The table of pages to map, PBUFR, is defined as follows:

```
PBUFR DEC <1st page>  First VMA/EMA page to map.
      DEC <2nd page>  Second VMA/EMA page to map.
       .                .
       .                .
       .                .
      DEC <last page> Last page to map.
```

NORMAL RETURN---Upon successful return, all the VMA/EMA pages will be mapped into logical memory and the B-Register will be equal to the logical address of the starting page of MSEG. Any error will cause the program to abort.

The maximum number of pages that can be mapped is 31 pages minus the starting logical page of MSEG.

.LBP, .LBPR Subroutine

The .LBP and .LBPR subroutines convert a virtual address to a logical address mapping the word pointed to into logical memory.

The Macro/1000 calling sequence is:

```
      EXT .LBP                        EXT .LBPR
      DLD PONTR        or             JSB .LBPR
      JSB .LBP                        DEF PONTR
```

where:

PONTR Double integer pointer containing the virtual address.

NORMAL RETURN---Upon a normal return the B-Register contains the logical address, and the A-Register contains the page number in physical memory of the data.

If the PONTR is a negative value (bit 31=1), then the lower word (bits 15-0) contains a logical address, and the B-Register is returned as this logical address with any indirection resolved.

.LPX, .LPXR Subroutine

The .LPX and .LPXR subroutines convert a virtual address plus an offset to a logical address mapping the word pointed to into logical memory.

The Macro/1000 calling sequence is:

```
      EXT .LPX                        EXT .LPXR
      DLD PONTR        or             JSB .LPXR
      JSB .LPX                        DEF PONTR
      DEF OFSET                       DEF OFSET
```

where:

PONTR Double integer pointer containing the virtual address.

OFSET Double integer offset from the virtual address.

NORMAL RETURN---Upon a normal return, the B-Register contains the logical address and the A-Register contains the page number in physical memory of the data.

If the PONTR plus the OFSET is a negative value (bit 31=1), then the lower word (bits 15-0) contains a logical address, and the B-Register is returned as this logical address with any indirection resolved.

.EMIO Subroutine

The .EMIO subroutine is available for compatibility purposes with RTE-IVA and RTE-IVB EMA programs. The preferred method of doing I/O to VMA/EMA in RTE-6/VM is with the VMAIO or VREAD/VWRIT routines.

Subroutine .EMIO is a subroutine used only in a VMA/EMA environment to ensure that a buffer to be accessed is entirely within the logical address space of the program. It will call MMAP (if appropriate) to alter the logical address space to contain the buffer, or if this is impossible it will return with an error.

The buffer length plus the offset between the start of the buffer and its page boundary must be less than or equal to the mapping segment size+1 (in words). To ensure this, it is recommended that the buffer length be less than or equal to (MSEG size) pages. If the buffer length is larger, the VMAIO subroutine should be called to perform the I/O transfer.

Subroutine .EMIO maps the special mapping segment if necessary and returns with the logical address of the start of the buffer in the B-Register.

The calling sequence is:

```
     EXT .EMIO
     JSB .EMIO
     DEF RTN          address for error-return
     DEF BUFL         number of words in the buffer
     DEF TABLE        table containing array parameters
     DEF An           subscript value for nth dimension
     DEF An-1         subscript value for (n-1) dimension
       .
       .
       .
     DEF A2           subscript value for 2nd dimension
     DEF A1           subscript value for 1st dimension
RTN  error return
     normal return
```

The content and structure of TABLE is as follows:

```
Number of Dimensions

     -L(n)
      d(n-1)
     -L(n-1)
      d(n-2)
        :
     -L(2)
      d(1)
     -L(1)
     Number of words per element
     Offset word 1 (bits 15-0)
     Offset word 2 (bits 31-16)
```

where:

L(i) is the lower bound of the ith dimension.

d(i) is the number of elements in the ith dimension.

ERROR RETURN--.EMIO makes an error return at location RTN with the A-Register containing "16" (ASCII) and the B-Register containing "EM" (ASCII). If the relocatable subroutine ERR0 is called to handle the error, the following message is sent to LU 6:

```
name 16-EM @ address
```

where:

name = the name of the program.

address = the location from which ERR0 was called.

Subroutine .EMIO makes an error return under any of the following conditions:

1. The buffer length is negative.

2. An EMA is not declared in the calling program.

3. A subscript is negative.

4. The buffer length plus the page offset of the start of the buffer is greater than the mapping segment size+1 (in words).

NORMAL RETURN---When .EMIO makes a normal return, the B-Register contains the logical address of the element. The contents of the A-Register is meaningless.

Appendix C
System Communication Area and System Tables

This appendix contains information about the following topics:

* SYSTEM COMMUNICATIONS AREA - Base page locations of area used for system communications.
* PROGRAM ID SEGMENT MAP - Format of ID segments kept in system area for user programs, ID segment extension, and short ID segments.
* DISC LAYOUT - Allocation of disc space for an RTE-6/VM system.
* TABLE AREA I AND II ENTRY POINTS

Other system tables relating to I/O considerations, such as the Equipment Table, Device Reference Table and Driver Mapping Table are described in Appendix E, "I/O TABLES AND PROCESSING".

System Communication Area

This area is a block of storage in the system base page, starting at location 1645, that is used by RTE-6/VM to define request parameters, I/O tables, scheduling lists, operating parameters, memory bounds, etc. The Macroassembler allows relocatable programs to reference this area by absolute addresses 1645 through 1777 octal. User programs can read information from this area but cannot alter it because of the memory protect feature.

The contents and description of each location in this area are listed in Table C-1.

Table C-1. System Communications Area Locations

OCTAL LOCATION	CONTENTS	DESCRIPTION
SYSTEM TABLE DEFINITION		
01645	XIDEX	Address of current program's ID extension
01646	XMATA	Address of current program's MAT entry
01647	XI	Address of index register save area
01650	EQTA	FWA of Equipment Table
01651	EQT#	Number of EQT entries
01652	DRT	FWA of Device Reference Table, word 1
01653	LUMAX	Number of logical units in DRT
01654	INTBA	FWA of Interrupt Table
01655	INTLG	Number of Interrupt Table Entries
01656	TAT	FWA of Track Assignment Table
01657	KEYWD	FWA of keyword block
I/O MODULE/DRIVER COMMUNICATION		
01660	EQT1	Addresses of first 11 words of current EQT entry (see 01771 for last four words)
01661	EQT2	
01662	EQT3	
01663	EQT4	
01664	EQT5	
01665	EQT6	
01666	EQT7	
01667	EQT8	
01670	EQT9	
01671	EQT10	
01672	EQT11	
01673	CHAN	Current DCPC channel number
01674	TBG	I/O address of time-base card
01675	SYSTY	EQT entry address of system TTY
SYSTEM REQUEST PROCESSOR/EXEC COMMUNICATION		
01676	RQCNT	Number of request parameters −1
01677	RQRTN	Return point address
01700	RQP1	Addresses of request parameters (set for a maximum of nine parameters)
01701	RQP2	
01702	RQP3	
01703	RQP4	
01704	RQP5	
01705	RQP6	
01706	RQP7	
01707	RQP8	
01710	RQP9	

Table C-1. System Communications Area Locations (continued)

OCTAL LOCATION	CONTENTS	DESCRIPTION
SYSTEM LISTS ADDRESSES		
01711	SKEDD	Schedule list
01712	PVCN	Privileged nest counter
01713	SUSP2	Wait Suspend list
01714	SUSP3	Available Memory list
01715	SUSP4	Disc Allocation list
01716	SUSP5	Operator Suspend list
PROGRAM ID SEGMENT DEFINITION		
01717	XEQT	ID segment address of current program
01720	XLINK	Linkage
01721	XTEMP	Temporary (five words)
01726	XPRIO	Priority word
01727	XPENT	Primary entry point
01730	XSUSP	Point of suspension
01731	XA	A-register at suspension
01732	XB	B-register at suspension
01733	XEO	E and overflow register suspension
SYSTEM MODULE COMMUNICATION FLAGS		
01734	OPATN	Operator/keyboard attention flag
01735	OPFLG	Operator communication flag
01736	SWAP	RT disc resident swapping flag
01737	DUMMY	I/O address of dummy interface flag
01740		Reserved
01741		Reserved
MEMORY ALLOCATION BASES DEFINITION		
01742	BPA1	FWA user base page link area
01743	BPA2	LWA user base page link area
01744	BPA3	FWA user base page link
01745	LBORG	FWA of resident library area
01746	RTORG	FWA of real-time COMMON
01747	RTCOM	Length of real-time COMMON
01750D	RTDRA	FWA of real-time partition
01751D	AVMEM	LWA +1 of real-time partition
01752	BGORG	FWA of background COMMON
01753	BGCOM	Length of background COMMON
01754D	BGDRA	FWA of background partition

Table C-1. System Communication Area Locations (continued)

OCTAL LOCATION	CONTENTS	DESCRIPTION
UTILITY PARAMETERS		
01755	TATLG	Negative length of track assignment table
01756	TATSD	Number of tracks on system disc
01757	SECT2	Number of sectors/track on LU2 (system)
01760	SECT3	Number of sectors/track on LU3 (aux.)
01761	DSCLB	Disc address of library entry points
01762	DSCLN	Number of user available library entry points.
01763	DSCUT	Disc address of relocatable disc resident library.
01764	SYSLN	Number of system library entry points
01765	LGOTK	LGO: LU#, starting track, number of tracks (same format as ID segment word 28)
01766	LGOC	Current LGO track/sector address (same format as ID segment word 26)
01767	SFCUN	LS: LU# and disc address (same format as ID segment word 26)
01770	MPTFL	Memory protect ON/OFF flag (0/1)
01771	EQT12	Address of last four words of current EQT
01772	EQT13	
01773	EQT14	
01774	EQT15	
01775D	FENCE	Memory protect fence address
01776		Reserved
01777	BGLWA	LWA memory background partition

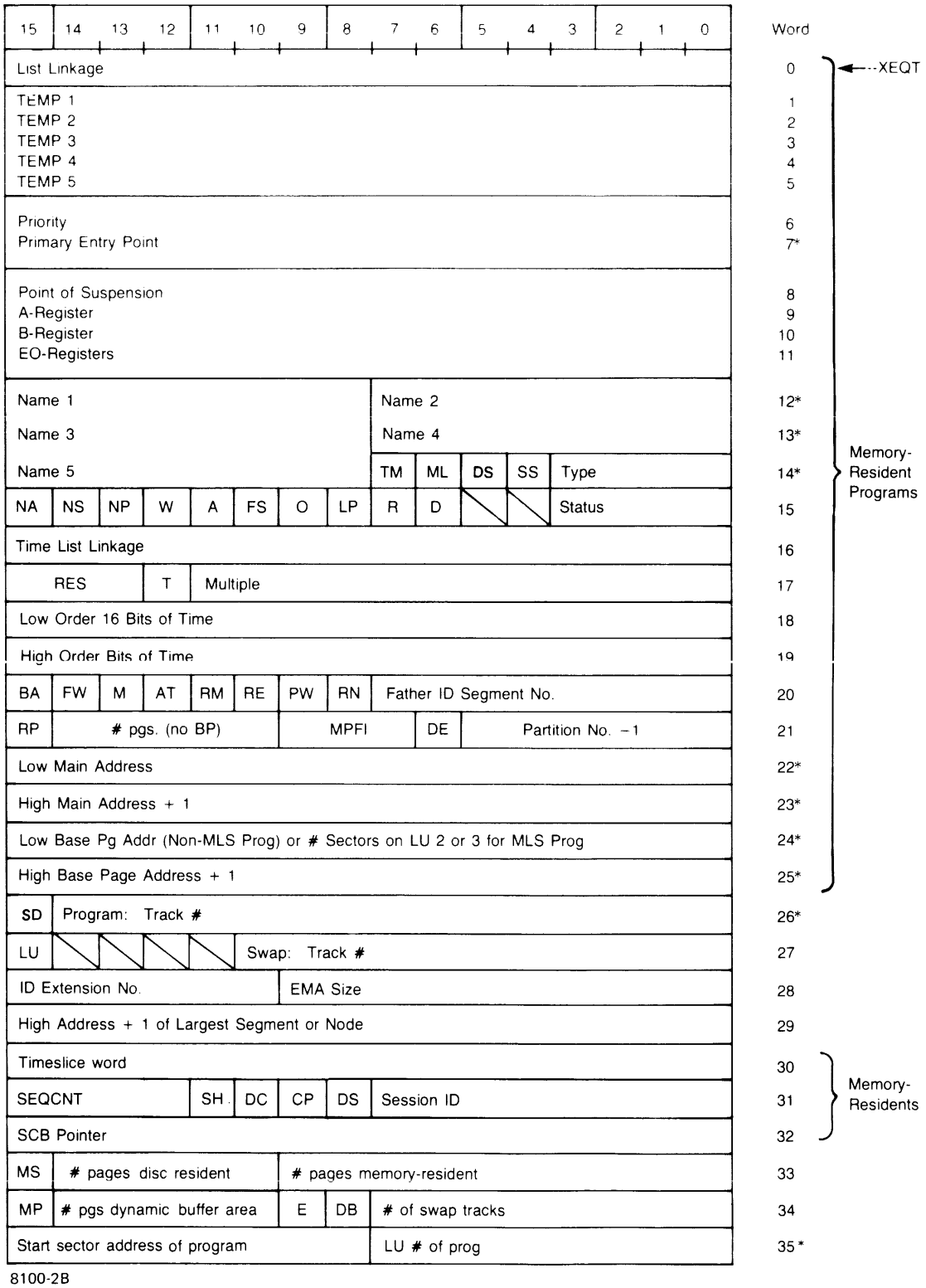
D letter indicates the contents of the location are set dynamically by the dispatcher.

Program ID Segment

Each user program has a 36-word ID segment located in Table Area II that contains static and dynamic information defining the properties of the program. The static information is set during generation time or when the program is loaded on-line. The dynamic information is maintained by the operating system Executive.

The number of ID segments contained in a system is established during system generation, and is directly related to the number of programs that can be in main memory at any given time. If all the ID segments are in use, no more programs can be added on-line unless some other existing program is first "offed" (removed from the system) to recover an ID segment.

The format of the ID segment is illustrated in Figure C-1. Each ID segments's address is located in the Keyword Table (see location 01657).



8100-2B

Figure C-1. ID Segment Format

where:

* = words used in short ID segments for program segments

TM = temporary load (copy of ID segment is not on the disc)

ML = memory lock (program may not be swapped)

TS = the program is transportable

SS = short segment (indicates a nine-word segment)

Type = specified program type (1-5)

NS = the no-suspend bit is set

NA = no abort (instead, pass abort errors to program)

NP = no parameters allowed on reschedule

W = wait bit (waiting for program whose ID segment address is in word 2)

A = abort on next list entry for this program

FS = file system bit (this program has files open)

O = operator suspend on next schedule attempt

LP = load in progress; program is being dispatched from disc.

R = resource save (save resources when setting dormant)

D = dormant bit (set dormant on next schedule attempt)

Status = current program status

T = time list entry bit (program is in the time list)

BA = batch (program is running under batch)

FW = father is waiting (father scheduled with wait)

M = Multi-Terminal Monitor bit

AT = attention bit (operator has requested attention)

RM = reentrant memory must be moved before dispatching program

RE = reentrant routine now has control

PW = program wait (some other program wants to schedule this one)

RN = Resource Number either owned or locked by this program

RP = reserved partition (only for programs that request it)

MPFI = memory protect fence index

DE = defer the EXEC 6 (terminate program) request

LU = 0 if LU2, 1 if LU3

SD = clear this ID segment when the program terminates.

TIMESLICE WORD (30):

The timeslice word defines the timeslicing status of a program. This word is defined as follows:

1 = This program has just been rescheduled or is not timesliced.

0 = This program has used a full timeslice or program is not scheduled.

<0 = This program was running (under timeslice control) and was "bumped" from execution by a higher priority program. This word represents the remaining timeslice for this program.

OPEN FLAG WORD(31):

SEQCNT = sequence counter. Each time a program is aborted or aborted or terminates (unless saving resources) the counter is incremented. The counter value is used to build FMP open Flags.

SH = the program is using shareable EMA.

DC = do not copy flag. Set by the generator (if 128 is added to program type) or the loader (using Don't Copy op-code).

CP = copy flag. Indicates that the program is a copy.

DS = used by the distributed systems software.

Session ID = System LU of terminal that program was loaded from. from. For programs permanently loaded or temporarily loaded by the system manager, a zero is shown here.

SESSION WORD(32):

The session word identifies the user of a program.

A negative value represents the logical unit number of the terminal from which the program was invoked (not under session).

A positive value represents the address of the SST length word of the session control block for the session currently using this program (under session).

Programs scheduled by interrupt will have a zero in this word.

MS = the program is an MLS (multilevel segmentation) program

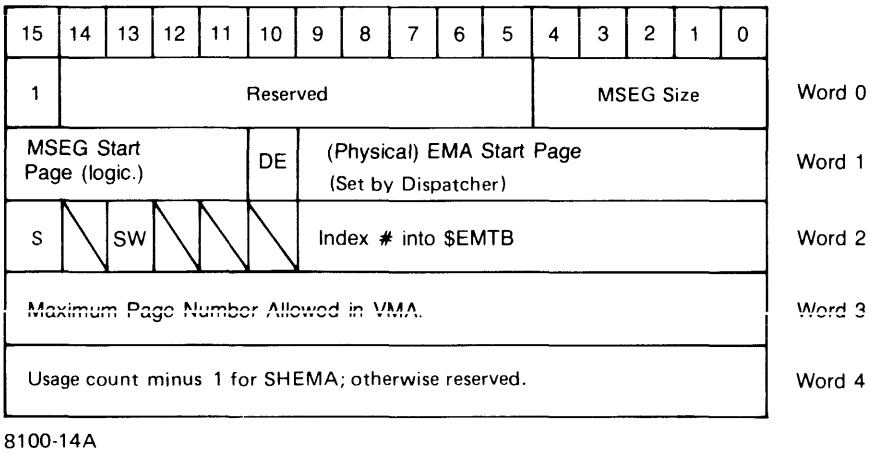
MP = the program is using modified maps for I/O

E = the program needs an EXEC 4 allocation request

DB = debug it.

ID Segment Extensions

Each EMA program requires a 5-word ID segment extension in addition to its 36-word ID segment. The number of ID extensions contained in the system is also set at generation time, and if all are in use, no more EMA programs can be added on-line. The format of the ID segment is illustrated in Figure C-2. SHEMA programs can share ID segment extensions.



DE = if 1, indicates default EMA size was used; otherwise, size was set by user

SW = if 1, indicates page translation table has been initialized.

S = if 1, indicates that this is a SHEMA program

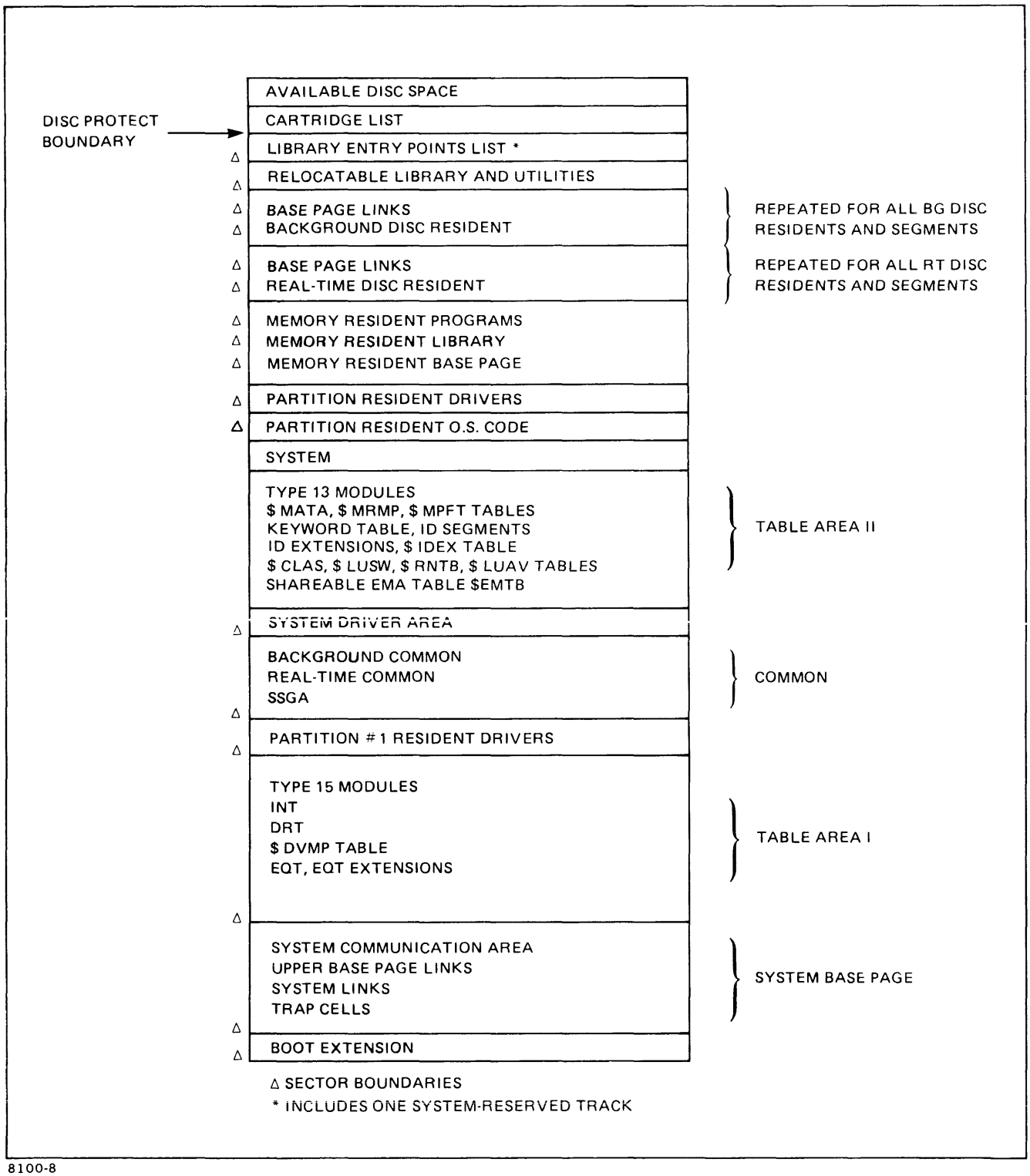
Figure C-2. ID Segment Extension

Short ID Segments

Short ID segments requiring nine words are used only for program segments. A short ID segment is required for each segment of a segmented program. If no empty short ID segments are available during an on-line load, a standard 36-word ID segment will be used. The information contained in a short ID segment is illustrated in Figure C-1.

RTE-6/VM System Disc Layout

Figure C-3 illustrates how disc space is allocated when a RTE-6/VM system is generated.



8100-8

Figure C-3. RTE-6/VM System Disc Layout

Appendix D
Program Related Tables

This appendix contains information on the following:

Program Types

Program States

Table D-1. Summary of RTE-6/VM Program Types

PROGRAM CATEGORY / EXECUTABLE PROGRAMS	PROGRAM TYPE	SYSTEM COMMON ACCESS: REAL-TIME COMMON	BACKGROUND COMMON	SSGA	RT COMMON & SSGA	BG COMMON & SSGA	EMA ALLOWED	LOAD POINT: NO COMMON DECLARED	SOME COMMON DECLARED	MEMORY PROTECT FENCE: NO COMMON DECLARED	SOME COMMON DECLARED
MEMORY RESIDENT*	1	√						L_1	L_1	F_5	F_3
	9		√					L_1	L_1	F_5	F_4
	17			√				L_1	L_1	F_1	F_1
	17				√			L_1	L_1	F_1	F_1
	25					√		L_1	L_1	F_1	F_1
REAL TIME DISC RESIDENT*	2	√					√	L_4	L_4	F_6	F_3
	10		√				√	L_4	L_4	F_6	F_4
	18			√			√	L_4	L_4	F_1	F_1
	18				√		√	L_4	L_4	F_1	F_1
	26					√	√	L_4	L_4	F_1	F_1
BACKGROUND DISC RESIDENT*††	3		√				√	L_4	L_4	F_6	F_4
	11	√					√	L_4	L_4	F_6	F_3
	19			√			√	L_4	L_4	F_1	F_1
	19					√	√	L_4	L_4	F_1	F_1
	27				√		√	L_4	L_4	F_1	F_1
LARGE BACKGROUND DISC RESIDENT WITHOUT TABLE AREA II ACCESS*††	4		√				√	L_3	L_2	F_2	F_4
	12	√					√	L_3	L_2	F_2	F_3
	20			√			√	L_2	L_2	F_1	F_1
	20					√	√	L_2	L_2	F_1	F_1
	28				√		√	L_2	L_2	F_1	F_1
EXTENDED BACKGROUND DISC RESIDENT WITHOUT TABLE I, II, COMMON, OR DRIVER ACCESS	6						√	L_5	N/A	F_7	N/A

*ADD 80 TO ANY OF THESE TYPES TO SPECIFY AUTOMATIC SCHEDULING AT SYSTEM STARTUP.

††ADD 128 TO ANY OF THESE TYPES TO SPECIFY THAT THE PROGRAM CANNOT BE DUPLICATED.

Table D-1. Summary of RTE-6/VM Program Types (Cont.)

SPECIAL PROGRAMS	TYPE	DESCRIPTION
SYSTEM MODULE	0	MODULE TO BE LOADED WITH RESIDENT SYSTEM. PART OF HP-SUPPLIED SYSTEM, USER-WRITTEN DRIVER, ETC.
PROGRAM SEGMENT	5	OVERLAYABLE MODULE USED WITH DISC RESIDENT MAIN. COMMON TYPE, MEMORY-PROTECT FENCE ADDR. AND LOAD PT. DETERMINED BY MAIN.
PROGRAM	6	EXTENDED BACKGROUND PROGRAMS CANNOT BE GENERATED INTO THE SYSTEM. TYPE 6 PROGRAMS CONNOT ACCESS SYSTEM COMMON.
SUBROUTINE	6	RELOCATED INTO RESIDENT LIBRARY IF CALLED BY ANY MEMORY RESIDENT PROGRAM (ALWAYS BECOME 7'S) AT GEN TIME.
SUBROUTINE	7	STORED ON DISC IN RELOCATABLE FORM. ANY PROGRAM CALLING A TYPE 7 HAS A COPY APPENDED TO IT.
SUBROUTINE	8	APPENDED TO CALLING PROGRAM. ALL TYPE 8 RELOCATABLES ARE DISCARDED AFTER GENERATION.
TABLE AREA II	13	MODULE TO BE LOADED WITH RESIDENT SYSTEM IN TABLE AREA II. PART OF HP-SUPPLIED SYSTEM, USER-WRITTEN TABLES, ETC.
SUBROUTINE	14	RELOCATED INTO RESIDENT LIBRARY, WHETHER CALLED OR NOT (ALWAYS BECOME TYPE 7).
TABLE AREA I	15	MODULE TO BE LOADED WITH RESIDENT SYSTEM IN TABLE AREA I. PART OF HP-SUPPLIED SYSTEM, USER-WRITTEN TABLES, ETC.
SSGA MODULE	30	RELOCATED INTO SUBSYSTEM GLOBAL AREA OF SYSTEM. ACCESSIBLE ONLY TO PROGRAMS OF PROPER TYPE (ABOVE)

LOAD POINT & FENCE DEFINITIONS

L_1 – NEXT AVAILABLE LOCATION DURING LOAD OF RESIDENTS PLUS 2

L_2 – 11TH WORD OF NEXT PAGE AFTER COMMON AREAS

L_3 – 11TH WORD OF NEXT PAGE AFTER DRIVER PARTITION

L_4 – 11TH WORD OF NEXT PAGE AFTER TABLE AREA II

L_5 – 11TH WORD OF NEXT PAGE AFTER BASE PAGE.

F_1 – FIRST WORD OF SSGA

F_2 – FIRST WORD OF PAGE FOLLOWING DRIVER PARTITION

F_3 – FIRST WORD OF RT COMMON

F_4 – FIRST WORD OF BG COMMON

F_5 – FIRST WORD OF RESIDENT PROGRAM AREA

F_6 – FIRST WORD OF PAGE FOLLOWING TABLE AREA II

F_7 – FIRST WORD OF PAGE FOLLOWING BASE PAGE.

Program States

With the RTE-6/VM operating environment, programs can exist in 7 states. The state of a program can be changed by the operator (RTE-6/VM operator command), by another program (EXEC call), or by the system to reflect an environmental change, i.e., program requested memory or disc space that was not available, requested I/O on a down device, etc. At any given time, the state of a program indicates its relationship to the RTE-6/VM operating environment.

State 0 - Dormant. This state indicates that a program is not scheduled to execute. A program can be in this state if it has never been scheduled or if it has been placed in this state from a previous state by an operator command or by another program via an EXEC call. A program can be dormant if it has been terminated serially reusable.

State 1 - Scheduled. This state indicates that a program is scheduled to execute; it has been placed in the scheduled list.

State 2 - I/O Suspended. This state indicates that a program has requested I/O servicing and the system is currently performing the I/O operation or the request is queued to be processed. This condition occurs with any input operation, or with an output operation to an unbuffered device. If the class I/O technique discussed in Chapter 2 is used, or if the output device is buffered, the program will not enter this state and will be allowed to continue executing while the I/O operation is being performed.

State 3 - General Wait. A program is placed in this state if it has requested system resources that are temporarily unavailable or services that temporarily cannot be performed. For example, a program would be placed in this state if it were waiting for a data buffer to be supplied by another program, or if it requested I/O on a device that was allocated and locked to another program.

State 4 - Memory Suspended. A program is placed in this state if it has requested an operation that requires the use of System Available Memory (SAM) and an insufficient amount of SAM is available. This is a temporary state and when enough SAM becomes available, the program will be placed back into the scheduled list.

State 5 - Disc Suspended. A program is placed in this state if it has requested disc space that is unavailable. This is a temporary condition and when disc space becomes available the program will be placed back into the scheduled list.

State 6 - Operator Suspended. A program is placed into this state by an operator command or by an EXEC call from within a program. A GO operator command is necessary to remove the program from this state.

Figure D-1 shows the various states a program can exist in and the "transition paths" between each state.

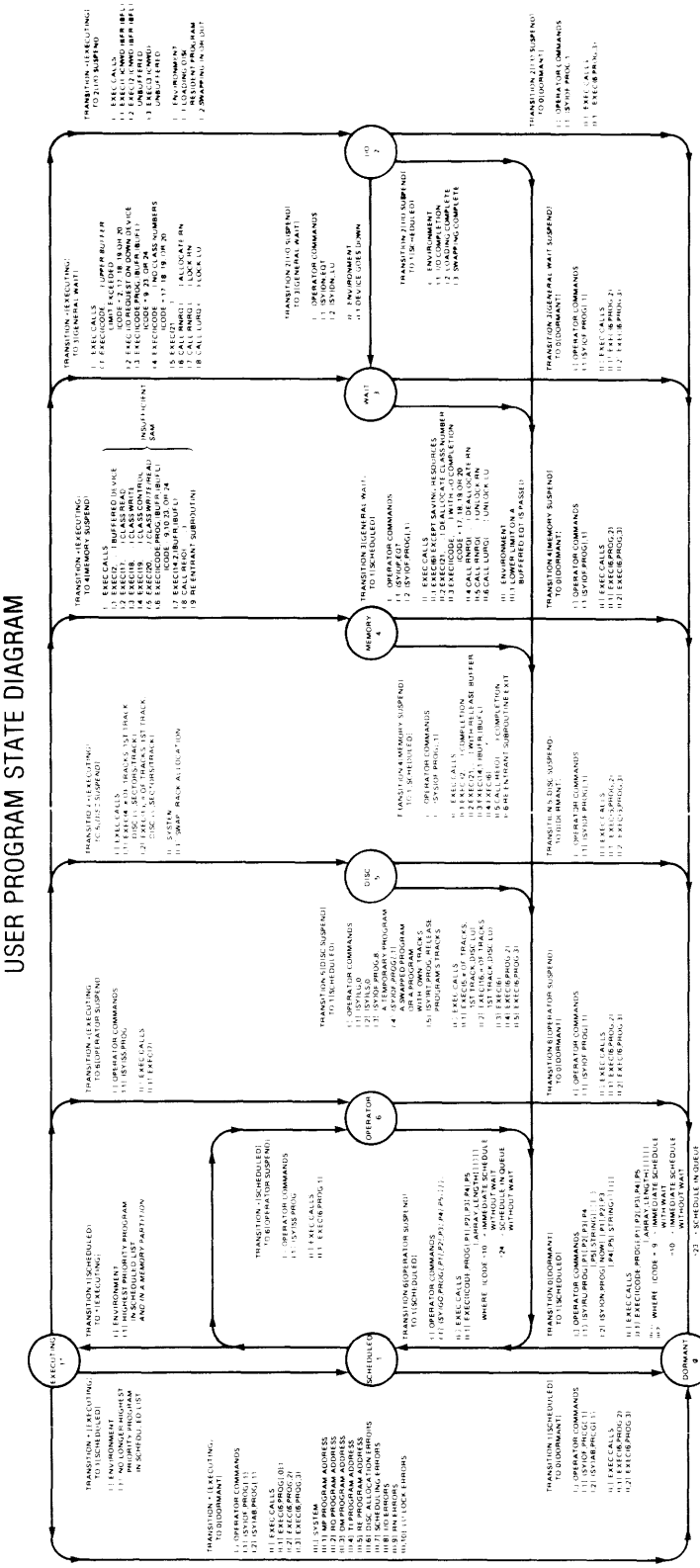


Figure D-1. User Program State Diagram

Appendix E
I/O Tables and Processing

This Appendix contains information on the following topics:

* Equipment Table (EQT) - EQT entry format
* Device Reference Table (DRT) - DRT table format and entry format.
* Driver Mapping Table (DMT) - DMT use and entry format.
* Interrupt Table And Trap Cells - Uses and contents.
* Power Fail/Auto Restart - Function and general flow.
* Standard I/O Request Flow - General flow of standard I/O request and general flow diagram.

For more information on I/O processing, refer to the Driver Writing Manual.

Equipment Table (EQT)

The Equipment Table (EQT) maintains a list of all the I/O equipment in the system. The table consists of 15-word entries, with one entry for each I/O controller defined in the system at generation time. Each EQT entry contains all of the information required by the system and associated driver to operate the device, including:

* I/O select code in which the controller is interfaced with the computer.
* Driver type.
* Various driver or controller requirements and specifications.

Some information contained in the EQT entry is static, i.e., fixed at generation time or I/O reconfiguration and not changed during on-line operation. Other information is dynamic and can be changed on-line or is modified by the system to reflect various I/O conditions. Word 1 of the EQT entry contains a pointer to the linked list of I/O request buffers pending on the EQT (see Figure E-1 and E-2 below).

All Equipment Table entries are located sequentially in Table Area I, beginning with EQT entry number 1. The address of the first entry and the total number of entries in the table can be found in the System Base Page Communications area, location 1650B and 1651B, respectively.

EQT entry words 14 and 15 in the EQT entry for each I/O controller function as a controller time-out clock. EQT entry word 15 is the actual working clock. Before each I/O transfer is initiated, it is set to a value m, where m is a negative number of 10 ms time intervals stored in EQT entry word 14. If the controller does not interrupt within the required time interval, it is to be considered as having "timed out". The EQT 15 clock word for each controller can be individually set by the three methods:

1. The system inserts the contents of EQT entry word 14 into EQT entry word 15 before a driver (initiation or completion section) is entered. EQT entry word 14 can be reset to m by entering (T=) at generation time (refer to the RTE-6/VM On-Line Generator Reference Manual).

2. By use of the TO operator command (refer to the RTE-6/VM Terminal User's Reference Manual).

3. By the driver (refer to the RTE Driver Writing Reference Manual).

For privileged drivers, the time-out parameter must be long enough to cover the period from I/O initiation to transfer completion.

Word	Contents																
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	R	I/O Request List Pointer <C>															
2	R	Driver Initiation Section Address <A >															
3	R	Driver Continuation/Completion Section Address <A>															
4	D <A>	B <B>		P <E>	S <E>	T <C>	lower 5 bits of Subchannel <C>						I/O Select Code # <A>				
5	AV <F>	EQUIPMENT TYPE CODE <A>									STATUS <E>						
6	CONWD (Current I/O Request Word) <C>													MSB <C>			
7	Request Buffer Address <C>																
8	Request Buffer Length <C>																
9	Temporary Storage <D> or Optional Parameter <C>																
10	Temporary Storage <D> or Optional Parameter <C>																
11	Temporary Storage for Driver <D>																
12	Temporary Storage for Driver <D> or EQT Extension Size, any <A>																
13	Temporary Storage for Driver <D> or EQT Extension Starting Address, if any <A>																
14	Device Time-Out Reset Value <B>																
15	Device Time-Out Clock <C>																

Figure E-1. Equipment Table Entry Format

where:

R = reserved for system use.

I/O Request List Pointer = points to list of requests queued up on this EQT entry. First entry in list is current request in progress (zero if no request).

D = 1 if DCPC required.

B = 1 if automatic output buffering used.

P = 1 if driver is to process power fail.

S = 1 if driver is to process time-out.

T = 1 if device timed out (system sets to zero before each I/O request).

Subchannel# = last subchannel addressed (lower 5 bits). Bit 2 (MSB) of EQT word 6 is the MSB of the subchannel.

I/O Select Code# = I/O select code for the I/O controller (lower number if a multi-board interface).

AV = I/O controller availability indicator:

0 = available for use.
1 = disabled (down).
2 = busy (currently in operation).
3 = waiting for an available DCPC channel.

EQUIPMENT TYPE CODE = type of device on this controller. When this octal number is linked with "DVy," it identifies the device's software driver routine. Some standard driver numbers are:

00 to 07 = paper tape devices or consoles

00 = teleprinter or keyboard control device

01 = photoreader

02 = paper tape punch

05 = 26xx-series terminals

07 = multi-point devices

10 to 17 = unit record devices

10 = plotter

11 = card reader

12 = line printer

15 = mark sense card reader

20 to 37 = magnetic tape/mass storage devices

23 = 9-track magnetic tape

31 = 7900 moving head disc

32 = 7905/06/20/25 moving head disc

33 = flexible disc drives 7908/11/12/35 moving head disc drive or cartridge tape drive.

36 = writable control store

37 = HPIB

40 to 77 = instruments

STATUS = actual physical status or simulated status at the end of each operation.

CONWD = combination of user control word and user request code word in the I/O EXEC call (see Figure E-2 below).

and where the letters in brackets (<>) indicate the nature of each data item as follows:

<A> = fixed at generation or reconfiguration time; never changes.

<B> = fixed at generation or reconfiguration time; can be changed on-line.

<C> = set up or modified at each I/O initialization.

<D> = available as temporary storage by driver.

<E> = can be set driver.

<F> = maintained by system.

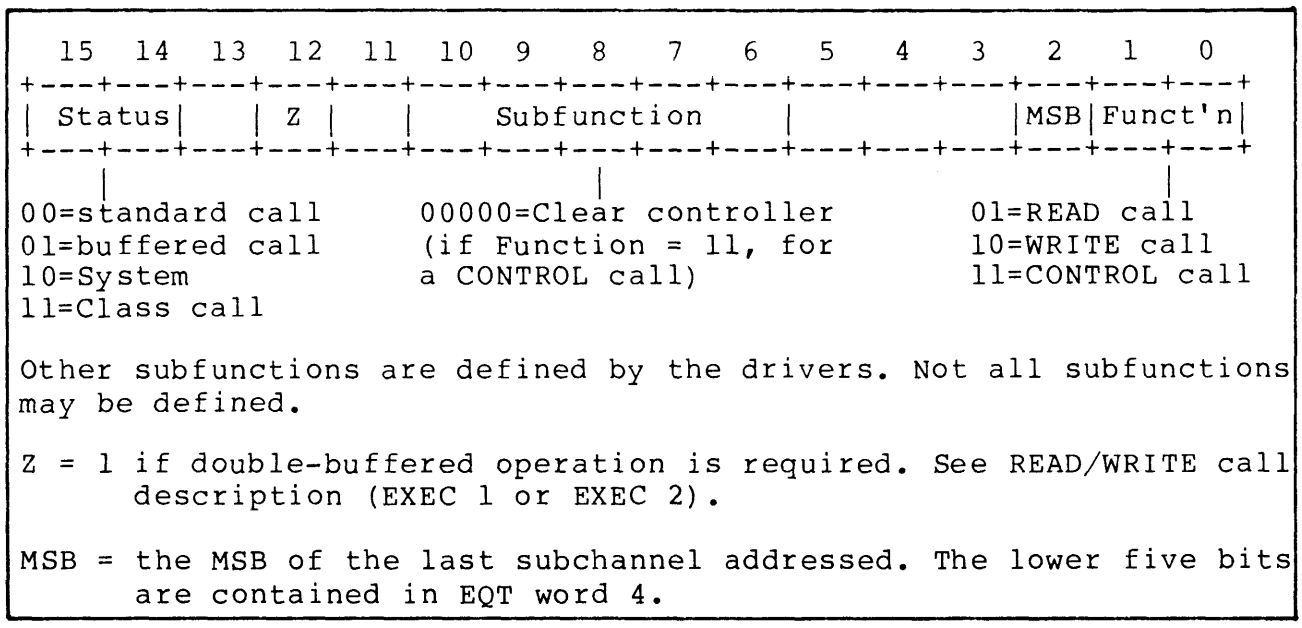


Figure E-2. CONWD Word (EQT Entry Word 6) Expanded

Device Reference Table (DRT)

The Device Reference Table (DRT) is used by the I/O processor to relate LU numbers to EQT entries. When a user makes an I/O request specifying an LU number, The LU is translated into an EQT entry via the DRT. Since the EQT entry contains all the information necessary to operate the I/O device, the transfer can then be initiated.

Each DRT entry is five bytes long. There is one entry for each LU defined at generation time, beginning with LU 1.

The first word (bytes 1 and 2) of each entry includes the EQT entry number of the controller assigned to the LU and the subchannel number of the specific device on that controller to be referenced.

The second word (bytes 3 and 4) of each DRT entry contains the current status of the LU; up (available) or down (unavailable). If the device is down, word 2 also contains a pointer to the list of requests waiting to access the LU.

The fifth byte indicates the condition of the LU; locked or unlocked.

There are separate tables for words 1 and 2. The word 2 table is located in memory immediately following the word 1 table. The starting address and length of the word 1 table are recorded in the System Base Page (1652B and 1653B, respectively). The format of the Device Reference Table is illustrated in Figure E-3, E-4, and E-5, below.

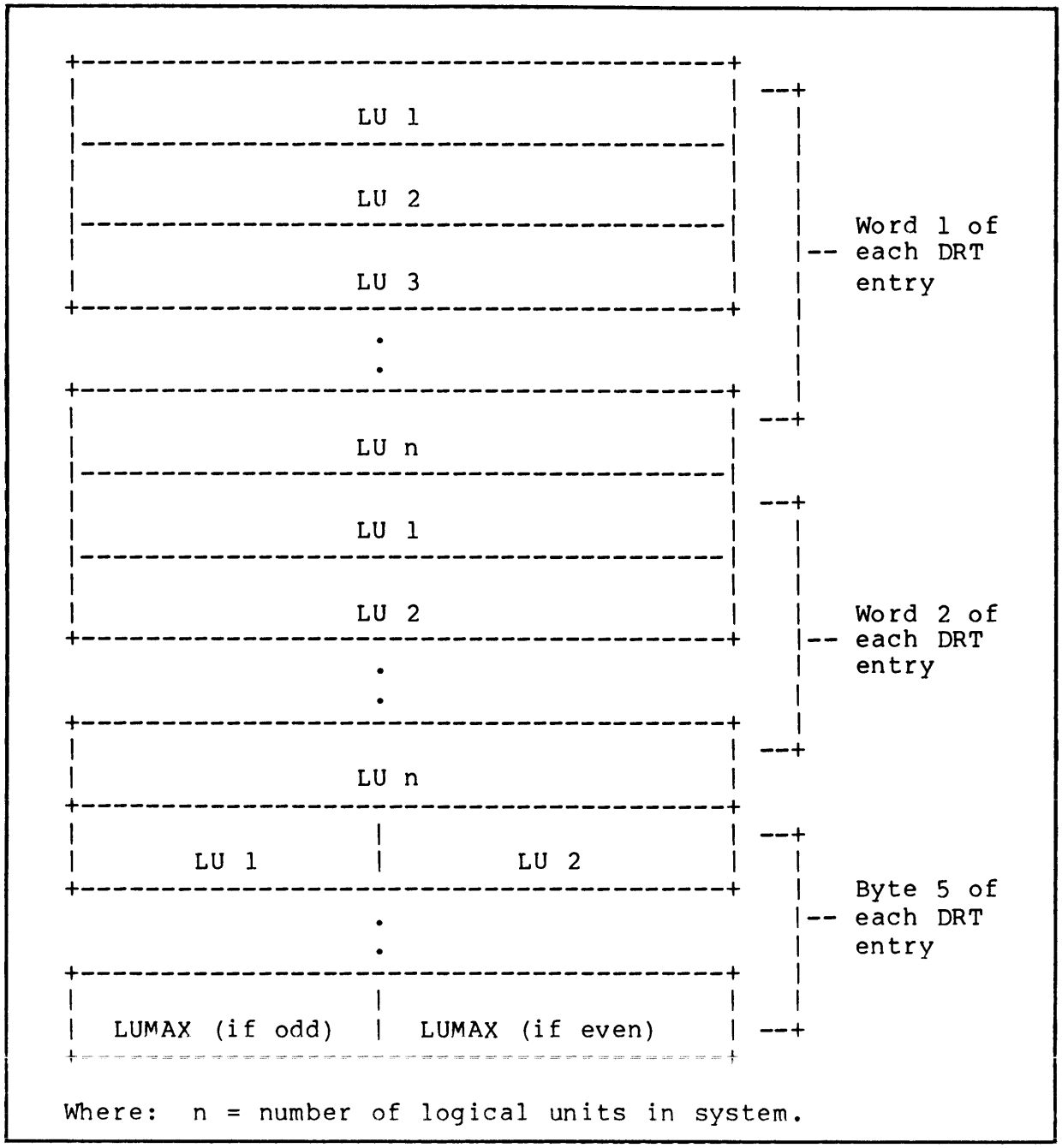
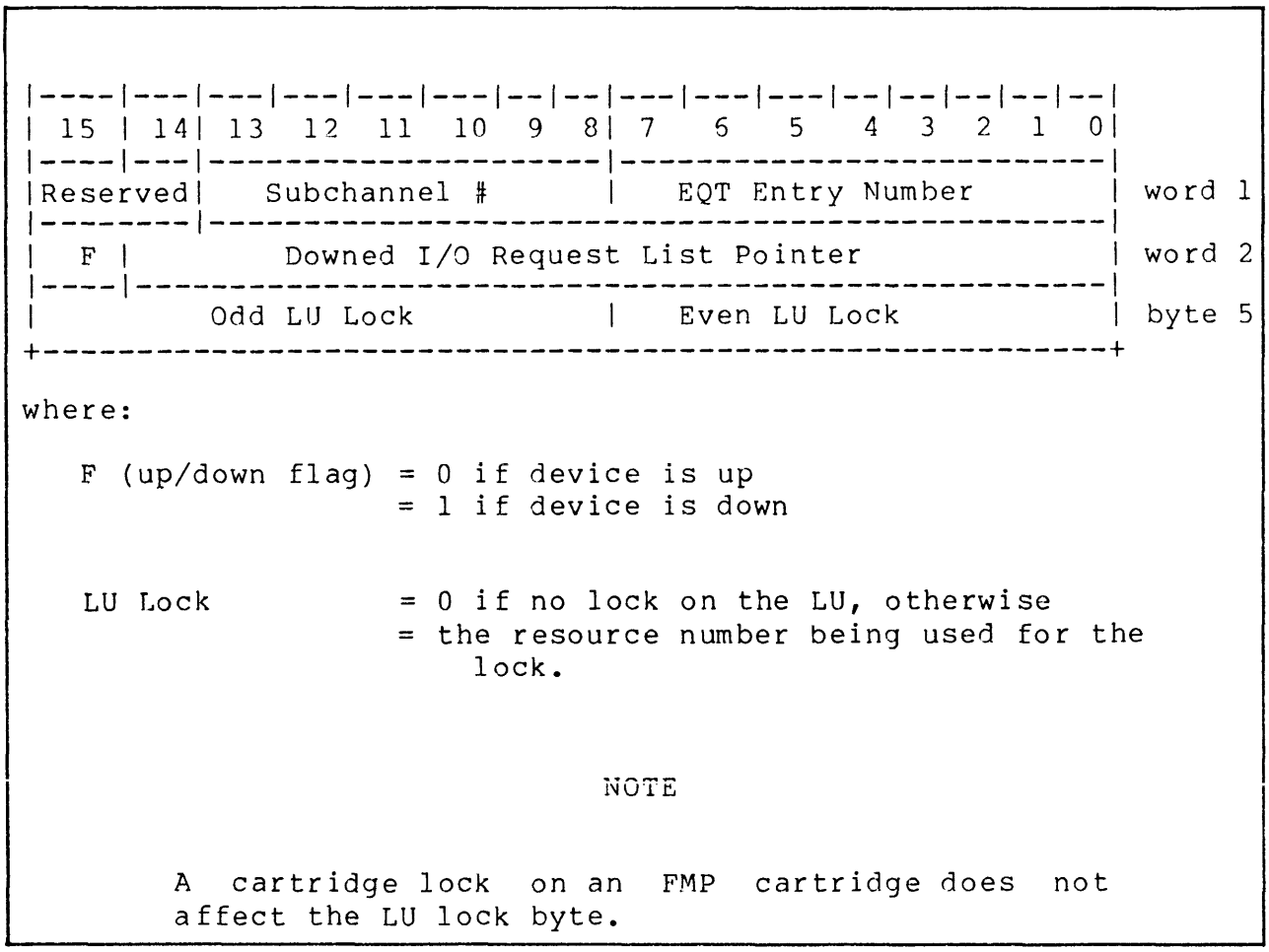


Figure E-3. Device Reference Table (DRT).



where:

F (up/down flag) = 0 if device is up
= 1 if device is down

LU Lock = 0 if no lock on the LU, otherwise
= the resource number being used for the lock.

NOTE

A cartridge lock on an FMP cartridge does not affect the LU lock byte.

Figure E-4. Device Reference Table Entry Format

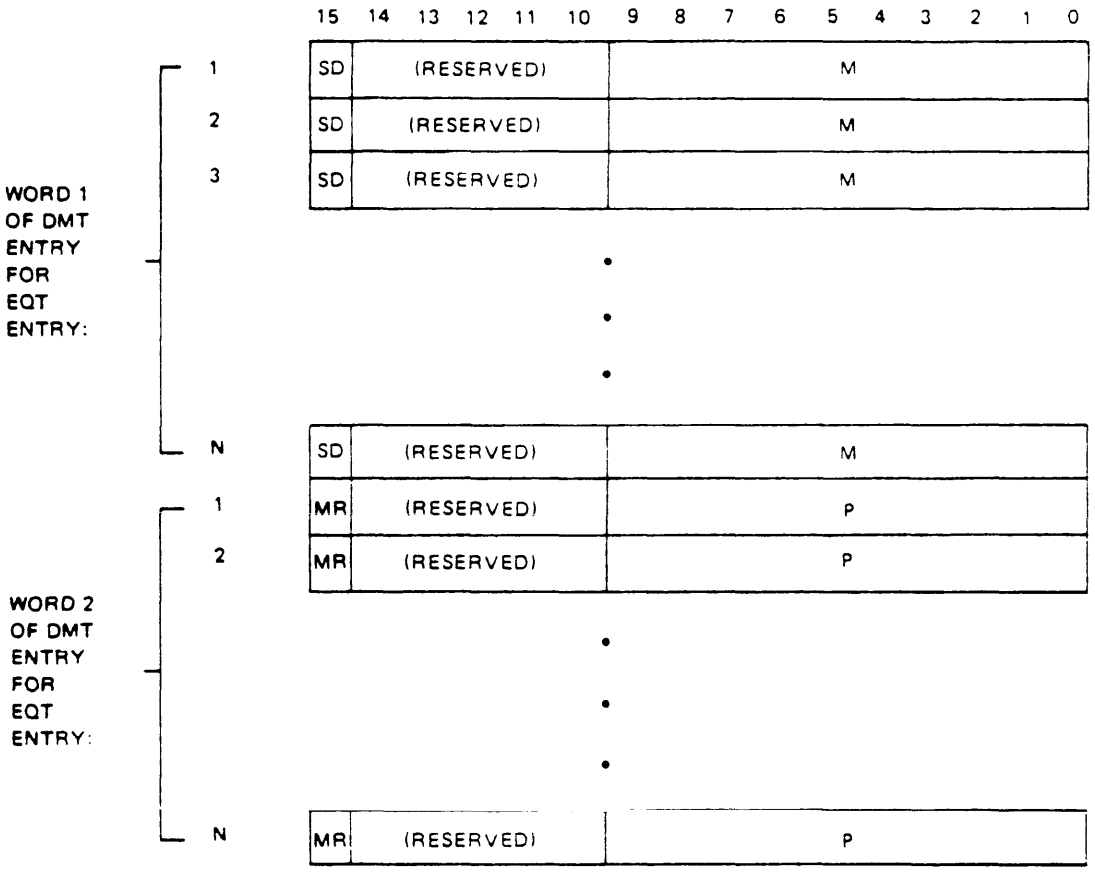
Driver Mapping Table (DMT)

In the RTE-6/VM Operating System, driver modules can be placed in one of two areas; the System Driver Area (SDA) or in one of the driver partitions. Most drivers are placed in driver partitions. The SDA is primarily used for privileged drivers, drivers that do their own mapping, and very large drivers.

The Driver Mapping Table (DMT) is used to record where a driver resides in physical memory, static and dynamic information about the driver, and the location of the I/O request buffer.

There is one DMT entry associated with each EQT entry defined at generation time. Each entry is two words long. Word 1 is set up at generation time and its contents are never changed. It indicates whether the driver resides in the System Driver Area (SDA) or in a driver partition. If it is in the SDA, it also indicates whether or not the driver is doing its own memory mapping. If the driver is in a partition, word 1 also indicates the starting physical memory page number of the driver partition in which it is located.

Word 2 of the DMT entry is dynamic in nature and is set up at each I/O initialization of the associated EQT entry. This word indicates whether the I/O request buffer is located within a disc-resident program, program, memory-resident program, or system area. If a disc-resident program is making the request and the I/O request buffer is located within the program (i.e., an unbuffered request), word 2 also indicates the physical memory page number of the disc-resident program's base page. This information is used to save time on setting up the program map when processing interrupts handled by the driver. The format of the DMT is shown in Figure E-5, below.



WHERE:

SD = 0 IMPLIES DRIVER RESIDES IN A DRIVER PARTITION, AND
M = STARTING PAGE NUMBER OF PARTITION IN BITS 0-9

SD = 1 IMPLIES DRIVER RESIDES IN SYSTEM DRIVER AREA, AND
M = 0 IMPLIES DRIVER NOT DOING ITS OWN MAPPING
M = 1 IMPLIES DRIVER DOING ITS OWN MAPPING

MR = 1 IMPLIES THAT THE I/O REQUEST BUFFER IS LOCATED IN A MEMORY RESIDENT PROGRAM.
(P VALUE NOT SIGNIFICANT – RESERVED FOR FUTURE USE)

MR = 0 IMPLIES THAT THE I/O REQUEST BUFFER IS NOT LOCATED IN A MEMORY RESIDENT PROGRAM. BUFFER LOCATION IS INDICATED BY THE VALUE OF P, AS FOLLOWS:
P = 0 IMPLIES BUFFER IS IN THE SYSTEM AREA
P NOT ZERO IMPLIES BUFFER IS LOCATED IN A DISC RESIDENT PROGRAM. P IS THE PHYSICAL PAGE NUMBER OF THE PROGRAM'S BASE PAGE

N = NUMBER OF EQT ENTRIES IN SYSTEM

Figure E-5. Driver Mapping Table

Interrupt Table and Trap Cells

After an I/O request has been routed through the EXEC processor, the actual data transfer is accomplished by processing interrupts from the device's controller. The I/O controller generates an interrupt after each word transferred.

When an interrupt is received, the computer transfers control to one of a group of memory locations, known as trap cells, in the system base page. The I/O select code of the interrupting controller determines the location of the transfer, i.e., interrupts from select code 12 cause a transfer to memory location 12; interrupts from select code 13 cause a transfer to location 13, etc. Memory locations from octal 4-77 comprise the entire set of the interrupt trap cells, where:

```
    4 = powerfail

    5 = memory protect/DMS/parity error

    6 = DCPC Port 1

    7 = DCPC Port 2

10-77 = I/O slots
```

Transferring control to an interrupt trap cell causes the instruction located there to be executed. This instruction is a JSB to a microcode routine. For M-Series or E/F-Series without firmware, this instruction is a JSB LINK,I where LINK contains the address of the entry point to the Central Interrupt Controller (CIC). This instruction is initially set up at generation time.

Interrupt Handling Without Microcode

After the CIC has been entered via the execution of a trap cell instruction, it checks the contents of the Interrupt Table entry associated with the select code that the interrupt occurred on.

The Interrupt Table contains an entry, established at generation time, for each I/O select code in the computer. The entries can be positive, negative, or zero.

If the contents of the entry is positive, the entry contains the address of the EQT entry associated with the I/O controller in that select code.

If the contents of the entry is negative, the entry contains the negated ID segment address of the program to be scheduled when an interrupt occurs on that select code.

When the CIC has obtained the contents of the Interrupt Table entry associated with the interrupting select code, it initiates the appropriate action. For positive Interrupt Table entries, CIC obtains the driver continuation/completion entry point and transfers control to that point. For negative Interrupt Table entries, CIC transfers control to the scheduling module which schedules the program indicated by the ID segment address.

Interrupt Table entries that contain zero, indicate that CIC and the Interrupt Table are by-passed when an interrupt occurs on that select code. In this case, the trap cell contains the direct entry point to an interrupt processing routine or an octal instruction code placed in the trap cell at generation. This technique would be used with time-critical events such as processing privileged interrupts or power fail interrupts. Table E-1, shows the four possible relationships between an Interrupt Table entry and the trap cell instruction associated with a specific select code.

Several conditions can occur during interrupt processing that cause a message to be displayed on the system console. These conditions are summarized at the end of Chapter 2.

Table E-1. Interrupt Table Example

Generation Entry (for SC 12)	Interrupt Table Contents	Trap Cell Contents
12,EQT,1	EQT entry address	JSB LINK,I
12,PRG,name	Negative ID segment	JSB LINK,I
12,ENT,entry	0	JSB entry,I
12,ABS,octal instruction code	0	octal instruction code

The beginning of the Interrupt Table and the number of entries are stored in System Base Page locations 1654B and 1655B respectively. The first entry in the table is for Select Code 6, (DCPC channel 1) and the second entry is for select code 7 (DCPC channel 2). These first two locations are dynamic in nature; they are changed each time a DCPC channel is assigned for a DMA transfer.

Interrupt Handling With Microcode

On E/F-Series computers with the appropriate firmware, interrupts can be handled faster through microcode routines. An interrupt causes control to be transferred to a trap cell containing a JSB to a microcode routine.

There are four possible microcode routines which can be called from the trap cells. These microcode routines branch to the approprite interrupt handling system routine depending on the type of interrupt; TBG, memory protect, DMA, or I/O request.

Power Fail/Auto Restart

The optional Power Fail/Auto Restart feature available with RTE-6/VM is designed to save the computer status when the line voltage drops below a predetermined level (power fail). When a power fail occurs, the machines status indicators are moved from their volatile hardware registers to memory locations where they can be maintained by a back-up battery power supply.

The Power Fail/Auto Restart routine is generated as a driver (DVP43) into the System Driver Area (SDA). DVP43 is a privileged driver that does its own mapping. The primary entry point to DVP43 is $POWR. The initiator and continuation/completion entry points are IP43 and CP43, respectively. Note tha the CPU switch (A1S2) must be set properly in order for the Power Fail/Auto Restart routine to execute with a power fail occurs.

The following steps illustrate the action taken by the system when a power fail occurs:

1. When power drops below a predetermined level, (or is restored), the condition is detected by the hardware power-sensing circuits of the HP-1000 computers. The hardware circuits cause an interrupt to occur on select code 4 which causes the instruction in Trap Cell 4 to be executed (indirect jump to $POWR).

2. When DVP43 is entered via $POWR, a check is made to determine if power is going down or coming up and the appropriate section of code is jumped to accordingly.

DOWN ROUTINES:

a) DMA transfers are stopped and the word counts are saved.

b) The registers (e.g., A, B, O, E, X, Y, memory status register) and the point of power-fail are saved.

c) The four memory maps are saved.

d) The power fail logic is reset so the next interrupt on select code 4 will be considered a power-up.

e) Halt.

UP ROUTINES:

a) Check if power down routine had time to complete. If so, continue; if not, halt.

b) Restore memory maps and memory status register.

c) Search the Equipment Table (EQT) for the DVP43 entry; set its time out counter to -1 (time-out on first tick of system clock).

d) Save the system time (the clock has not been restarted yet, so this is the time of powerfail).

e) Determine state of interrupt system when power failed, and set up to restore when DVP43 was exited.

f) Restore registers and start system clock.

g) Exit to point of power-fail.

3. After the exit from DVP43, the first tick of the system clock will time out DVP43's EQT entry, and DVP43 is entered at CP43 (continuation/completion entry point). The code at CP43 does the following:

   a) Sets up DVP43 to time out in one tick of system clock.

   b) Searches the Equipment Table to determine the state of the interrupt system at the time of power fail and takes appropriate action as follows:

      * If the EQT entry was busy (AV=2) and its power fail bit ("P") set, enter the driver at InXX. The driver's initiator section notes that AV=2 and thus the request is for a power-failure and the driver takes appropriate action.

      * If the EQT entry was waiting for a DCPC channel (AV=3), no action is taken.

      * $UPIO is called for all other EQT entries to restart requests that were in progress or were pending by calling each driver at InXX.

   c) After an EQT entry has been serviced, CP43 (or $UPIO) "idles", waiting for the system clock to tick and time out DVP43 again (re-enter DVP43 at CP43).

4. When all the EQT entries have been serviced, a FORTRAN program (AUTOR) is aborted (could have been scheduled from a previous power-fail) and the scheduled. AUTOR is written in FORTRAN to allow for easy user modification to meet site-dependent requirements. AUTOR does the following:

   a) Does an EXEC read on the power fail LU (enter at IP43) to get the power fail time.

   b) Formats the time into a message and writes it to each interactive terminal.

   c) Reenables all terminals.

   d) Does a second EXEC read to the power fail LU to signal that the recovery process is complete.

Standard I/O Request Flow

A user program makes an EXEC call to initiate I/O transfers. If the device's controller is not buffered or the I/O transfer is for input, the calling program is suspended until the transmission is completed. The next lower priority program is allocated execution time during the suspension of a higher priority program.

An I/O request (READ, WRITE, CONTROL) is channeled to the I/O processor by the EXEC processor. After the necessary legality checks are made, the request is linked into the request list associated with the I/O controller's EQT entry.

If the device's controller is available (i.e., no prior requests pending), preparation is made to enter the driver's initiation section. The parameters from the request are set in the temporary storage area of the EQT entry.

The proper mapping registers are set up if the Driver Mapping Table indicates they are needed. The decision to choose the User Map or the System Map is decided by the type of I/O request. All system I/O, class I/O, and buffered user I/O requests require the use of the System Map since their I/O buffers are located in SAM.

Unbuffered user requests require the User Map. Note that in the case of a driver located in the System Driver Area making unbuffered requests, the program must be Type 1, 2, or 3, unless the driver does its own cross map instructions.

If the disc-resident program's User Map needs to be modified to map in a partition resident driver, the User Map is saved in the program's physical base page. The second word of the driver's mapping table entry is modified to record the type of map needed and, if it is a disc-resident program's map, the physical base page number is also kept. This second word is used to save time on setting up the map registers for a subsequent continuation interrupt. The initiation section initializes the device's controller and starts the data transfer or control function.

If the device's controller is busy on return from the initiation section or a required DCPC channel is not available, the I/O processor returns to the scheduling module to execute the next lower-priority program.

If the device's controller (EQT entry) or the device (LU) is down, the calling program is automatically suspended in the general wait list (status=3) and a diagnostic message is sent to the users teminal. The program is swappable while in this list. If a down LU or EQT entry is set UP, the program is automatically rescheduled.

Interrupts from the device's controller cause the Central Interrupt Control (CIC) module of the I/O processor to call the continuation/completion section of the driver. The I/O processor sets up the correct map before entering the driver. This is done by checking the Driver Mapping Table entry associated with the EQT entry. At the end of the operation, the driver returns to the I/O processor.

The I/O processor causes the requesting program to be placed back into the scheduled list and checks for additional requests queued on the EQT. If there are no queued requests the I/O processor exits to the dispatching module; otherwise, the initiation section is called to begin the next operation before returning.

Figure E-6 shows the various tables and control modules involved in standard I/O processing.

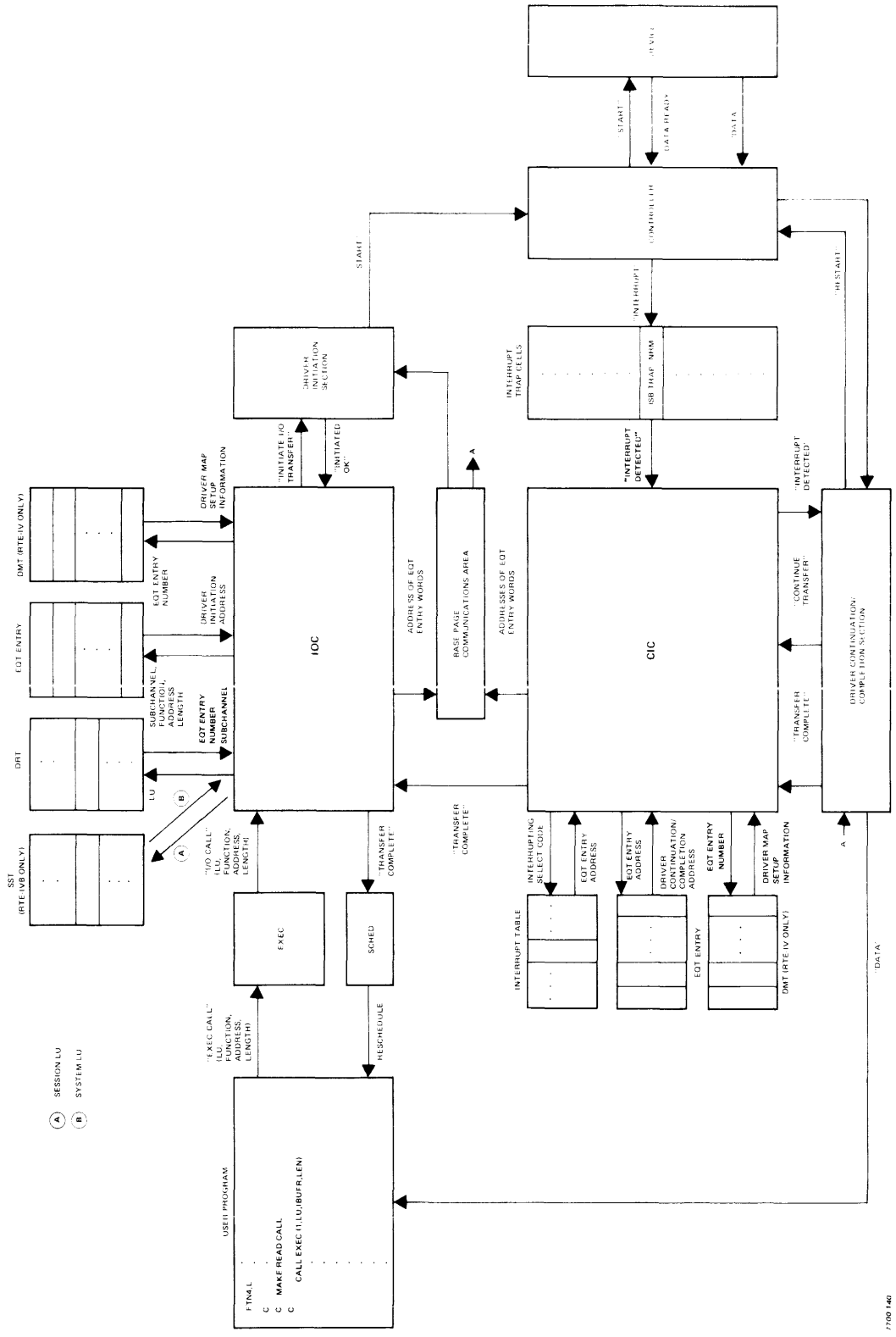


Figure E-6. Unbuffered EXEC Read Request Flow

Appendix F
Memory Management and Related Tables

This appendix contains information on the following:

* ADDRESS TRANSLATION
* LOGICAL MEMORY AND BASE PAGE
* MEMORY ALLOCATION TABLE (MAT)

Address Translation

There are four memory maps used by the DMS. Each map consists of 32, 12-bit hardware registers. The contents of the first 10 bits (bit 0 - bit 9) of each map register points to a 1024 word section (one page) of physical memory. DMS breaks the basic 15-bit address into a 5-bit page index (bit 10 - bit 14) and a 10-bit page off-set (bit 0 - bit 9). The page index points to one of the 32 map registers in the currently enabled map. The contents of this map register (pointer to page in physical memory) is then concatenated with the 10-bit page off-set obtained from the original address (pointer to word within page) to form the final 20-bit address necessary to access 1024K word of physical memory (see Figure F-1).

When using virtual memory, several address translations occur. The virtual address is a 26-bit address which is translated into a 15-bit logical address via the VMA/EMA firmware and the page table. This is further translated into a 20-bit physical address by the DMS hardware.

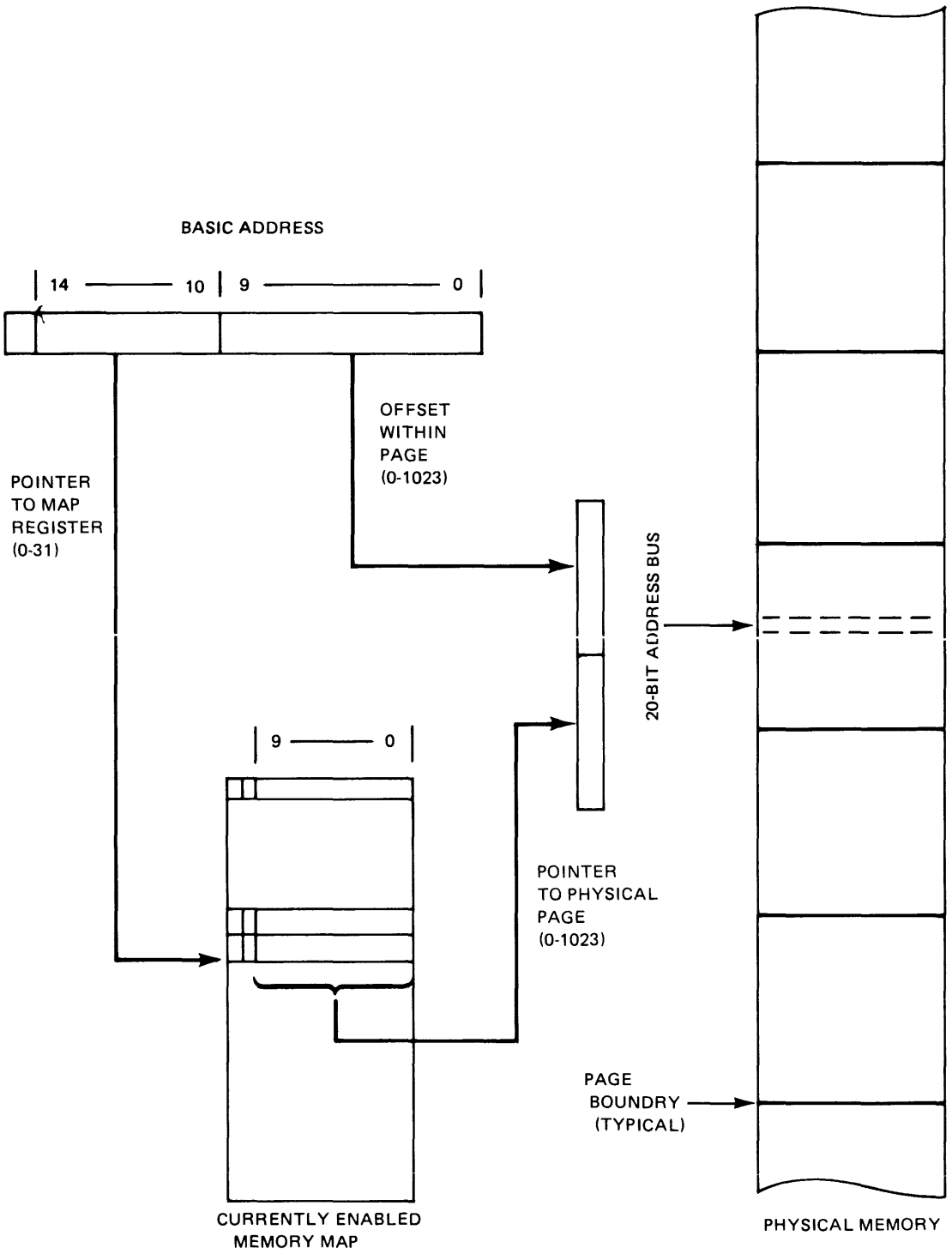


Figure F-1. Address Translation via Memory Mapping

Logical Memory and Base Page

Logical memory is the 32K-word address space described by the currently enabled memory map. Figure F-2 shows the five possible configurations of the 32K word logical address space. The first configuration illustrates how this space appears under control of the System Map. Note that there is always a total of 32 pages to be divided up, however, the particular boundaries shown for the various parts are examples only, and a user's system could be configured differently.

The second configuration illustrates how the logical address space appears under control of the User Map when a memory-resident program is executing.

The third configuration illustrates how the logical address space appears under control of the User Map when either an RT or Type 3 (BG) disc-resident program is executing.

The fourth configuration illustrates how the logical address space appears under control of the User Map when a Type 4 (LB) disc-resident program is executing.

The fifth configuration illustrates how the logical address space appears under control of the User Map when a Type 6 (EB) disc-resident program is running.

Many programs will not require a full 32K of space, and unneeded pages will be READ/WRITE protected as illustrated in the User Map given in Figure F-2, configuration 3.

The system area, memory-resident program area and each disc-resident program have their own logical base pages, as follows:

a. The system base page contains the communication area, system links, driver links, SSGA links, tables area links and trap cells for interrupt processing.

b. The disc-resident program base page contains the system communication area, driver links, SSGA links, table area links, and disc-resident program links.

c. The memory-resident base page has the memory-resident program links, resident library links, System Communication area, table area links, SSGA links, and driver links.

The System Communications area (described in the RTE-6/VM General Information Guide), driver links, SSGA and table area links located in physical page 0 are common to all base pages. Base page structures are illustrated in Figure F-3.

The Base Page Fence (refer to the HP/1000 M/E/F-Series Computer Operating and Reference Manual) is automatically set by the system for all user base pages so that the bottom portion of the base page will contain the user program links.

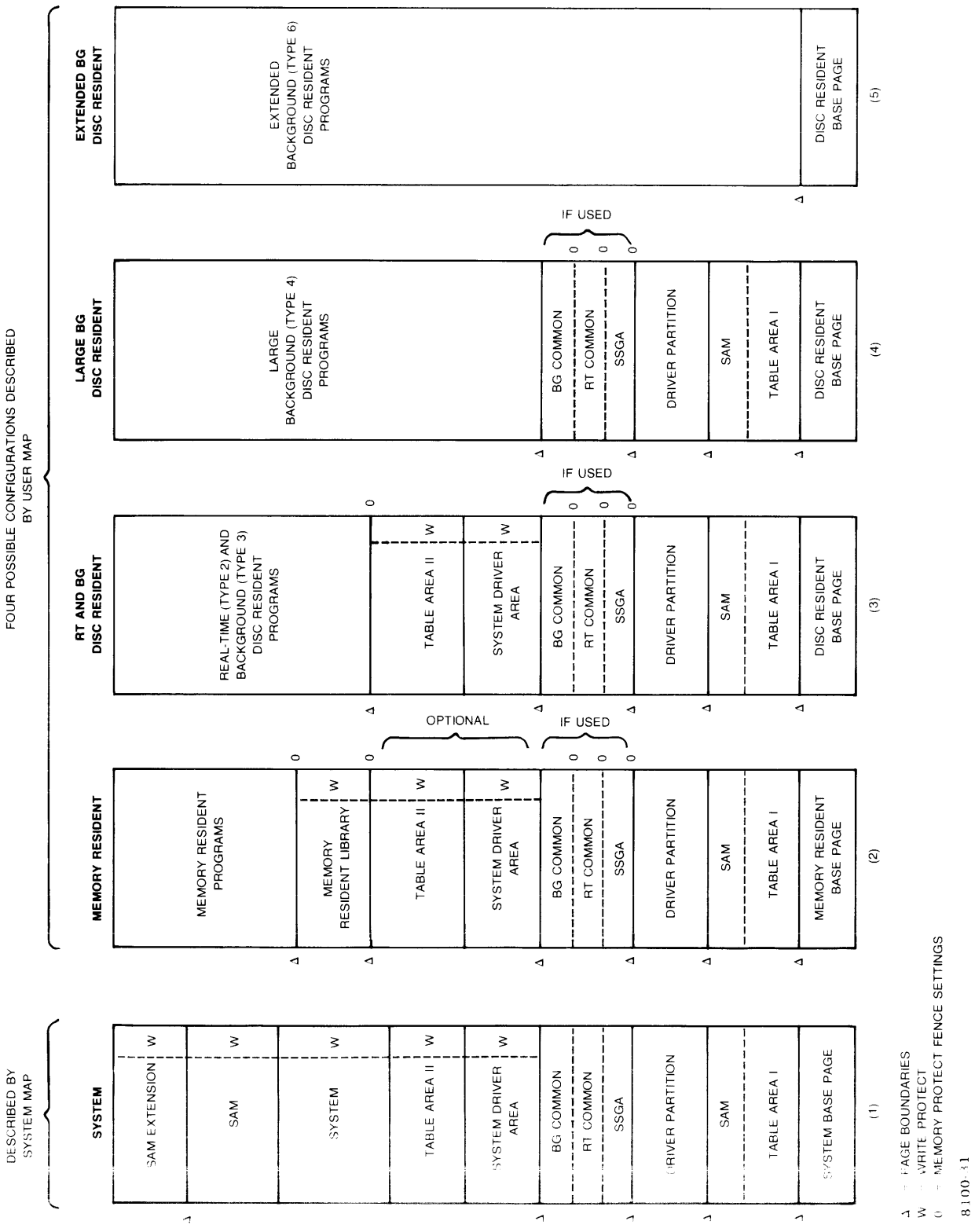


Figure F-2. RTE-6/VM 32K Word Logical Memory Configurations

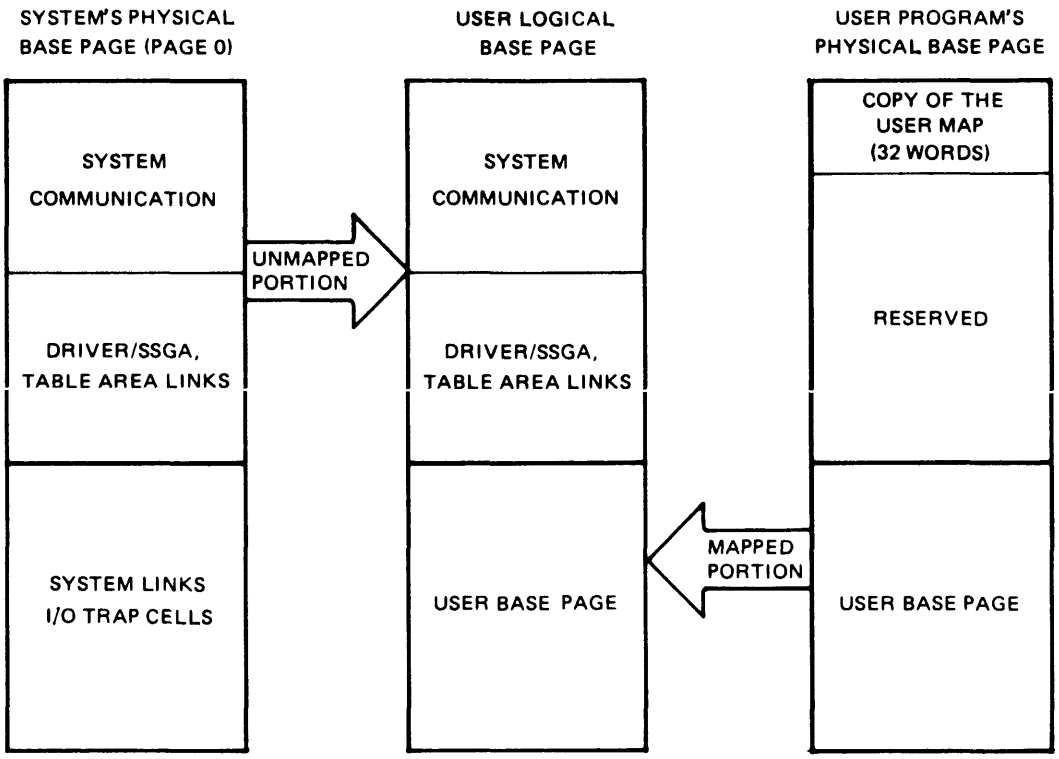


Figure F-3. Base Page Structure

Memory Allocation Table Entry

Each partition defined by the user during generation contains an entry in the Memory Allocation Table (MAT). This table starts at the system entry point $MATA and extends upward toward high memory. Each entry is seven words long, arranged as illustrated in Figure F-4.

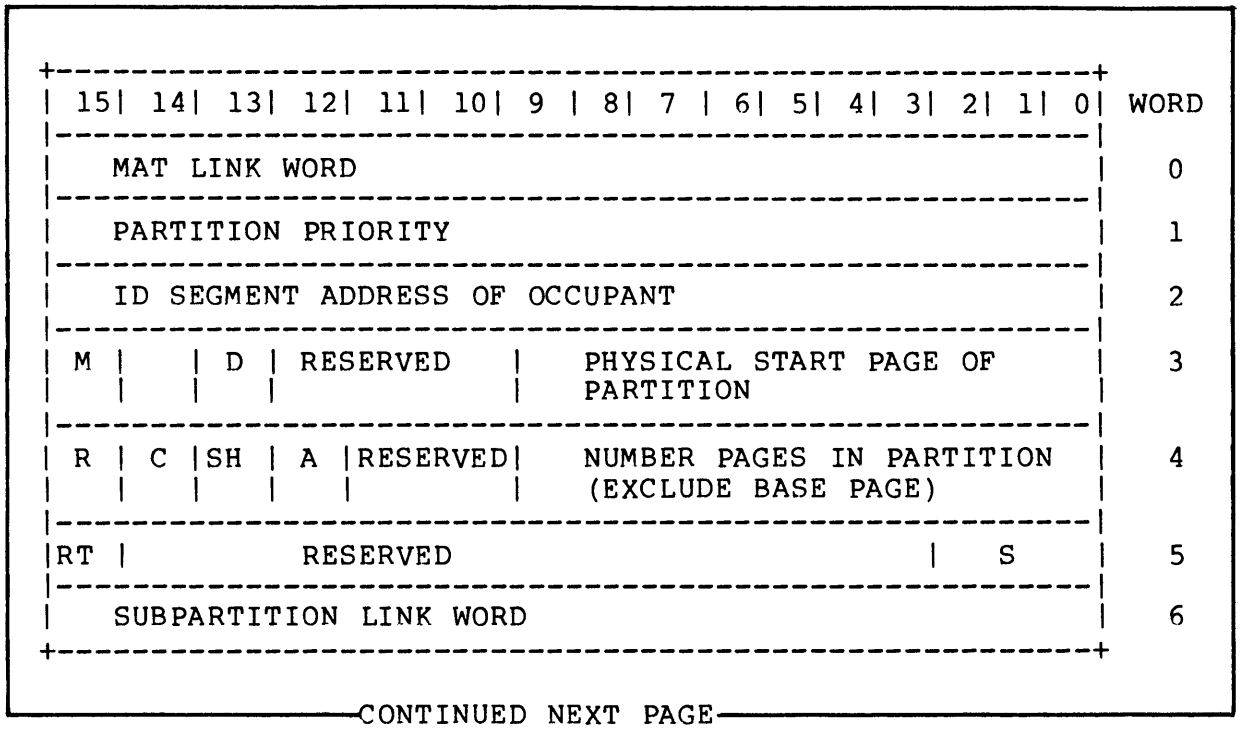


Figure F-4. Memory Allocation Table Entry Format

CONTINUED FROM PREVIOUS PAGE

Where:

MAT LINK WORD

= -1 if partition not defined either during generation memory reconfiguration, or by parity error.

= 0 is end of list.

= positive = next partition address in list.

M = 1 if MAT entry is for a mother partition.

D = 1 if program is dormant after save-resource or serially-reusability termination, or operator suspended.

R = 1 if partition is reserved.

C = 1 if partition is in use as part of a chained mother partition.

SH = 1 if MAT entry is for a shareable EMA partition or a subpartition of a shareable EMA partition.

A = 1 if partition is active in shareable EMA mode.

RT = 1 if MAT entry is for real-time partition.
0 if MAT entry is for background partition.

S = Program's dispatching status:
= 0 - program being loaded.
1 - program is in memory.
2 - segment or MLS disc-resident node is being loaded or is being swapped out.
3 - program is swapped out.
4 - subprogram swap-out started for mother partition.
5 - subpartition swap-out completed. Mother cleared.

SUBPARTITION LINK WORD

= 0 if MAT entry is not a subpartition.
= next partition address is this is a subpartition.
= mother partition MAT address if this entry is the last subpartition.

Figure F-4. Memory Allocation Table Entry Format (cont)

Appendix G
DCB and Directory Formats

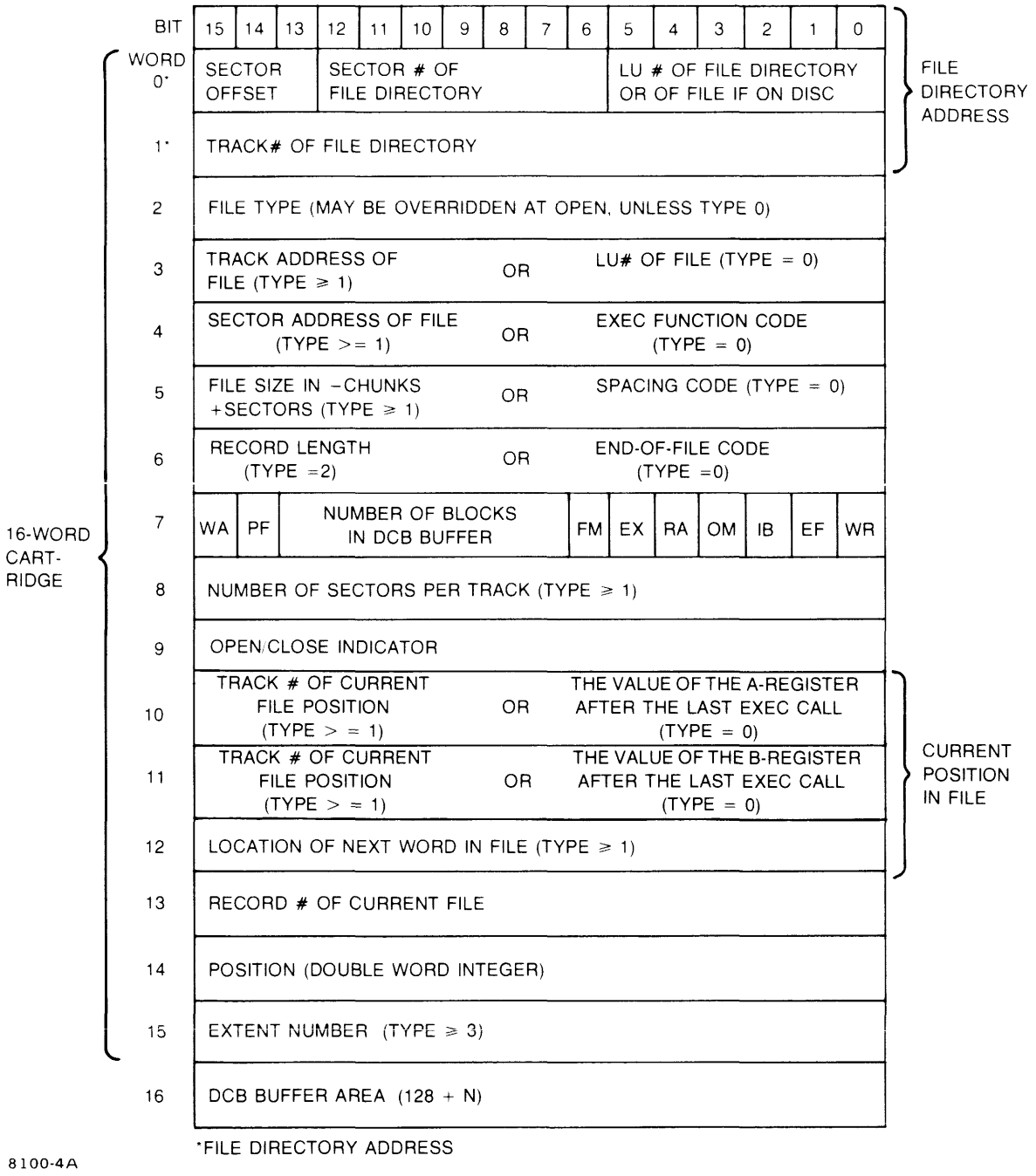
This Appendix contains information on the following:

* DATA CONTROL BLOCK (DCB) FORMAT

* CARTRIDGE DIRECTORY FORMAT

* FILE DIRECTORY FORMAT

Data Control Block (DCB) Format

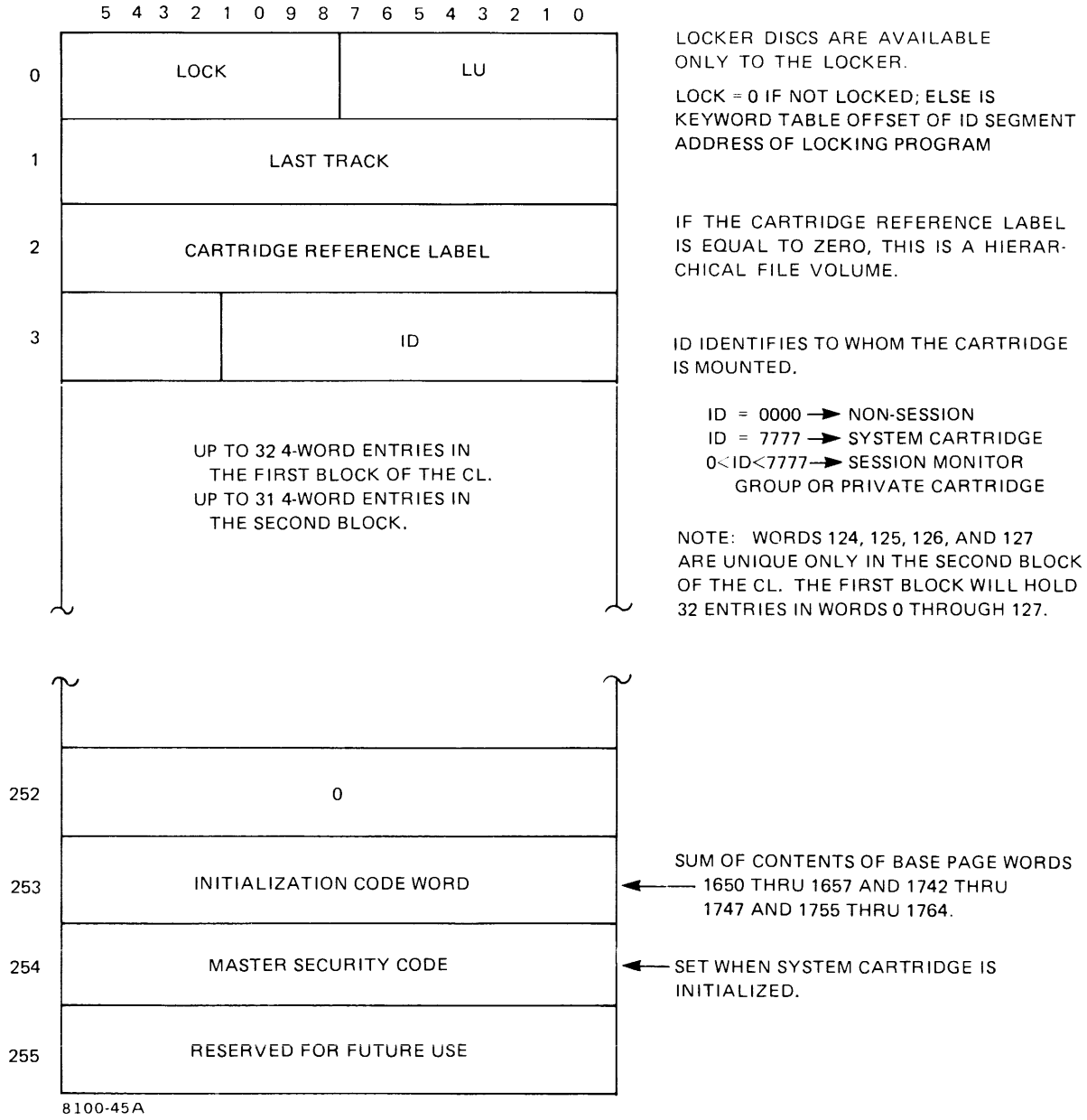


Legend for Data Control Block (DCB)

Word	Contents
0 File Directory Address:	bits 6-12 = Physical sector number of the file directory. bits 13-15 = Entry offset from the beginning of the block (origin 0).
5 Spacing Code: (type 0 file)	bit 15 = 1 - backspace legal. bit 0 = 1 - forward space legal.
6 End-of-File Code: (type 0 file)	01 lu = EOF on Magnetic Tape. 10 lu = EOF on Paper Tape. 11 lu = EOF on Line Printer.
7 Status Information	
(WA) Write Allowed:	bit 15 = 1 - write to file allowed. = 0 - write to file not allowed.
(PF) Partially Full:	bit 14 = 1 - DCB is only partially full. = 0 - DCB is full.
DCB Buffer:	bits 13-7 = Number of blocks in the DCB buffer.
(FM) File Modify:	bit 6 = 1 - File has been modified. = 0 - File has not been modifed.
(EX) Extendable:	bit 5 = 1 - File is not extendable. = 0 - File is extendable.
(RA) Read Allowed:	bit 4 = 1 - Read from file allowed. = 0 - Read from file not allowed.
(OM) Open Mode:	bit 3 = 1 - update open. 0 - standard open.
(IB) In Buffer Flag:	bit 2 = 1 - data in DCB buffer. = 0 - data not in DCB buffer.
(EF) EOF Read Flag:	bit 1 = 1 - EOF has been read. = 0 - EOF has not been read.
(WR) To Be Written:	bit 0 = 1 - data in DCB buffer to be written. = 0 - data in DCB buffer not to be written.
9 Open/Close Indicator: if open, contains ID segment location of program performing open. If closed, set to zero.	

Cartridge Directory Format

The cartridge directory is located in the system area on LU 2. Its length is two blocks.

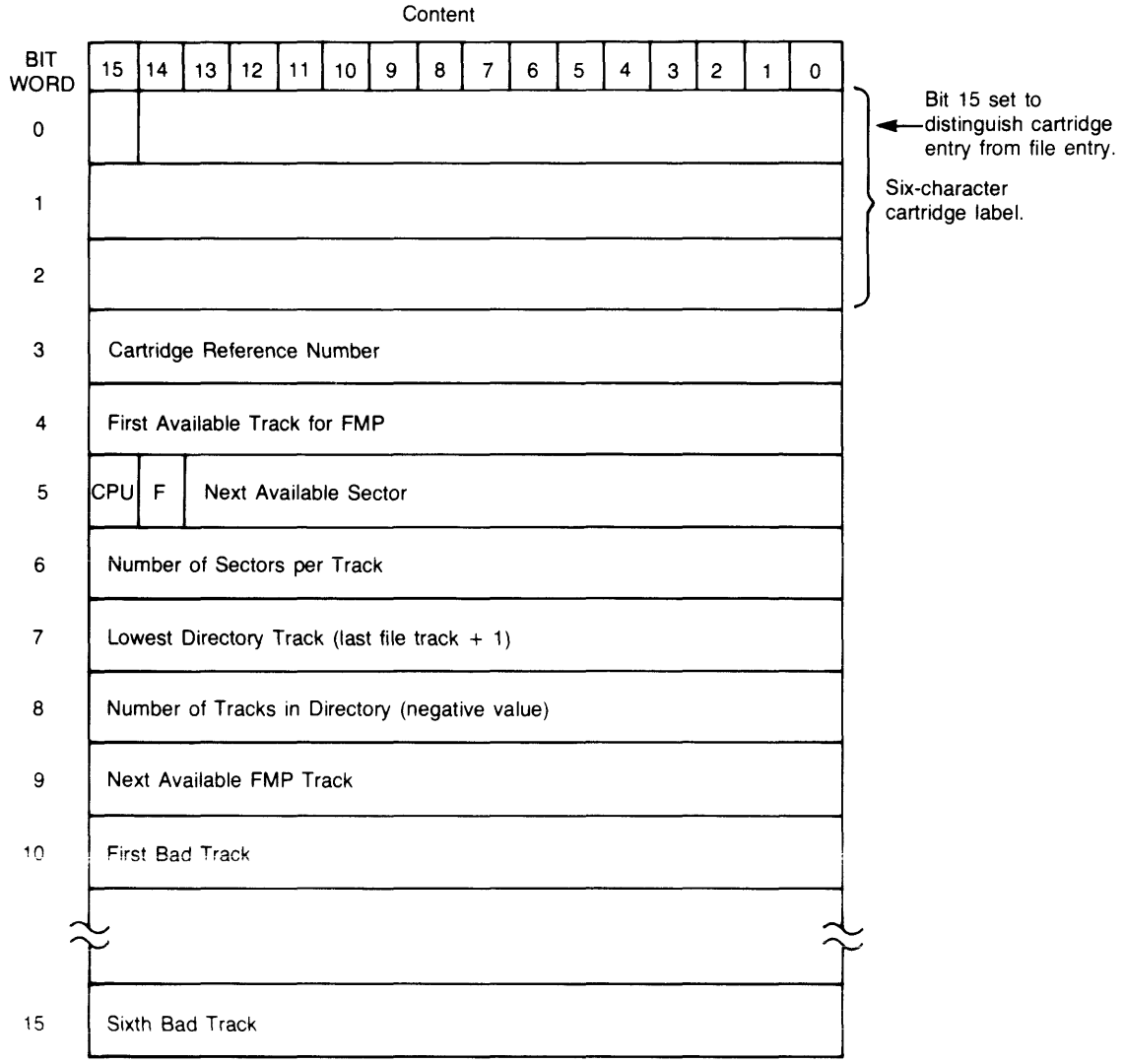


File Directory

The first entry in each File Directory is the specification entry for the cartridge itself. The directory starts on the last FMP track of each cartridge in sector zero on all discs. The directory blocks are written using sector skipping. The directory sector address can be obtained from the block address by the following formula:

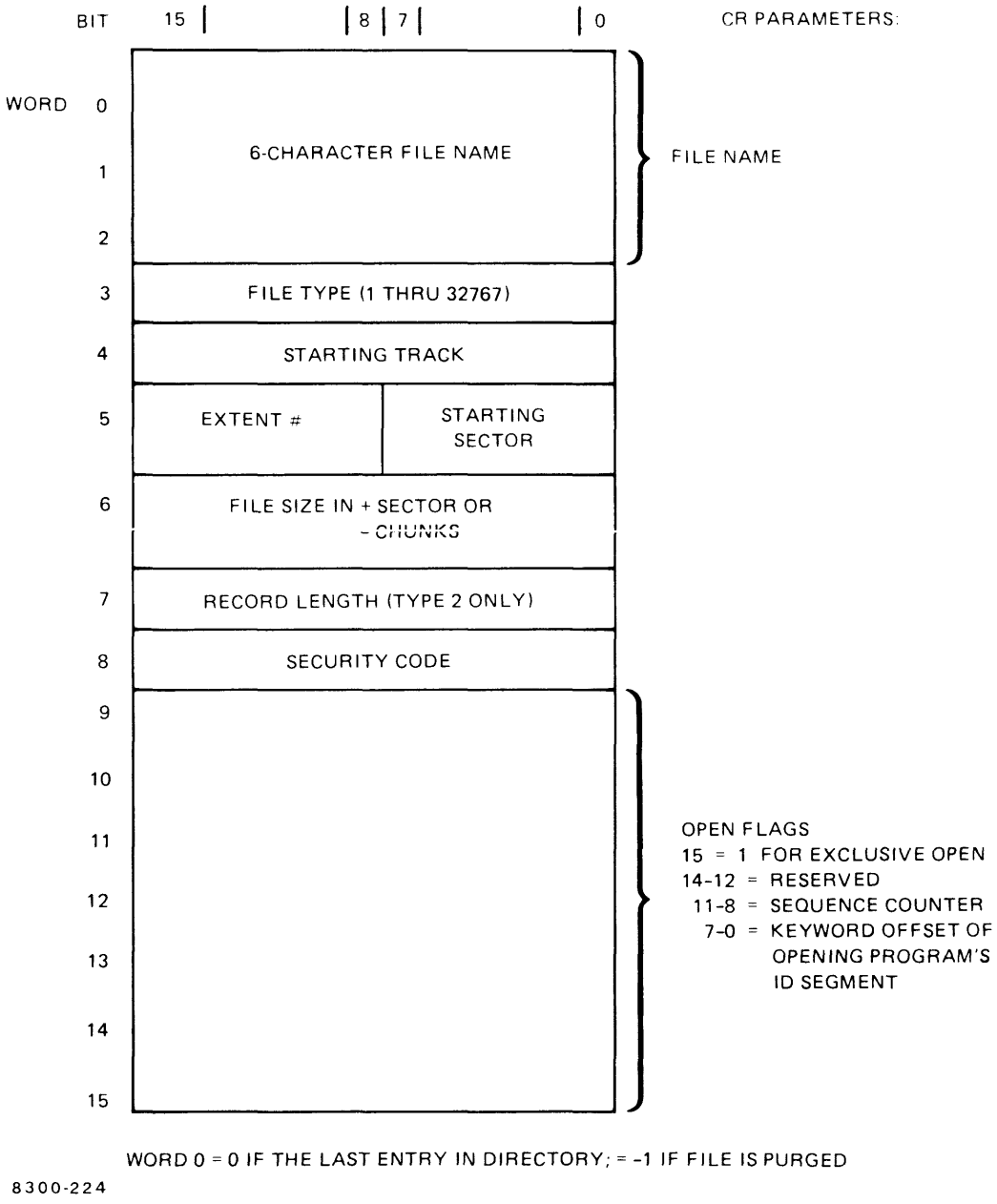
sector address = (block *14) modulo S/T

where S/T is the number of sectors per track. Directory blocks are 128 words long. Each Directory entry is 16 words long.



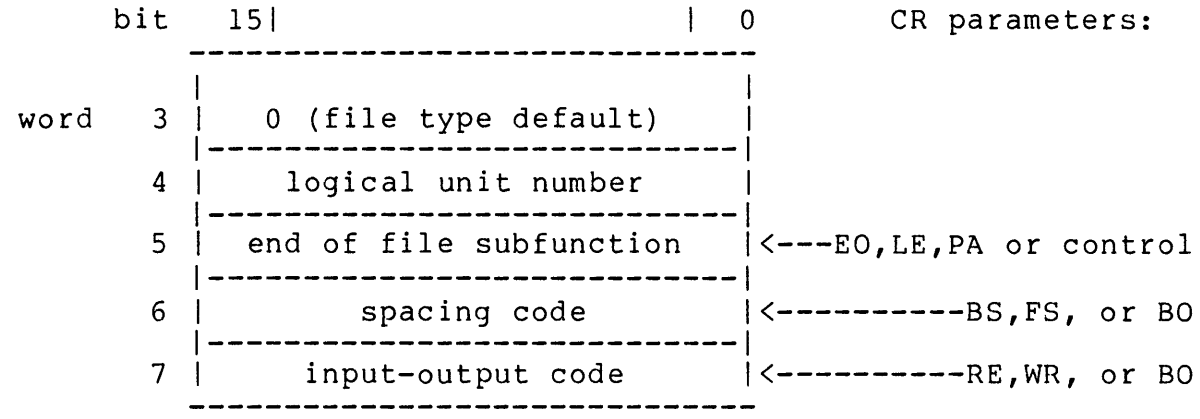
8100-13

Disc File Directory



Type 0 File Directory Entry

The entries for non-disc (type 0) files differ from those for disc files in words 3 through 7:



Words 5-7 are octal codes:

end-of-file subfunction = 01LU for MT(EO)
10LU for paper tape (LE)
11LU for line printer (PA)
or subfunction code

spacing code = bit 15 = 1 backspace legal (BS)
bit 0 = 1 forward space legal (FS)

input/output code = bit 15 = 1 input legal (RE)
bit 0 = 1 output legal (WR)

Appendix H
Session Monitor Tables

This appendix contains information on the following:

* SESSION CONTROL BLOCK (SCB)

* SESSION SWITCH TABLE (SST) AND CONFIGURATION TABLE

* SESSION TABLE RELATIONSHIP

Session Control Block (SCB)

A Session Control Block (SCB) is established for each user who has successfully "logged-on" to the system. The SCB contains the information necessary to identify the user to the system and describe his capabilities in terms of command processing and I/O addressing space.

The format of the SCB is shown in Figure H-1.

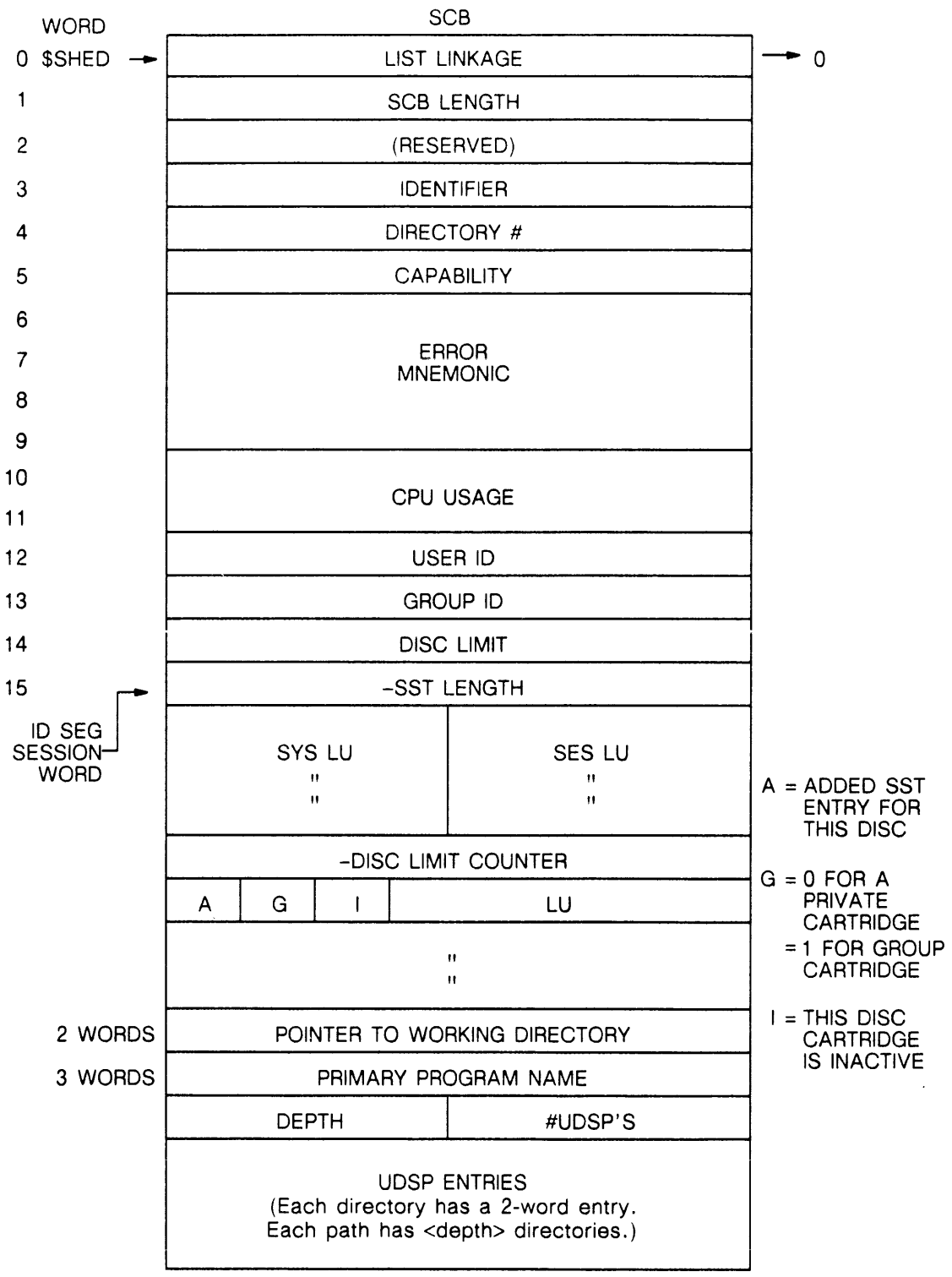


Figure H-1. SCB

Session Switch Table (SST) and Configuration Table

When operating in the session environment every I/O request is routed to the appropriate I/O device via the Session Switch Table (SST). Each SST entry describes a session LU, which the user addresses, and associated system LU where the I/O request will actually be directed. The SST describes the session user's I/O addressing capabilities by defining the system LUs the user has access to and the associated session LUs by which the user accesses them.

When the user makes an I/O request the SST is searched for the specified session LU. If the requested LU is found, it is switched to the associated system LU as specified in the SST entry and the I/O request is processed. Zero-offset addressing makes the internal representation of the LU in the Session Control Block (SCB) one less than the assigned number. If the requested LU is not found, an error is returned (IO12-LU not defined for this session).

The Session Switch Table is maintained in memory as part of the SCB. The format of the SST is shown in Figure H-2 below:

System LUs can be integer numbers between 1 and 255. Session LUs can be integer numbers between 1 and 63. Session LUs are assigned:

* at log-on, via user and group account file entries, or

* on-line using SL command (see RTE-6/VM Terminal User's Reference Manual), or

* at log-on, via Configuration Table entries.

The Configuration Table describes the default logical units to be used for specific device logical units. Each station (terminal) logical unit defined in the Configuration Table has associated with it a set of device logical units which are assigned default logical units to be used when a user logs on at this station (terminal). The default logical unit associated with the station itself is always 1.

At log-on, these default values are written from the Configuration Table in the account file into the user's Session Control Block (SCB), unless overridden by entries in this particular user's SST. The format of the Configuration Table is shown in Figure H-3, below.

Session Table Relationship

The interaction among the session tables is shown in Figure H-4.

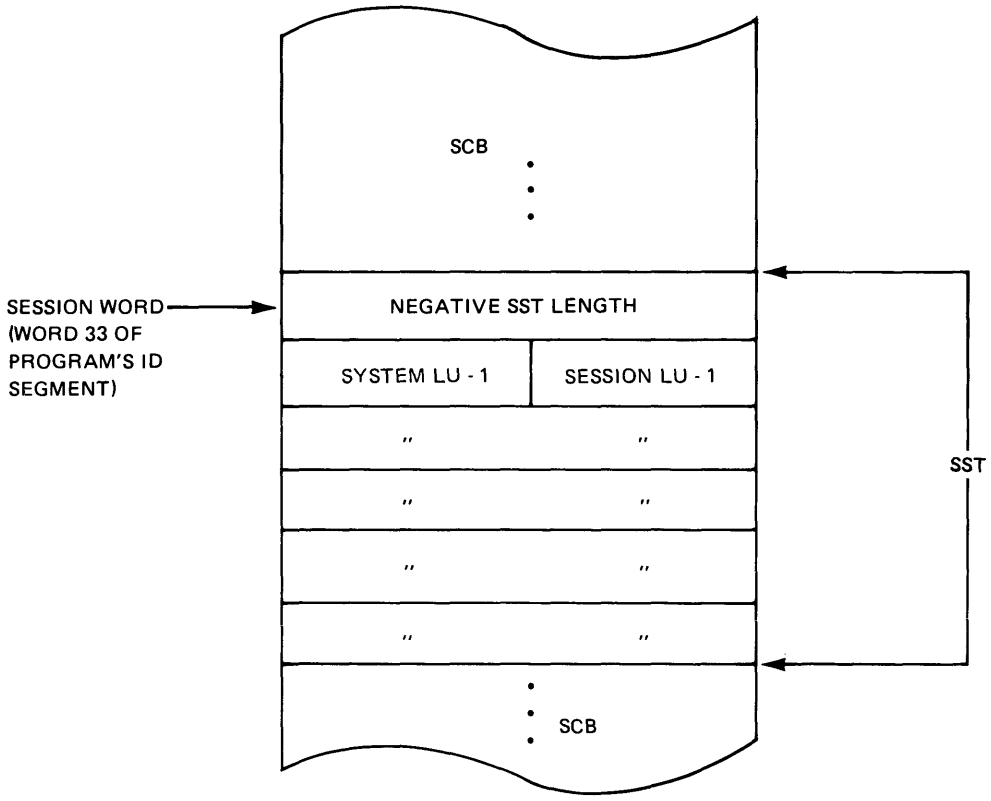


Figure H-2. Session Switch Table (SST) Format

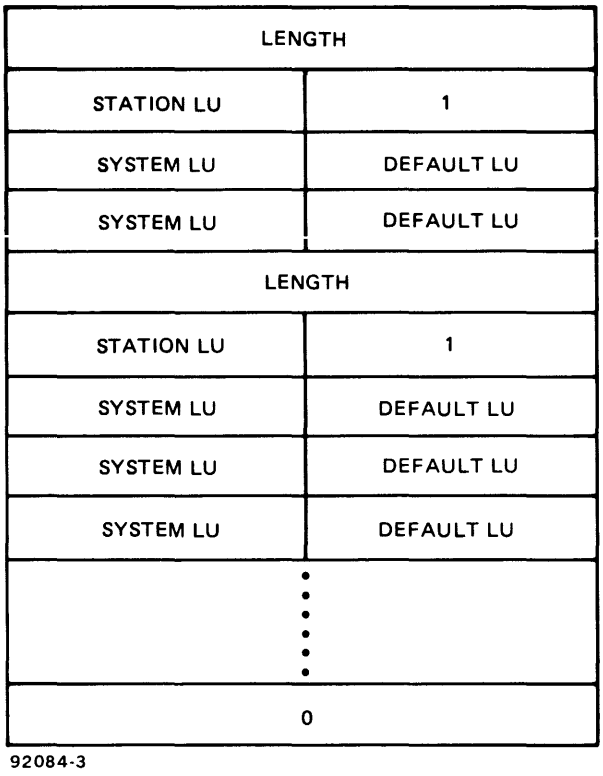


Figure H-3. Configuration Table

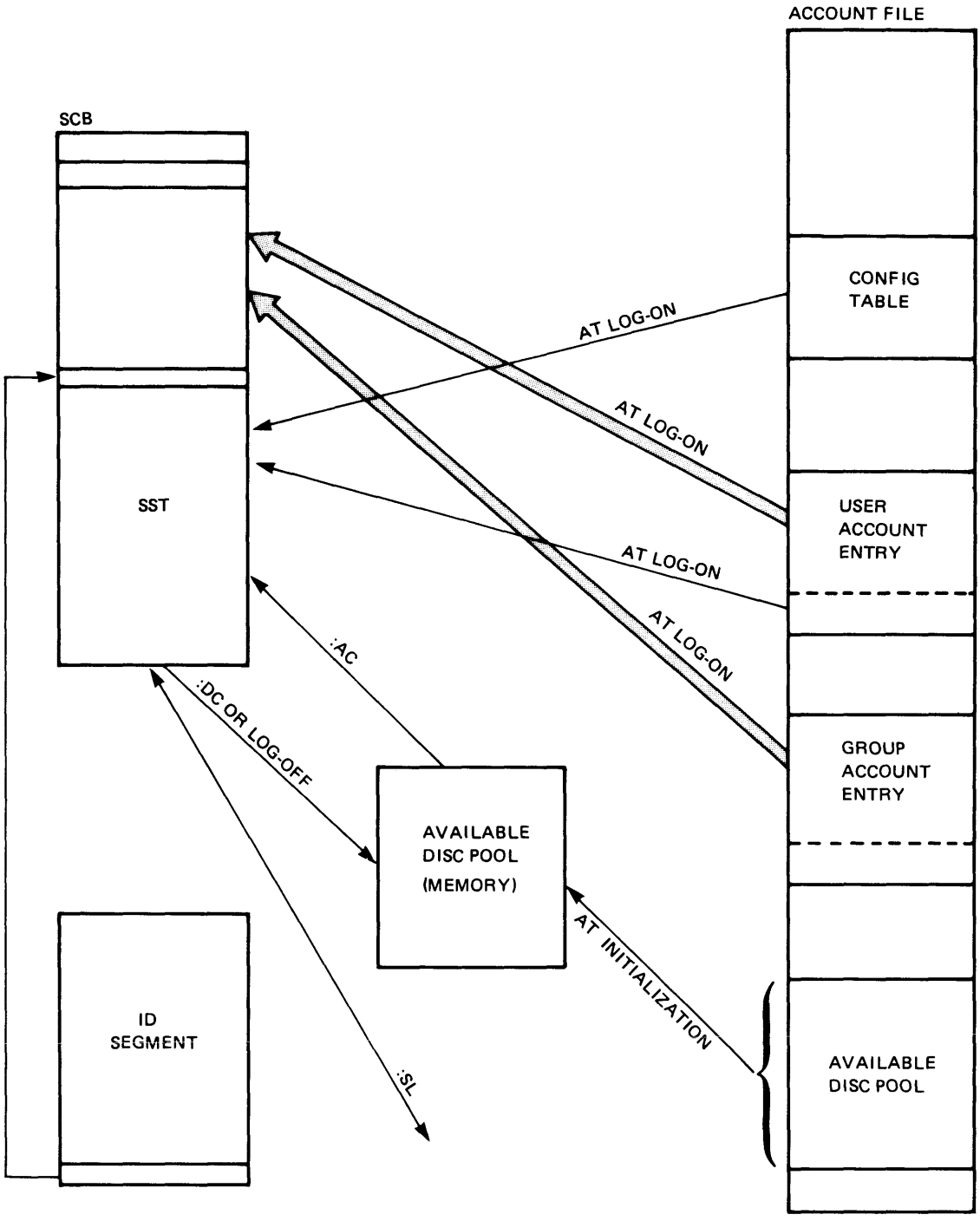


Figure H-4. Session Table Relationships

Appendix I
Record Formats

This Appendix contains information on the following:

* SOURCE RECORD FORMAT
* RELOCATABLE AND ABSOLUTE RECORD FORMATS
* ABSOLUTE TAPE FORMAT
* DISC FILE RECORD FORMATS
* SIO TAPE RECORD FORMATS
* MEMORY-IMAGE PROGRAM FILE FORMAT (TYPE 6)

Source Record Formats

The source format used for the disc records by the system program EDIT and FMGR is given in Figure I-1. All records are packed ignoring sector boundaries. Binary records are packed directly onto the disc. After an END record, a zero word is written and the rest of the sector is skipped. If this zero word is the first word of the sector, it is not written. Binary files are always contiguous so a code word is not required.

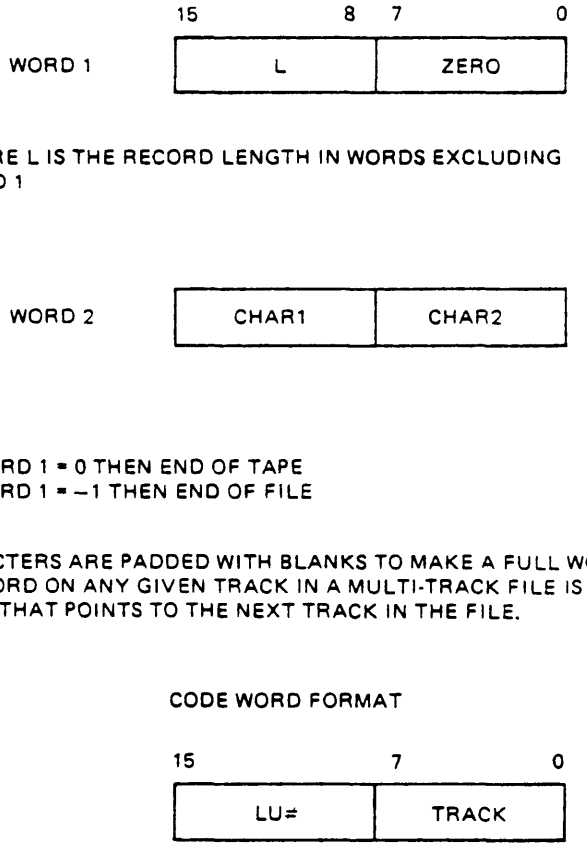


Figure I-1. Source Record Formats

Relocatable and Absolute Record Formats

The following describes the formats of relocatable and absolute records produced as object code for a given source program. The relocatable records are generated by compilers or by the assembler for a relocatable assembly. These records are stored in a relocatable file. The generator or the loader processes these relocatable records to produce an absolute module which has all program links resolved and the program is relocated and ready to run.

The absolute records are produced by the assembler for an absolute assembly. The module of records thus produced requires no processing by the generator or loader. Absolute programs must be loaded into memory and run off-line.

NAM RECORD

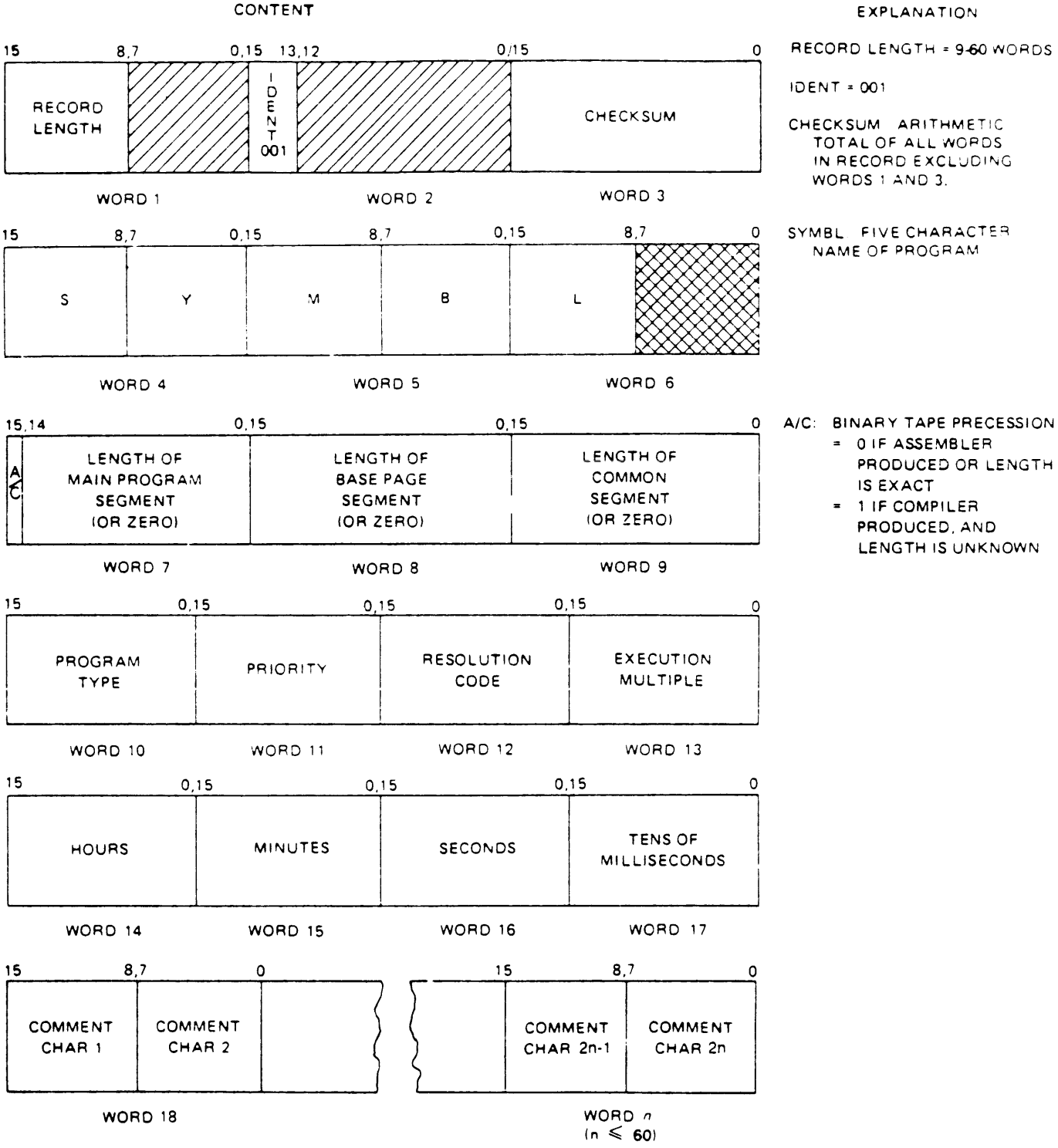


Figure I-2. Record Formats

ENT RECORD

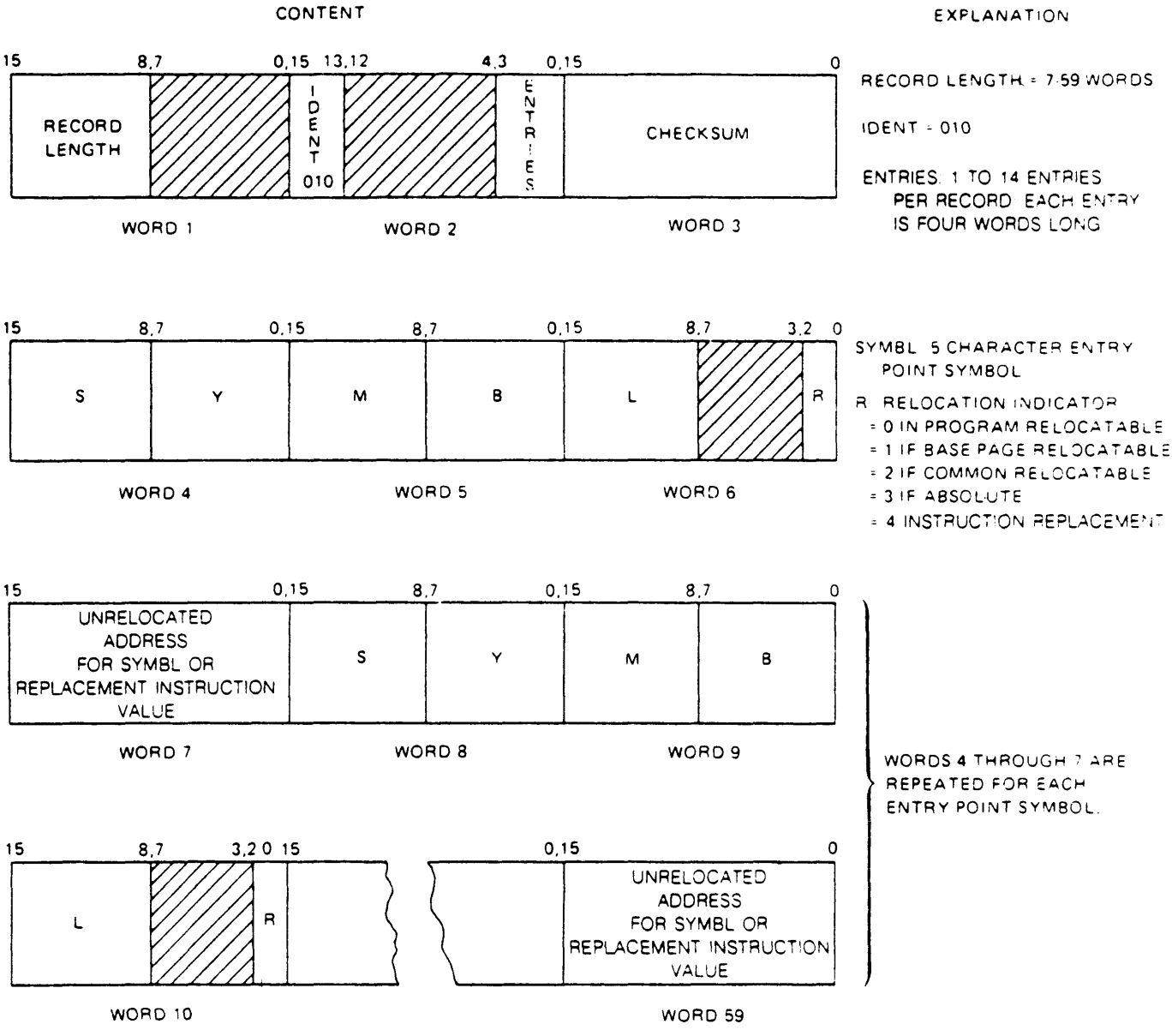


Figure I-2. Record Formats (continued)

EXT RECORD

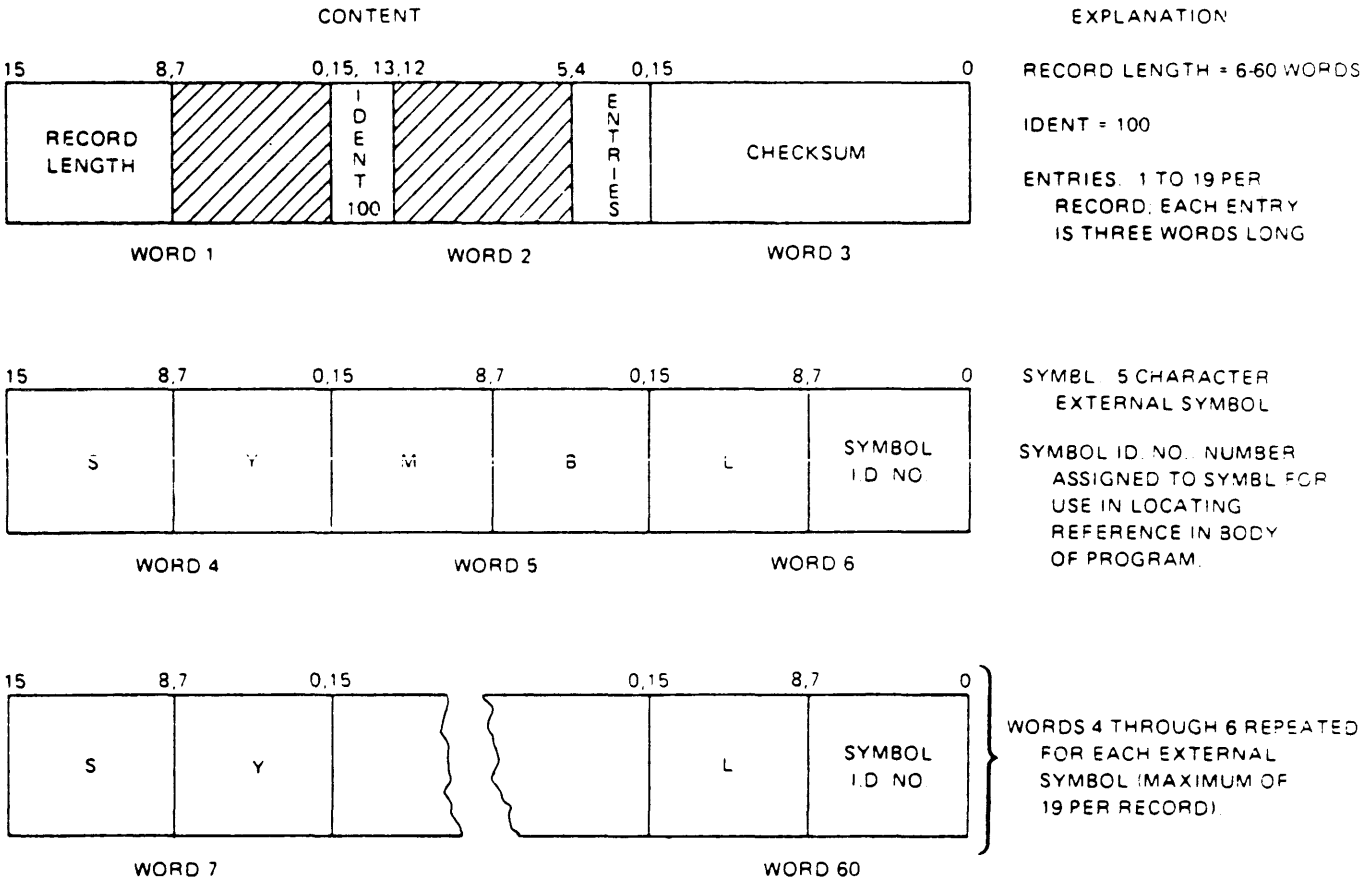
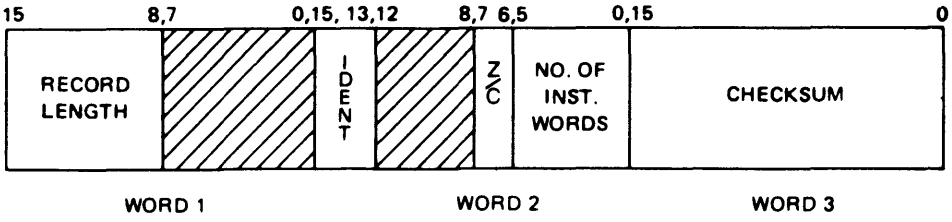
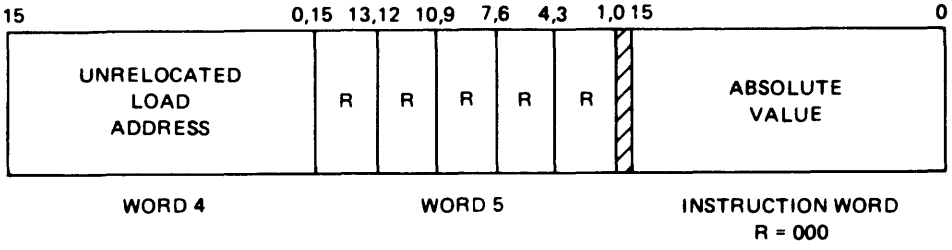


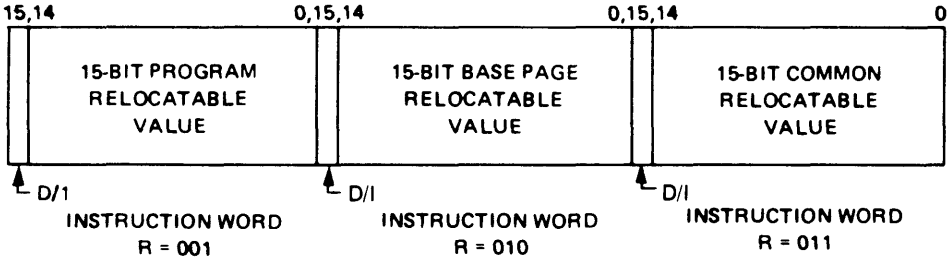
Figure I-2. Record Formats (continued)

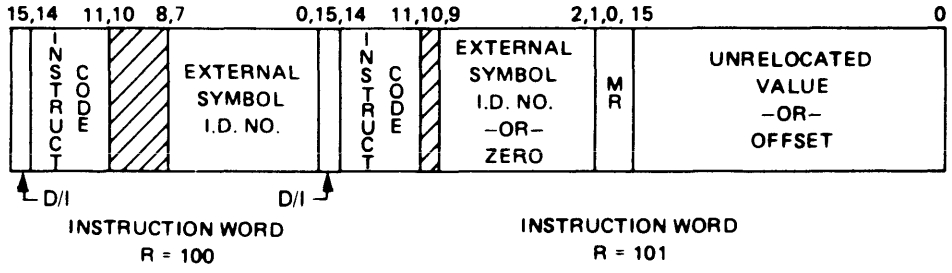
DBL RECORD

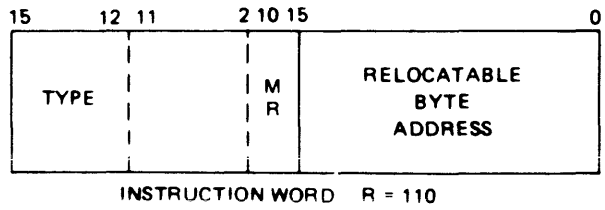
CONTENT











EXPLANATION

RECORD LENGTH = 6-60 WORDS
IDENT = 011
Z/C: RELOCATION OF LOAD ADDRESS
- = 0 FOR BASE PAGE
- = 1 FOR PROGRAM
- = 2 FOR ABSOLUTE
- = 3 FOR COMMON

NO. OF INST. WORDS: 1 TO 45 LOADABLE INSTRUCTION WORDS PER RECORD

RELOCATABLE LOAD ADDRESS: STARTING ADDRESS FOR LOADING THE INSTRUCTIONS WHICH FOLLOW;

R's: RELOCATION INDICATORS:
- 000 = ABSOLUTE
- 001 = 15-BIT PROGRAM RELOCATABLE
- 010 = 15-BIT BASE PAGE RELOCATABLE
- 011 = 15-BIT COMMON RELOCATABLE
- 100 = EXTERNAL REFERENCE
- 101 = MEMORY REFERENCE
- 110 = BYTE REFERENCE

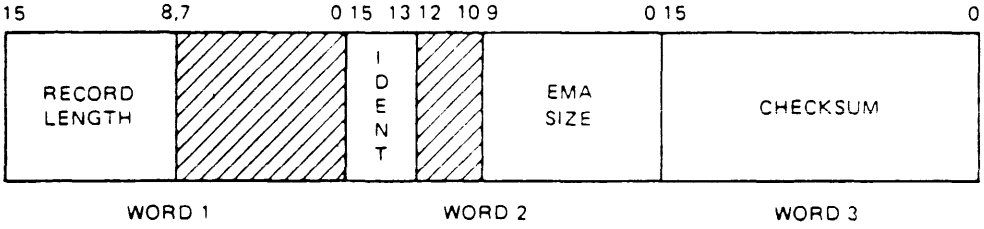
R_1 IS RELOCATION INDICATOR FOR INSTRUCTION $WORD_1$; R_2, FOR INSTRUCTION $WORD_2$; ETC.

D/I: INDIRECT ADDRESSING
- 0 = DIRECT
- 1 = INDIRECT

MEMORY REFERENCE INSTRUCTIONS USE TWO WORDS, WITHIN THE TWO-WORD GROUP?, "MR" INDICATES RELOCATABILITY OF OPERAND SPECIFIED IN SECOND WORDS:
- 00 = PROGRAM RELOCATABLE
- 01 = BASE PAGE RELOCATABLE
- 10 = COMMON RELOCATABLE
- 11 = ABSOLUTE

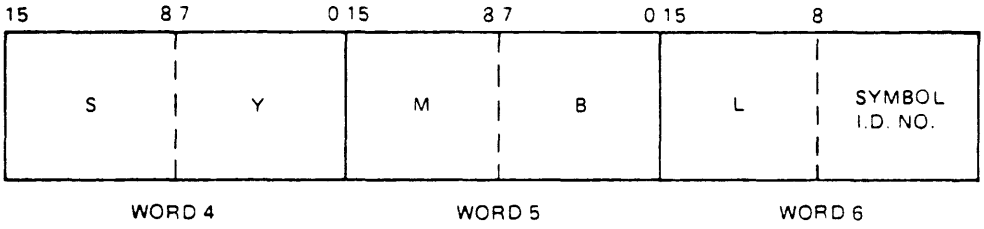
Figure I-2. Record Formats (continued)

EMA RECORD

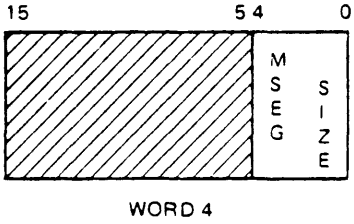


EXPLANATION

RECORD LENGTH = 7 WORDS
IDENT = 110

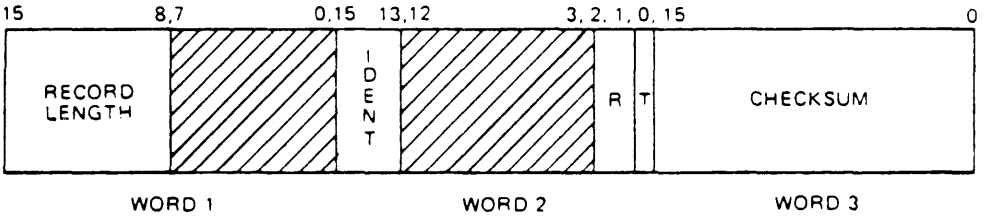


SYMBOL I.D. NO.. NUMBER ASSIGNED TO SYMBOL FOR USE IN LOCATING REFERENCE IN COPY OF PROGRAM.



END RECORD

CONTENT

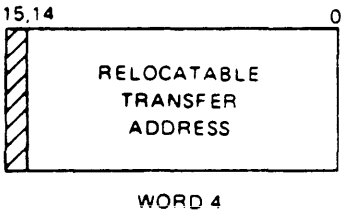


EXPLANATION

RECORD LENGTH = 4 WORDS
IDENT = 101

R: RELOCATION INDICATOR FOR TRANSFER ADDRESS

= 0 IF PROGRAM RELOCATABLE
= 1 IF BASE PAGE RELOCATABLE
= 2 IF COMMON RELOCATABLE
= 3 IF ABSOLUTE

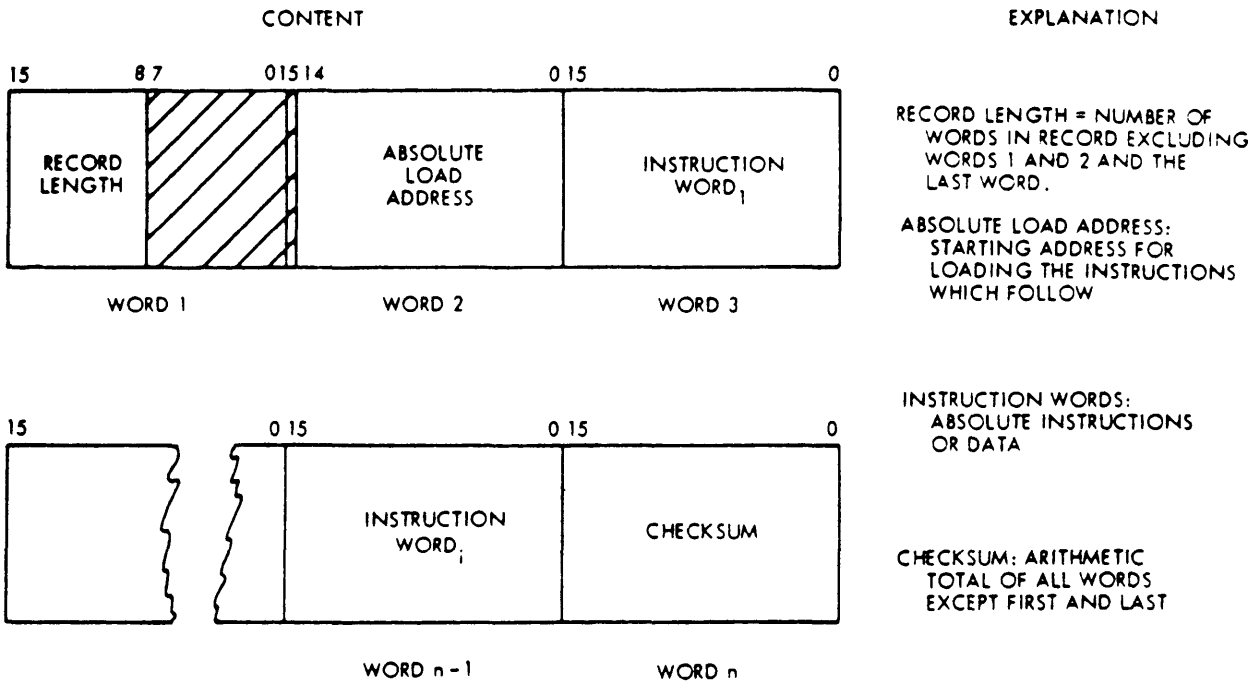


T: TRANSFER ADDRESS INDICATOR

= 0 IF NO TRANSFER ADDRESS IN RECORD
= 1 IF TRANSFER ADDRESS PRESENT

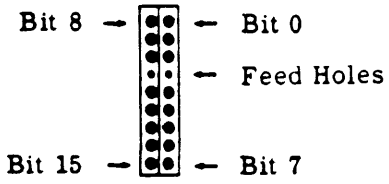
Figure I-2. Record Formats (continued)

Absolute Tape Format



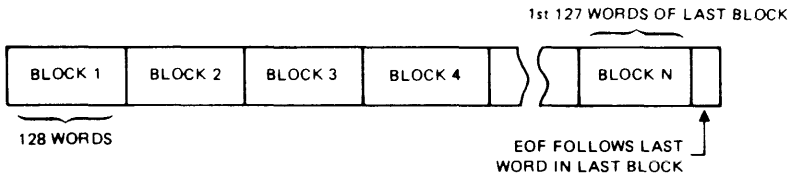
---

†On paper tape, each word represents two frames arranged as follows:



Disc File Record Formats

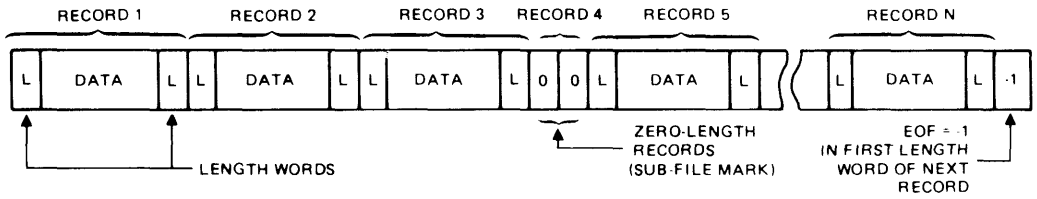
Fixed Length Formats (Types 1 and 2)



Type 1 Record length = Block length = 128 words

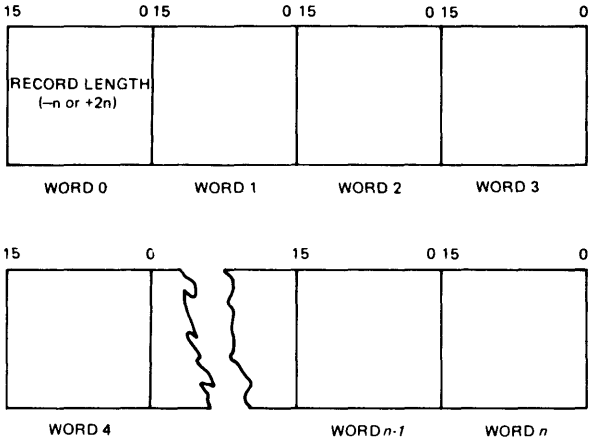
Type 2 Record length is user defined; may cross block boundaries but not past EOF

Variable Length Formats (Types 3 and Above)



SIO Record Format

Magnetic tape SIO binary records have the following format:

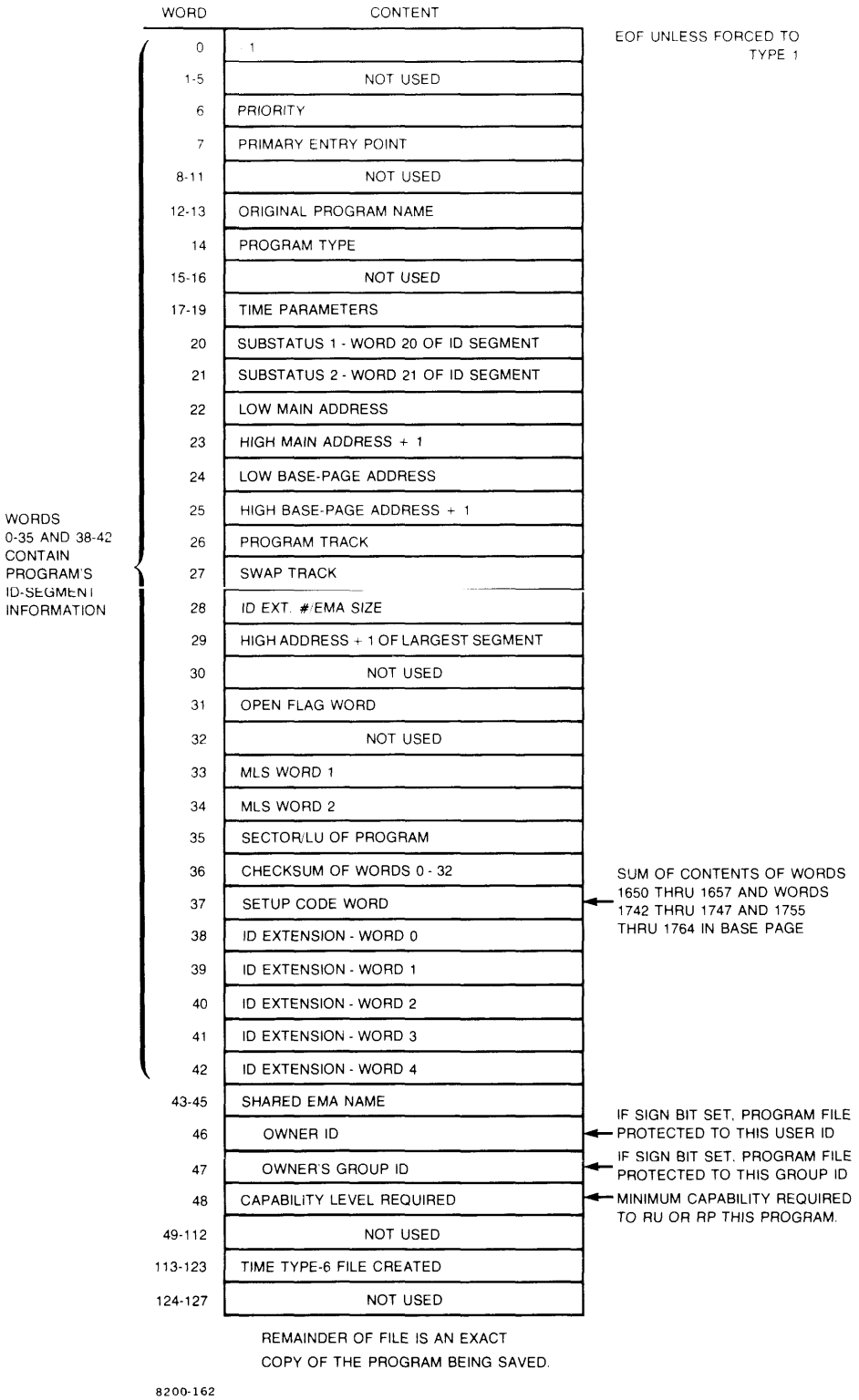


Record length = number of words or characters in record, excluding word 0; negative value denotes words, positive value denotes characters.

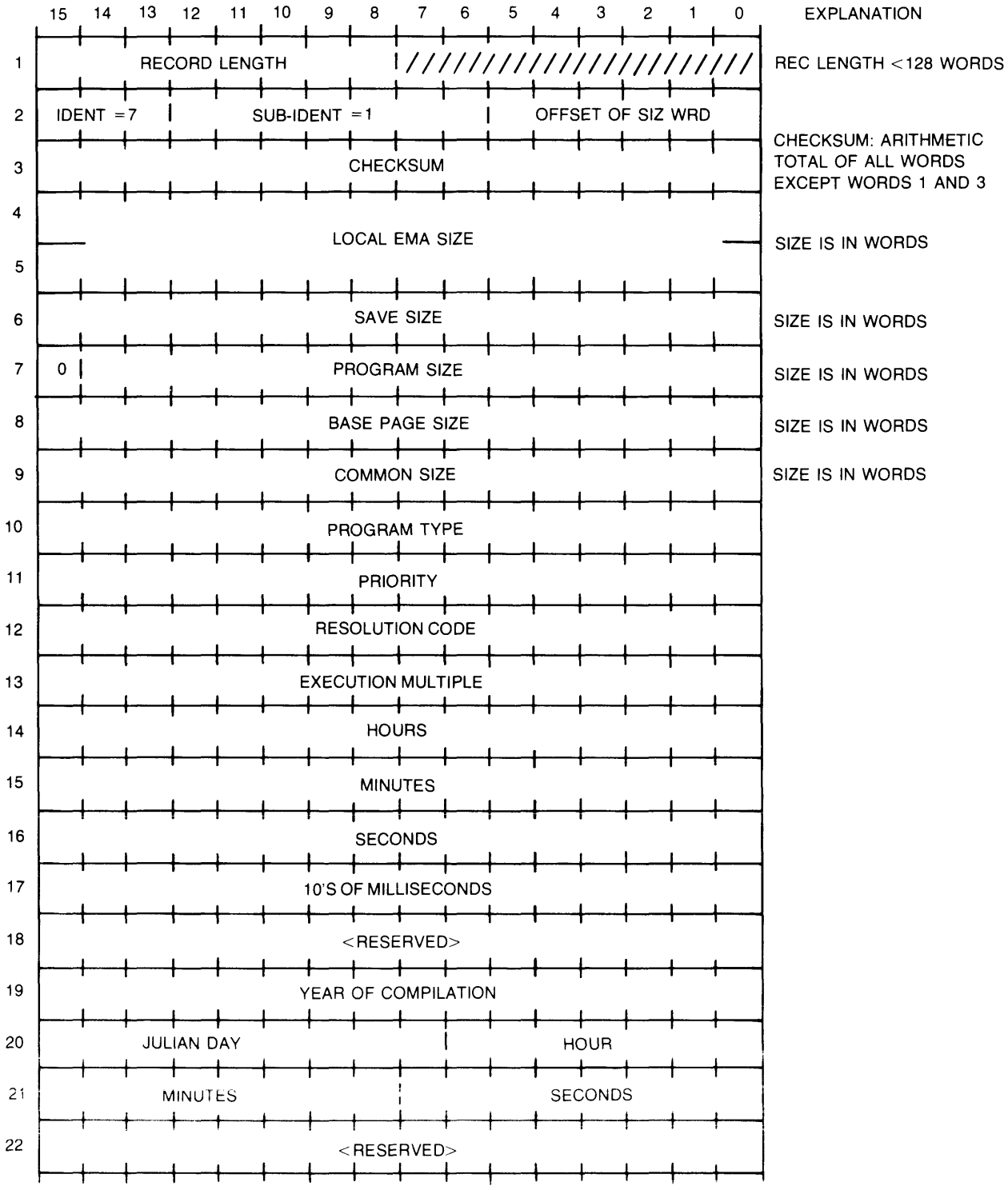
NOTE

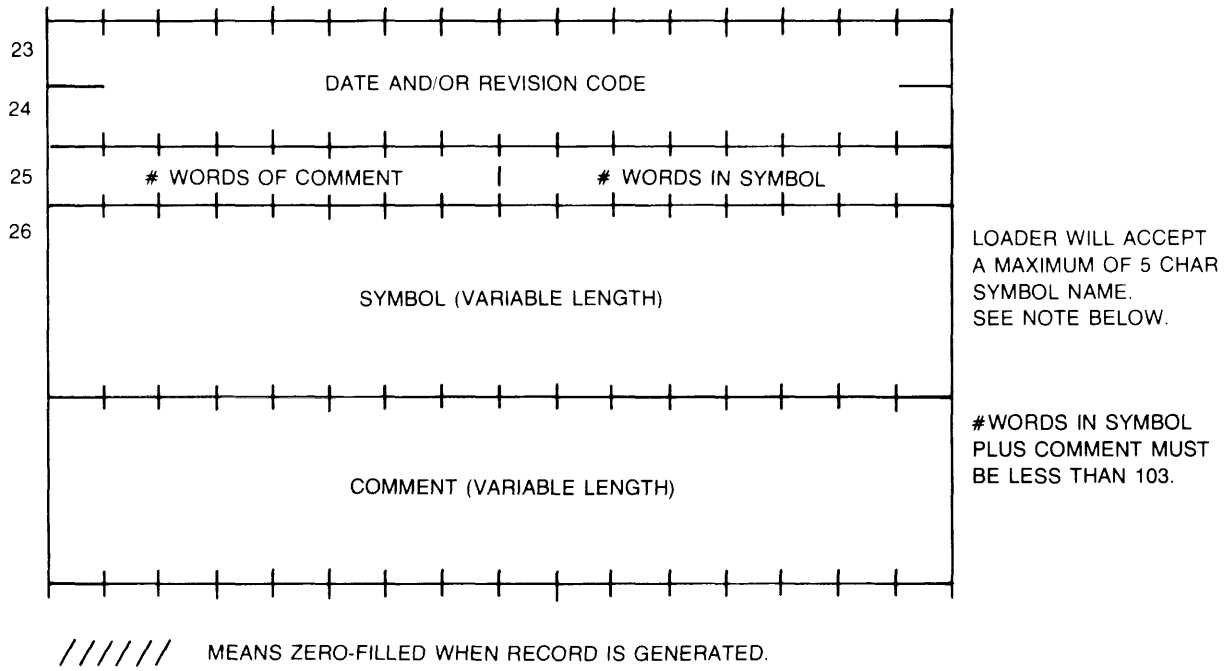
The length (word 0) is not considered part of the data record. When written with the MS option of the DU command, the length is supplied by FMGR. When read with the MS option of the ST command, the length is removed (in this case, the length word is used instead of the length supplied by the driver).

Memory-Image Program File Formats (Type 6)



EXTENDED NAM RECORD





NOTE

All Operating systems allow space for only five characters in main-program and segment names.

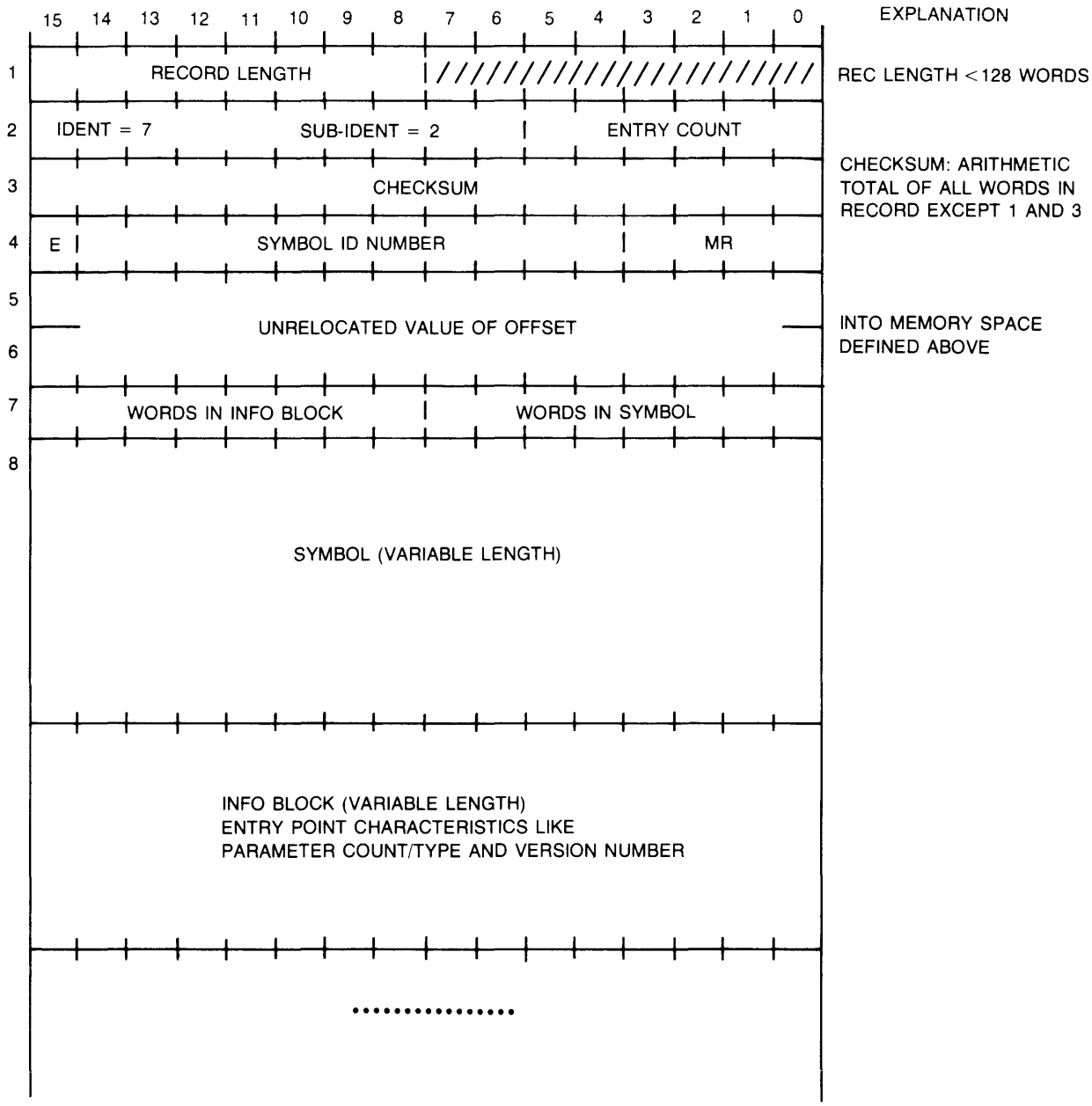
The loaders for RTE-6/VM and RTE-XL will handle symbols longer than five characters.

The loaders for RTE-4B, RTE-4E and earlier operating systems will not handle the longer symbols.

All system generators are limited to five-character symbols, module names and entry points.

Use of the OLDRE Utility is recommended to ensure conformance to these standards.

EXTENDED ENT RECORD

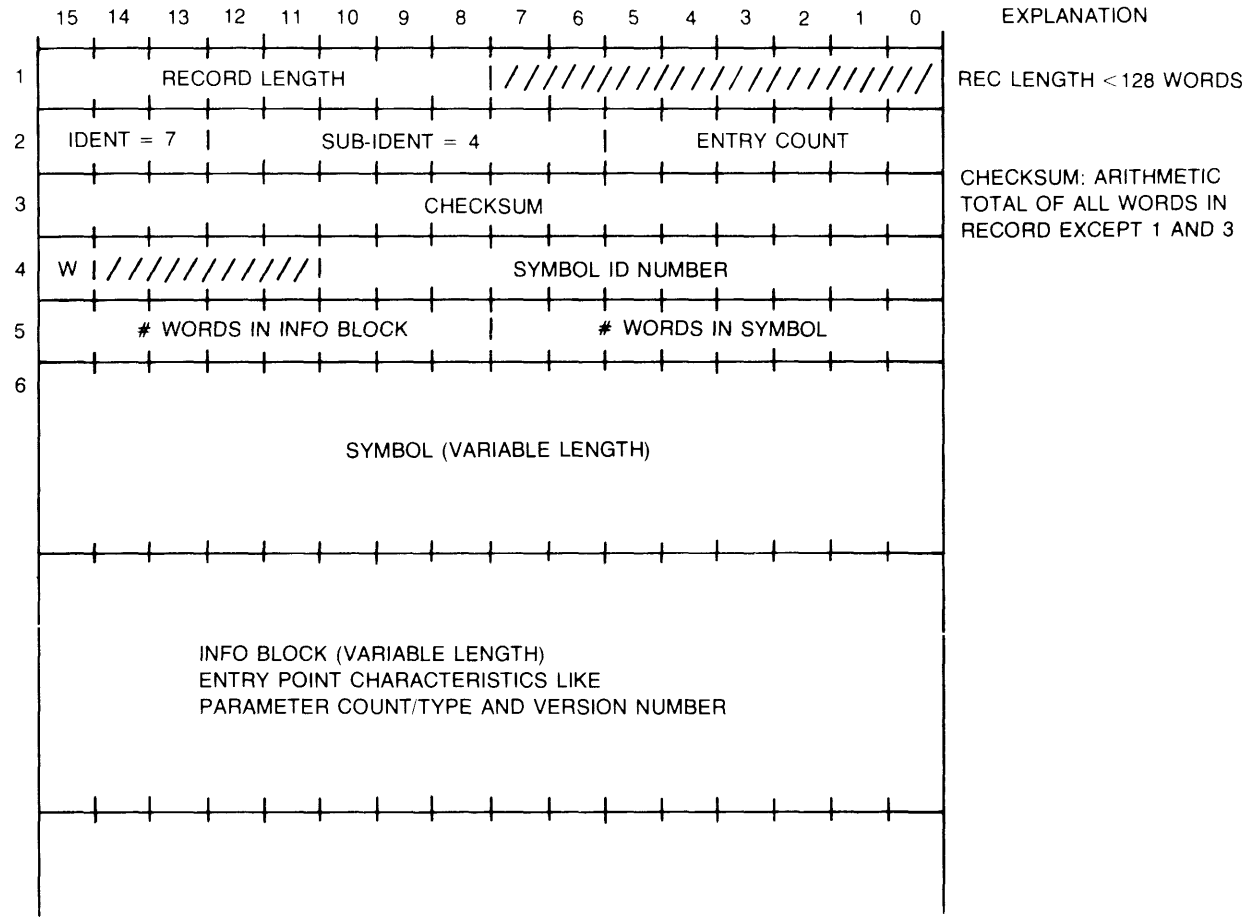


////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

E = 1, IF EMA
  = 0, IF MR

MR IS MEMORY SPACE
- = 0, IF ABSOLUTE SPACE
- = 1, IF PROGRAM RELOCATABLE SPACE
- = 2, IF BASE PAGE RELOCATABLE SPACE
- = 3, IF COMMON RELOCATABLE SPACE
- = 4, <RESERVED>
- = 5, IF EMA SPACE
- = 6, IF SAVE AREA SPACE

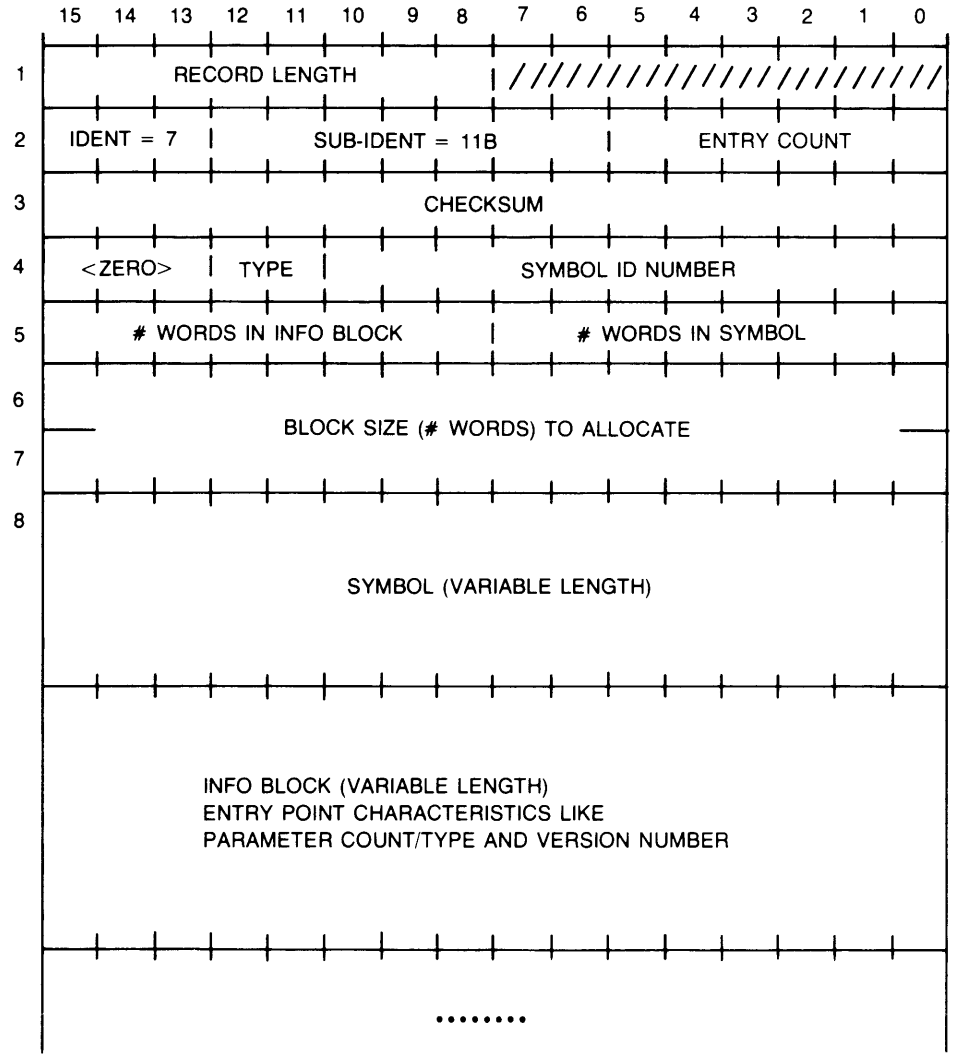
EXTENDED EXT RECORD



///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

W = 1, IF WEAK EXTERNAL
= 0, IF REGULAR EXTERNAL

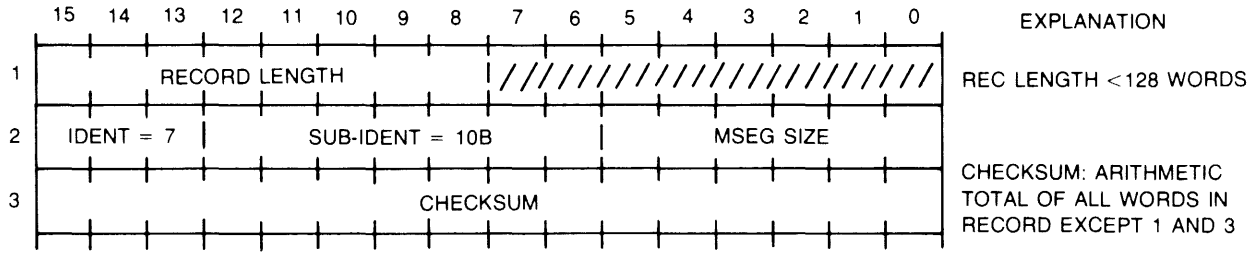
ALLOCATE RECORD



////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

TYPE = 0, IF NAMED COMMON (PROGRAM ALLOCATE)
1, IF NAMED SAVE COMMON (SAVE ALLOCATE)
2, IF NAMED EMA COMMON (EMA ALLOCATE)

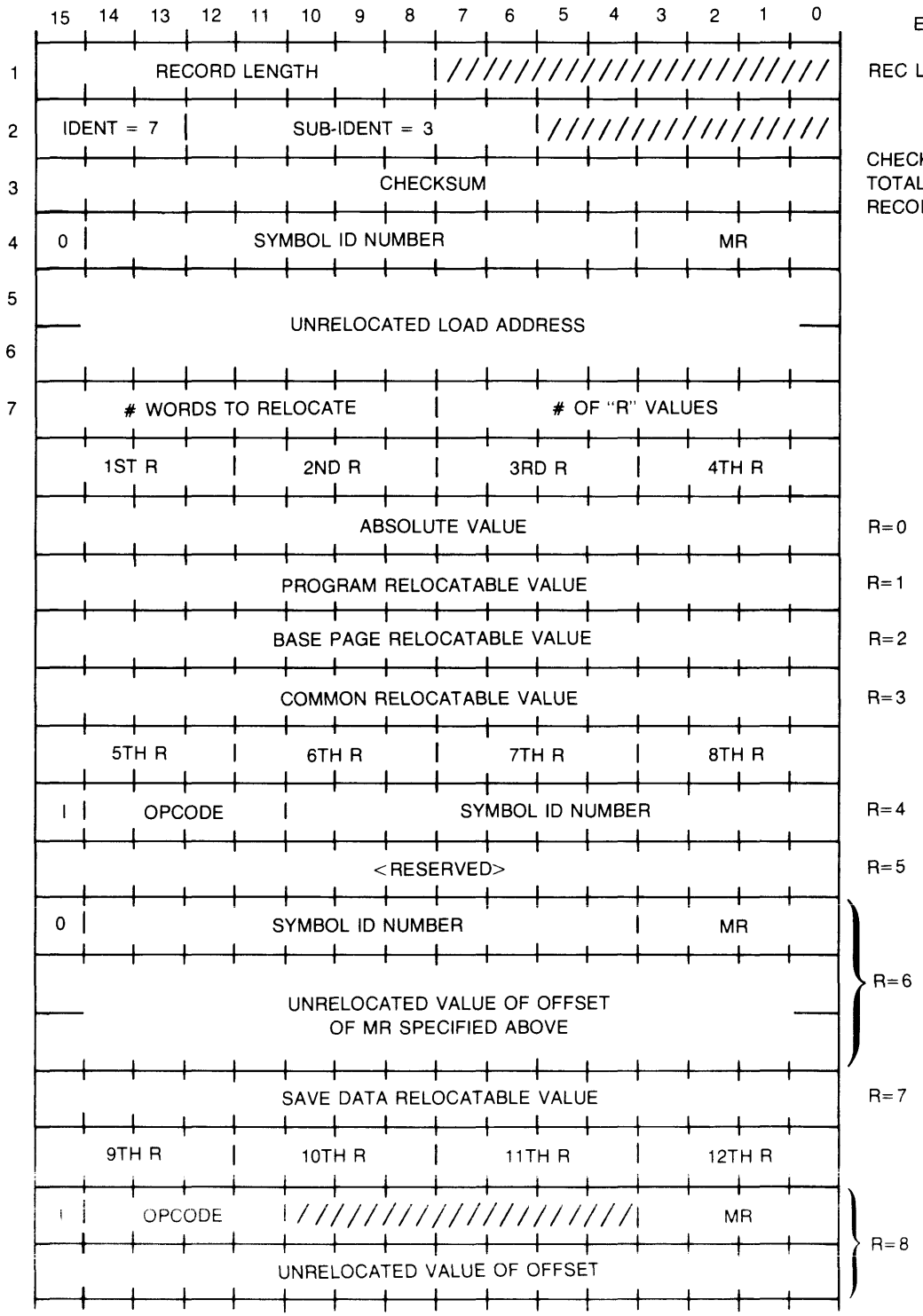
MSEG RECORD



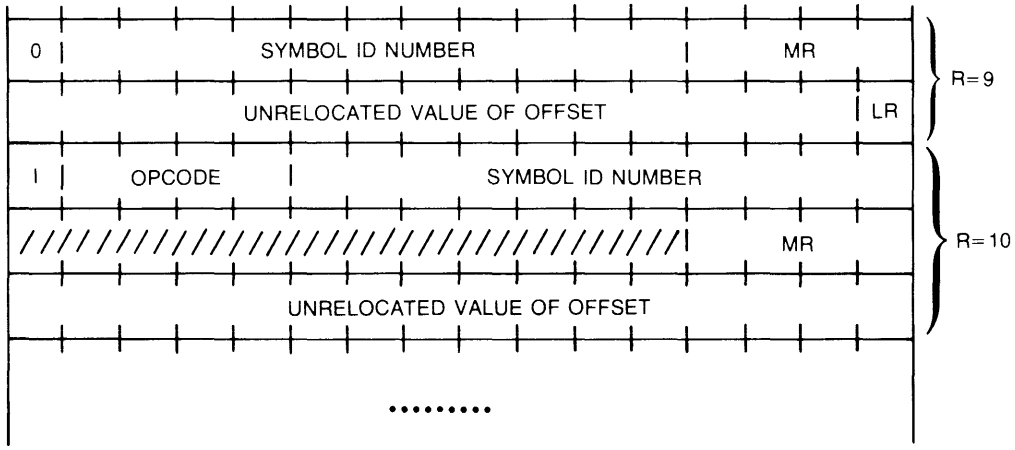
////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

MSEG SIZE IN PAGES, $1 \leq$ MSEG SIZE ≤ 32

EXTENDED DBL RECORD



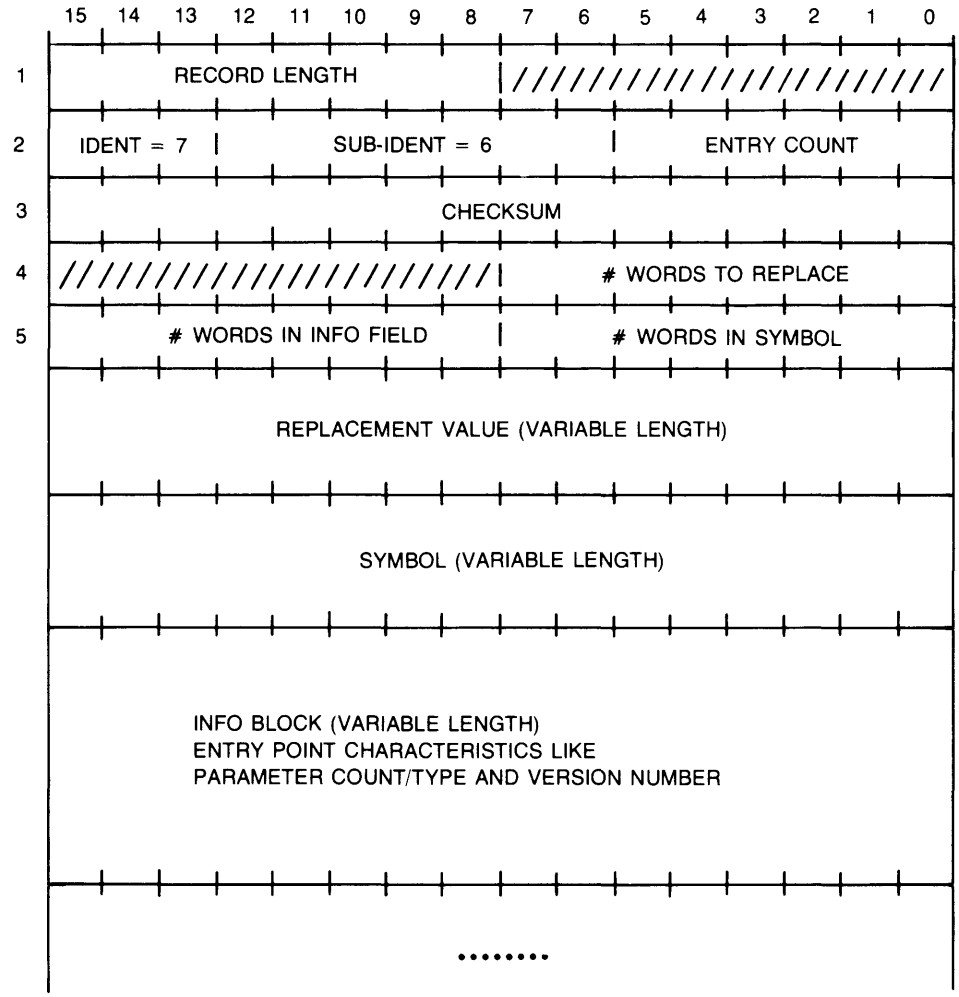
EXTENDED DBL RECORD (continued)



///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

MR IS MEMORY SPACE
- = 0, IF ABSOLUTE SPACE
- = 1, IF PROGRAM RELOCATABLE SPACE
- = 2, IF BASE PAGE RELOCATABLE SPACE
- = 3, IF COMMON RELOCATABLE SPACE
- = 4, <RESERVED>
- = 5, IF EMA SPACE
- = 6, IF SAVE AREA SPACE

RPL RECORD



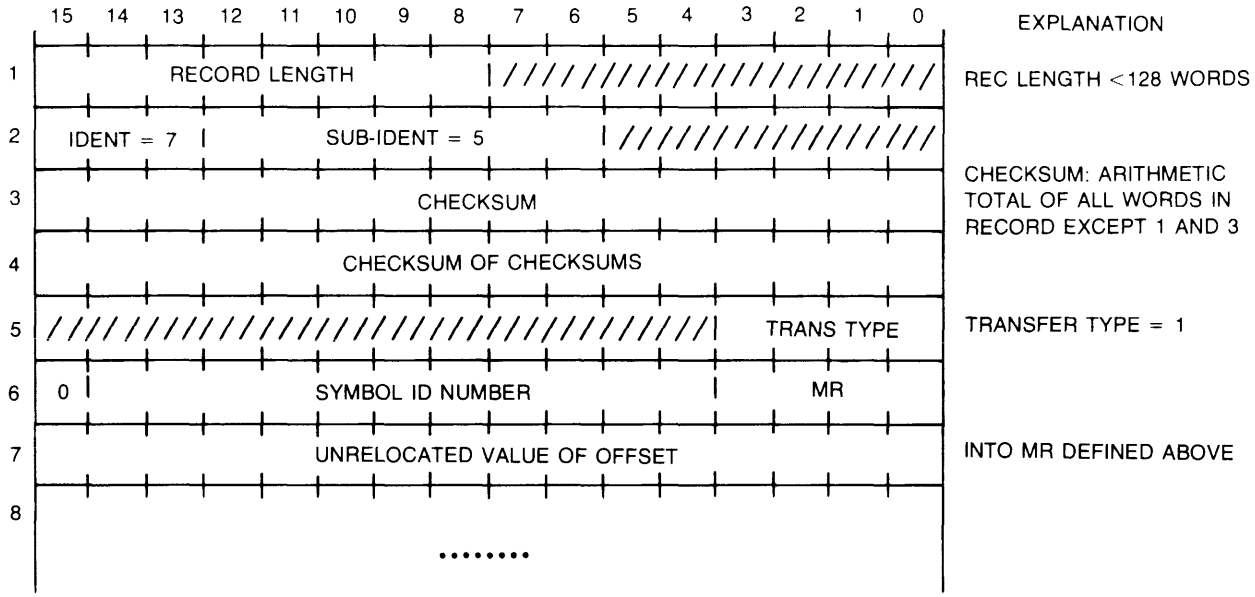
EXPLANATION

REC LENGTH <128 WORDS

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS IN RECORD EXCEPT 1 AND 3

////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

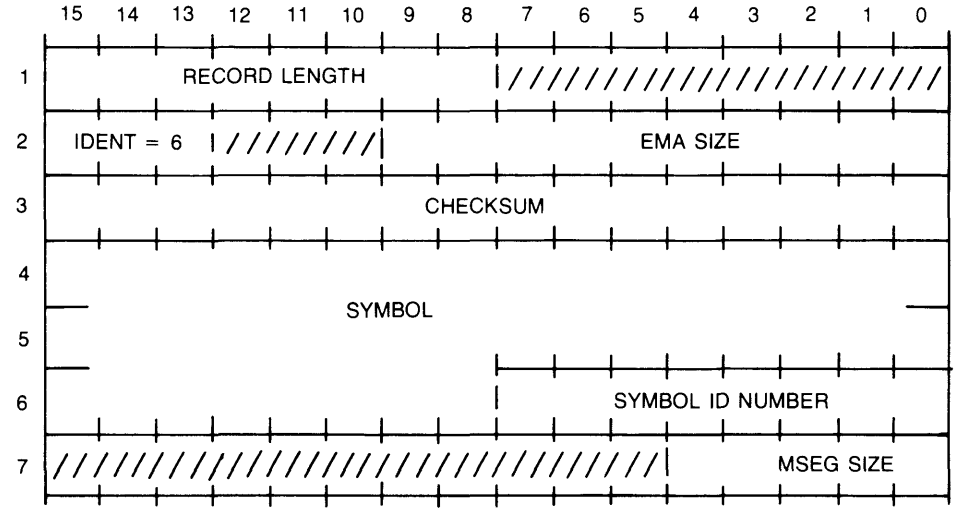
EXTENDED END RECORD



///// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

MR IS MEMORY SPACE
- = 0, IF ABSOLUTE SPACE
- = 1, IF PROGRAM RELOCATABLE SPACE
- = 2, IF BASE PAGE RELOCATABLE SPACE
- = 3, IF COMMON RELOCATABLE SPACE
- = 4, <RESERVED>
- = 5, IF EMA SPACE
- = 6, IF SAVE AREA SPACE

EMA RECORD



EXPLANATION

REC LENGTH = 7 WORDS

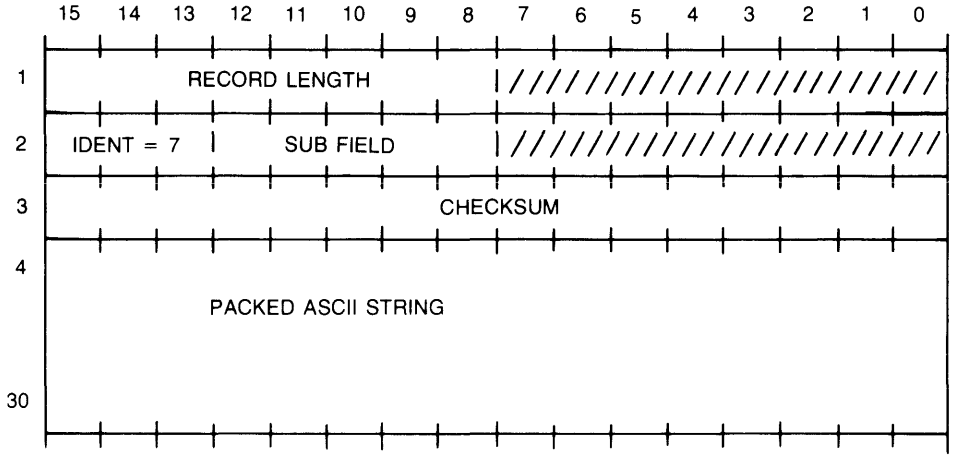
SIZE IN PAGES

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS IN RECORD EXCEPT 1 AND 3

////// MEANS ZERO-FILLED WHEN RECORD IS GENERATED.

MSEG SIZE IS IN PAGES, $1 \leq$ MSEG SIZE ≤ 32

LOADER/GENERATOR INFORMATION RECORD



EXPLANATION

REC LENGTH 4-60 WORDS

SUB FIELD = 0 GEN
= 5 LOD

CHECKSUM: ARITHMETIC TOTAL OF ALL WORDS IN RECORD EXCEPT 1 AND 3

Appendix J
Scheduling FMGR Programmatically

To request FMGR from a program use the following call. A command string can be optionally passed to FMGR. For more information on interactively scheduling FMGR, refer to the RTE-6/VM Terminal User's Reference Manual.

```
CALL EXEC(ICODE,6HFMGR  ,INP,LIST,ISV,LOG,IDUM,IBUFR,IBUFL)
                         --- ---- --- --- ---- ----- -----
CALL EXEC(ICODE,6HFMGR  ,2HXX,LIST,ISV,LOG,IDUM,IBUFR,IBUFL)
                         ---- ---- --- --- ---- ----- -----
```

ICODE - 23 to schedule FMGR queue with wait
- 24 to schedule FMGR queue without wait

INP - Input device where the FMGR commands will be entered. If INP is any two ASCII characters, input is entered from the file whose name appears in IBUFR; otherwise INP is the LU of the input device for FMGR commands.

LIST - LU of the device used to list results of FMGR commands; if omitted, LU 1 is assumed.

ISV - Severity code. Refer to the RTE-6/VM Terminal User's Reference Manual for details.

LOG - LU of the log device used to log and to correct any diagnosed errors; must be an interactive device. If omitted, the input device is assumed unless it is non-interactive, in which case, LU 1 is assumed.

IDUM - Dummy placeholder for the fifth optional parameter.

IBUFR - Optional buffer containing command string to be passed to FMGR or the name of a file in which FMGR commands are stored. Refer to the examples below.

IBUFL - Data buffer length; positive number of words or negative number of characters (bytes) to be passed from IBUFR to FMGR.

COMMENTS:

INP (input device)---If this parameter is omitted, LU 1 is assumed. If in a multi-terminal environment, the default input is the LU of the device where FMGR was scheduled. If the input device is an LU, IBUFR must contain a FMGR command which is to be executed before control is passed to that LU. Refer to example 2 below.

COMMAND STRING PASSAGE---The command string to be passed to FMGR via IBUFR must begin with a colon (:). The command string is ignored if:

a. The input device specified is not an interactive device.

b. The command string does not start with a colon.

FILE PASSAGE---If a file is being passed to FMGR, two commas must be placed before the file name as shown in example 1.

EXAMPLE 1: A program schedules FMGR (queue schedule with wait) passing it the procedure file HELLOO which contains a set of FMGR commands to be executed. The list and the log device are both set to LU 1.

```
      PROGRAM SHFM1
C
C SET UP THE PARAMETERS TO SCHEDULE FMGR
C
      DIMENSION IBUFR(4)
      DATA IBUFR/8H,,HELLOO/
      DATA LIST/1/,ISV/0/,LOG/1/,IBUFL/4/
C
C SCHEDULE FMGR WITH WAIT
C
      CALL EXEC(23,6HFMGR  ,2HXX,LIST,ISV,LOG,IDUM,IBUFR,IBUFL)
          :
```

EXAMPLE 2: A program schedules FMGR (queue with wait) and passes it the command string contained in IBUFR to schedule WHZAT.

```
      PROGRAM SHFM2
C
C SET UP THE PARAMETERS TO SCHEDULE FMGR
C
      DIMENSION IBUFR(2)
      DATA IBUFR/4H:WH /,INP/1/
      DATA LIST/1/,ISV/0/,LOG/1/,IBUFL/2/
C
C SCHEDULE FMGR
C
      CALL EXEC(23,6HFMGR  ,INP,LIST,ISV,LOG,IDUM,IBUFR,IBUFL)

          :
```

NOTE

To run these examples, first OFF your copy of FMGR, and then schedule the program from the break-mode prompt.

Index

$CVT1 (KCVT) subroutine, 5-70
$CVT3 (CNUMD,CNUMO) subroutine, 5-70
$EMA statement, 4-13
$EMCLB library, 4-12
$OPSY value, 5-55
$PARS subroutine, 5-77
.DRCT subroutine, 5-72
.EMIO subroutine, B-14
.ESEG subroutine, B-11
.IMAP subroutine, B-4
.IRES subroutine, B-6
.JMAP subroutine, B-7
.JRES subroutine, B-8
.LBP, .LBPR subroutine, B-12
.LPX, .LPXR subroutine, B-13
.STIO subroutine, 5-38

A

ABREG subroutine, 2-12
absolute record formats, I-3
absolute tape format, I-9
absolute time mode, 2-63
address translation, F-1
allocate class number CLRQ, 5-3
APOSN FMP call, 3-64
ASCII parse subroutine
   $PARS, 5-77
   PARSE, 5-77
automatic buffering, 1-18
automatic track switching, 2-84
auxiliary system cartridges, 3-12

B

background COMMON, 1-12
Backing Store File, 4-2, 4-3, 4-5
   close, 4-32
   create, 4-27
   initialized, 4-29
   open, 4-28
   purge, 4-30
base page and logical memory, F-3
base page links, 1-10
Basic/1000C, 1-26
Basic/1000D, 1-26

binary to ASCII conversion subroutines, 5-70
BINRY subroutine, 5-39
buffer conversion subroutine, 5-74
buffer passage father/son, 2-68
buffering
   automatic, 1-18
   I/O, 1-18
buffers
   packing (DCB), 3-17
   user, 3-17, 3-23

C

Cartridge Directory, 3-12
cartridge directory format, G-4
Cartridge Reference Number (CRN), 3-7
cartridges, 3-7
   auxiliary system, 3-12
   disc, 1-25
   global, 3-13
   group, 1-25, 3-12
   in the session environment, 3-12
   initialization, 3-9
   non-session, 3-12
   organization, 3-8
   pool, 1-25
   private, 1-25, 3-12
   system, 1-25, 3-12
class, 2-28
Class Control, EXEC 19, 2-39
Class Get EXEC 21, 2-41
class I/O, 1-18
   EXEC calls, 2-27
class I/O applications, 5-3
   mailbox I/O, 5-10
   multiple terminals, 5-7
class I/O operation, 2-30
class I/O requests, 2-28
   flush - CLRQ, 5-5
class I/O terms
   class, 2-28
   class number, 2-28
   class request, 2-28
   completed class queue, 2-28
   pending class requests, 2-28
class number, 2-28, 2-29
   allocation - CLRQ, 5-3
   allocation CLRQ subroutine, 2-29
class ownership, 5-4, 5-5
   CLRQ, 5-3
Class Read EXEC 17, 2-33

Class Write EXEC 18, 2-33
Class Write/Read EXEC 20, 2-33
CLOAD - compile and load utility, 1-31
CLOSE FMP call, 3-44
CLRQ subroutine - Class I/O Management, 5-3
CLSVM subroutine, 4-32
CMD - help utility, 1-34
command string passage, J-2
COMMON, 1-10
   areas, 1-12
   background, 1-12
   labeled, 4-13
   local, 1-12
   parameters, 3-21
   real-time, 1-12
COMPL - compile utility, 1-31
completed class queue, 2-28, 2-29
configuration table, H-3
conversion routines, 5-70
COR.A subroutine, 5-64
COR.B subroutine, 5-64
CREAT call, 3-29
CRETS call, 3-34
CREVM subroutine, 4-26
current time subroutine FTIME, 5-73

D

D.RTR, 3-10
Data Control Block (DCB), 3-3, 3-15, 3-21, 3-80
   legend, G-3
Data Control Block (DCB) Format, G-2
data transfer, 3-17
DBUGR - interactive debugger, 1-32
DCPC (dual channel port controller), 1-8
deadly embrace, 5-26
declaring a shareable EMA, 4-8
declaring extended memory area (EMA), 4-13
declaring VMA array, 4-3
Device Reference Table (DRT), 2-74, E-6
device time-out, 1-18
directory entry type 0 files, G-7
disc allocation error messages, A-8
disc cartridges, 1-25
disc errors, 3-9
disc file directory, G-6
disc file record formats, I-10
disc parity error (track error), A-5
disc read/write subroutine BINRY, 5-39
dispatching, 1-3
dispatching errors, A-3

DM error, A-4
driver mapping table (DMT), E-9
driver partition, 1-10
DRREL - on-line driver replacement utility, 1-34
DRRPL - on-line driver replacement utility, 1-34
dual channel port controller (DCPC), 1-8
dynamic mapping system, 1-6
dynamic mapping violations, A-2

E

EAPOS FMP call, 3-64
ECLOS FMP call, 3-44
ECREA call, 3-29
EDIT/1000 - interactive editor, 1-32
EIOSZ subroutine, 4-22
ELOCF FMP call, 3-61
EMA, 4-6
EMA and memory structure, 4-6
EMAST subroutine, 4-16
EPOSN FMP call, 3-67
EQLU subroutine, 5-41
equipment table (EQT), 1-20, 2-74, E-1
EREAD FMP call, 3-50
error codes
   DM, A-4
   EX, A-3
   LU lock, A-13
   MP, A-4
   RE, A-5
   RQ, A-5
   TI, A-5
   TR, A-5
   VMA/EMA, A-33
errors, A-1
   disc allocation, A-8
   dispatching, A-3
   EXEC other, A-8
   executive, A-1
   executive halt, A-14
   I/O call, A-10
   I/O format, A-12
   parity, A-6
   program management, A-13
   routing, A-16
   VMA/EMA, 4-12, A-33
EWRIT FMP call, 3-56
EX errors, A-3
EXEC 14, 5-36

EXEC call
error returns, 2-9
formats, 2-3, 2-4, 2-5, 2-6, 2-7, 2-8
standard, 2-24
summary, 2-1
EXEC call error messages, A-1
EXEC calls
EXEC 1 - standard I/O read, 2-19
EXEC 2 - standard I/O write, 2-19
EXEC 3 - I/O control call, 2-24
EXEC 4 - local disc track allocation, 2-85
EXEC 5 - local disc track release, 2-87
EXEC 6 - program completion, 2-50
EXEC 7 - program suspend, 2-53
EXEC 8 - program segment load, 2-55
EXEC 9 - immediate schedule with wait, 2-57
EXEC 10 - immediate schedule without wait, 2-57
EXEC 11 - time request, 2-72
EXEC 12 - program time scheduling, 2-63
EXEC 13 - I/O status, 2-74
EXEC 14 - sample program, 5-36
EXEC 14 - string passage, 2-67
EXEC 15 - global disc track allocation, 2-85
EXEC 16 - global disc track release, 2-87
EXEC 17 - class read, 2-33
EXEC 18 - class write, 2-33
EXEC 19 - class I/O control, 2-39
EXEC 20 - class write/read, 2-33
EXEC 21 - class get, 2-41
EXEC 22 - program swapping control, 2-70
EXEC 23 - queue schedule with wait, 2-57
EXEC 24 - queue schedule without wait, 2-57
EXEC 25 - partition status, 2-79
EXEC 26 - memory size, 2-81
EXEC description conventions, 2-16
EXEC errors other, A-8
EXEC processor, 2-1
executive communication, 1-27, 2-1
executive halt errors, A-14
Extended Memory Area (EMA), 4-1, 4-2, 4-6
declaring, 4-13
I/O transfers, 4-20
programming, 4-12
shareable EMA, 4-7
size, 4-18

F

father program, 2-57
father/son buffer passage, 2-68
FC - file backup utility, 1-32

FCONT FMP call, 3-73
file access, 3-5
   random, 3-49
   sequential, 3-49
File Directory, 3-10, G-5
file management, 3-1
File Management Package (FMP), 1-28, 3-1
file management system, 1-28
file manager (FMGR), 1-28, 3-1
   scheduling programmatically, J-1
file passage, J-2
file security, 3-11
   code, 3-11, 3-24
file types, 1-28, 3-2
   0 - non-disc devices, 3-2
   1 - fixed-length (128-word records), 3-3
   2 - fixed-length (user-defined record length), 3-3
   3 - variable-length records, 3-3
   4 - ASCII data, 3-4
   5 - relocatable binary code, 3-4
   6 - memory image code, 3-4
   7 - absolute binary code, 3-4
   greater than 7, 3-31
   greater than 7 - user-defined data format, 3-4
file updates, 3-6
files, 1-28, 3-1
   definition FMP calls, 3-28
   extents, 3-5
   names, 3-23
   non-update mode, 3-6
   positioning, 3-60
   protection, 3-24
   scratch, 3-28
   sequential, 3-64
   truncation, 3-45
   update mode, 3-6
flush class request, 5-5
flush class requests on LU, 5-5
FMP call parameters, 3-15
   IBUF, 3-23
   ICR, 3-25
   IDCB, 3-21
   IDCBS, 3-80
   IDCBZ, 3-26
   IERR, 3-23
   INAM, 3-23
   ISC, 3-24
FMP calls, 3-15
   APOSN, 3-64
   CLOSE, 3-44
   CREAT, 3-29
   CRETS, 3-34

description conventions, 3-27
EAPOS, 3-64
ECLOS, 3-44
ECREA, 3-29
ELOCF, 3-61
EPOSN, 3-67
EREAD, 3-50
EWRIT, 3-56
examples using, 3-84
FCONT, 3-73
formats, 3-19
FSTAT, 3-76
IDCBS, 3-80
LOCF, 3-61
NAMF, 3-81
OPEN and OPENF, 3-37
POSNT, 3-67
POST, 3-82
PURGE, 3-47
READF, 3-50
RWNDF, 3-71
special purpose, 3-72
summary, 3-16
WRITF, 3-56
FMP error, 3-23
FMP Tracks, 3-12
FORTRAN, 1-26, 2-4, 2-5
FSTAT FMP call, 3-76
FTIME subroutine, 5-73
function codes, 2-25, 3-74

G

GENIX - help utility, 1-34
get SCB error mnemonic, 5-48
get session capability, 5-51
GETST subroutine, 5-35
global cartridges, 3-13
global tracks, 2-84
group cartridges, 1-25, 3-12
GTERR subroutine, 5-48
GTSCB subroutine, 5-50

H

halt errors executive, A-14
hard parity errors, A-6
hardware considerations, 1-19
HELP - help utility, 1-34

I

I/O and swapping, 2-22
I/O buffering, 1-18
I/O call error codes, A-10
I/O control call - EXEC 3, 2-24
I/O controller, 1-19
   time-out, 1-21
I/O device, 1-19
I/O error message format, A-12
I/O EXEC calls, 1-20
   class I/O, 2-27
   standard I/O, 2-18
I/O processing, 1-17
I/O requests and related routines, 5-38
I/O select codes, 1-19
I/O status - EXEC 13, 2-74
I/O Tables and Processing, E-1
I/O transfers to/from the VMA/EMA, 4-20
I/O without wait, 2-27
IBUF call, 3-23
ICAPS subroutine, 5-51
ICR call, 3-25
ID extension, 4-18
ID segment, 1-6, 2-57, 2-80, 4-18
ID segment extensions, C-10
ID segments short, C-11
IDCB call, 3-21
IDCBS call, 3-80
IDCBZ call, 3-26
IDGET subroutine, 5-66
IERR call, 3-23
IFBRK subroutine, 5-67
IFTTY subroutine, 5-42
immediate schedule with or without wait, 2-57, 2-58
INAM call, 3-23
indirect address subroutine, 5-72
INDXR - indexed relocatable library utility, 1-32
initial offset mode, 2-63
initialization of cartridges, 3-9
initialized virtual memory, 4-25
INP (input device), J-2
INPRS subroutine, 5-74
interactive LU query, 5-42
interrupt
   handling with microcode, E-13
   handling without microcode, E-11
   privileged, 1-17
   processing, 1-17
   table, E-11

interrupting LU query, 5-41
ISC call, 3-24

K

KEYS - soft key program, 1-33
KYDMP - soft key program, 1-33

L

labeled common block, 4-13
language
   Basic/1000C, 1-26
   Basic/1000D, 1-26
   FORTRAN, 1-26
   Macro/1000, 1-26
   Micro-Assembler, 1-26
   Pascal/1000, 1-26
language support, 1-25
LGTAT - track assignment table log program, 1-33, 2-84
LIMEM subroutine, 5-68
LINK - indexing utility, 1-32
LINK - relocating linker, 1-32
LKEMA - lock a shareable EMA partition, 4-7, 4-23
LOADR - relocating loader, 1-32
local COMMON, 1-12
local tracks, 2-84
LOCF FMP call, 3-61
logical memory, 1-6, 4-2
logical memory and base page, F-3
logical unit (LU)
   lock, 1-18, 5-20
   lock applications and resource numbers, 5-14
   lock error codes, A-13
   lock subroutine LURQ, 5-20
   numbers, 1-20
   session, 1-20
   system, 1-20
LOGLU subroutine, 5-43
LU query
   interactive, 5-42
   interrupting, 5-41
   terminal, 5-47
LURQ subroutine, 5-20
LUSES subroutine, 5-52
LUTRU subroutine, 5-44

M

Macro/1000, 1-26, 2-8
mailbox I/O, 2-27
Mapping Segment (MSEG), 4-2
master security code, 3-11
memory allocation table entry, F-7
memory lock, 2-70
memory management, 1-6
memory management and related tables, F-1
memory maps, 1-6, 1-7
memory protect fence, 1-13, 1-14
memory protect violations, A-2
memory protection, 1-13
memory size - EXEC 26, 2-81
memory-image form, 3-1
memory-resident
   library, 1-11
   programs, 1-7, 1-11
MERGE - file merge utility, 1-32
message files, 3-14
message processor interface, 5-75
MESSS subroutine, 5-75
Micro-Assembler, 1-26
miscellaneous, 5-72
MLS-LOC loader (MLLDR), 1-16, 1-31
MMAP subroutine, B-9
mother partitions, 1-15, 2-79, 2-80, 4-9
   swapping, 4-10
MP error, A-4
multilevel segmentation (MLS), 1-16
multiprogramming and timeslicing, 1-3

N

NAMF FMP call, 3-81
no-abort option, 2-9
no-suspend I/O option, 2-14
non-disc devices, 3-2
non-session cartridges, 3-12
non-update mode, 3-6
normal completion, 2-50

O

OPEN FMP calls, 3-37
   exclusive, 3-40
   file options, 3-39

flags, 3-41
non-exclusive, 3-40
update, 3-41
OPENF FMP calls, 3-37
defaults, 3-40
operating system, 3-12
operating system code partitions, 1-11
OPNVM subroutine, 4-28
OPSYS subroutine, 5-55

P

Page Fault, 4-2
Page Table (PTE), 4-2, 4-3
parameter passage applications, 5-30
parameter string recovery, 5-35
parameters common, 3-21
parity errors
hard, A-6
soft, A-7
PARSE subroutine, 5-77
partition status - EXEC 25, 2-79
partitions
background, 1-15
considerations, 4-9
mother, 1-15, 4-9
real-time, 1-15
shareable EMA, 1-15, 4-7, 4-9, 4-11
subpartitions, 4-10
Pascal/1000, 1-26, 2-6, 2-7
passage
command string, J-2
file, J-2
string - EXEC 14, 2-67
PCOPY - physical backup utility, 1-33
pending class request, 2-29
physical memory, 1-6, 1-8, 4-2
physical read or write, 3-18
pool cartridges, 1-25
port maps, 1-8
positioning type 0 files, 3-68
positioning type 1 and 2 files, 3-68
positioning type 3 and above files, 3-69
POSNT FMP call, 3-67
POST FMP call, 3-82
power fail, 1-21
power fail/auto restart, E-13
priority program, 1-3
private cartridges, 1-25, 3-12
privileged fence, 1-22

privileged interrupts, 1-17
   processing, 1-22
procedure files, 3-14
program completion - EXEC 6, 2-50
program ID segment, C-5
program management error codes, A-13
program managment EXEC calls, 2-49
program related tables, D-1
program scheduling - EXEC 9,10,23, and 24, 2-57
program segment load - EXEC 8, 2-55
program states, D-4
   disc suspended, D-5
   dormant, D-4
   general wait, D-4
   I/O suspended, D-4
   memory suspended, D-5
   operator suspended, D-5
   scheduled, D-4
   user diagram, D-6
program status routines, 5-64
program suspend - EXEC 7, 2-53
program swapping control - EXEC 22, 2-70
program time scheduling, 2-63
program types summary, D-2
program-to-program communication, 5-10
programs
   father and son, 2-57
   partitions, 1-15
   priority, 1-3
   segmentation, 1-16
   swapping, 2-22
   types, 1-6
PRSTR - physical backup utility, 1-33
PRTM subroutine, 5-33
PRTN subroutine, 5-33
PSAVE - physical backup utility, 1-33
PSTVM subroutine, 4-31
PTERR subroutine, 5-53
PURGE FMP call, 3-47
PURVM subroutine, 4-30

Q

queue schedule with or without wait, 2-57, 2-58

R

random file access, 3-49
RE error, A-5
re-entrant I/O, 1-18

read call - EXEC 1, 2-19
READF FMP call, 3-50
READT - disc cartridge save restore utility, 1-33
real-time COMMON, 1-12
   area, 1-10
record formats, I-1
records, 3-1
recover parameter string, 5-35
reentrant I/O subroutine REIO, 5-45
reformat current time, 5-80
REIO subroutine, 5-45
relocatable record formats, I-3
resource management, 1-23
   subroutine RNRQ, 5-16
resource numbers and logical unit lock applications, 5-14
resource numbers (RN), 1-23, 5-16
return parameter subroutines, 5-33
RMPAR - Retrieve Parameters, 5-31
RNRQ subroutine, 5-16
   allocate options, 5-19
   set options, 5-19
RQ error, A-5
RT6GN - on-line generator, 1-32
RWNDF FMP call, 3-71

S

saving-resources completion, 2-51
schedule call error codes, A-8
scheduled list, 1-3
scheduling, 1-3
scheduling FMGR programmatically, J-1
SCOM - source file comparison utility, 1-33
scratch files, 3-28, 3-34
security codes
   file, 3-24
   files, 3-11
   master, 3-11
SEGLD subroutine, 5-56
   with link, 5-56
   with loader, 5-57
segmentation
   multilevel, 1-16
   programmatically, 1-16, 2-55
   techniques, 1-16
SEGRT subroutine, 5-59
select codes I/O, 1-19
self time scheduling, 2-65
send message to system console, 5-79

sequential files, 3-64
   access, 3-49
serially-reusable completion, 2-50
Session Control Block (SCB), 1-24, H-1
session LU, 1-20, 1-24
Session Monitor, 1-20, 1-24, 3-13
Session Switch Table (SST), 1-24, H-3
session table relationship, H-4
SESSN subroutine, 5-54
SETTM subroutine, 5-61
SGMTR - MLLDR command file utility, 1-31
SH loader command, 4-8
shareable EMA
   declaration, 4-8
   label, 4-7
   label file, 4-8
   program considerations, 4-8
   using, 4-7
shareable EMA partitions, 1-15, 2-80, 4-7, 4-9, 4-11
   lock - LKEMA subroutine, 4-23
   unlock - ULEMA subroutine, 4-23
short ID segments, C-11
SIO Record Format, I-11
soft parity errors, A-7
son program, 2-57
source record formats, I-1
spooling system, 1-30
standard I/O EXEC calls, 2-18
standard I/O request flow, E-16
status EXEC calls, 2-72
string passage - EXEC 14, 2-67
subchannel, 1-20
subpartitions, 4-10
subroutines
   ABREG, 2-12
   LKEMA, 4-7
   optional parameters, 3-24
   return parameter, 5-33
   ULEMA, 4-7
   VMA/EMA, 4-12
   VMA/EMA mapping management, B-1
subsystem global area (SSGA), 1-10, 1-12
suspend program - EXEC 7, 2-53
swapping mother partition, 4-10
swapping control - EXEC 22, 2-70
swapping programs, 2-22
SXREF - MLLDR command file utility, 1-31
SYCON subroutine, 5-79
syntax conventions, 5-2
system, 1-11
   base page, 1-10
   cartridge directory, 3-14

cartridges, 1-25, 3-12
driver area, 1-11
library, 1-29, 3-12
map, 1-7
scratch tracks, 3-12
System Available Memory (SAM), 1-11, 2-27
system calls and entry points, 5-55
system communication area, C-1
system disc layout RTE-6/VM, C-11
system library subroutines
CLRQ, 5-3
GETST, 5-35
LURQ, 5-20
PRTM, 5-33
PRTN, 5-33
RMPAR, 5-31
RNRQ, 5-16
system LU, 1-20, 1-24
System Manager, 3-12, 3-13
system routines, 5-1

T

table area I, 1-10
table area II, 1-11
tables program related, D-1
TATMP subroutine, 5-63
terminating
normal, 2-50
saving-resources completion, 2-51
serially-reusable, 2-50
test break flag subroutine IFBRK, 5-67
TF - tape filer, 1-33
TI error, A-5
time quantum, 1-4
time request - EXEC 11, 2-72
time scheduling
other programs, 2-64
self, 2-65
timeslicing and multiprogramming, 1-3
TMVAL subroutine, 5-80
TR error, A-5
track allocation - EXEC 4 or 15, 2-85
Track Assignment Table (TAT), 2-84
track management EXEC calls, 2-84
track release - EXEC code 5 or 16, 2-87
track switching automatic, 2-84
tracks
global, 2-84
local, 2-84
trap cells, E-11

TRMLU subroutine, 5-47
type 0 file directory entry, G-7
type 0 files
   non-disc devices, 3-2
   positioning, 3-68
type 1 files
   access, 3-41
   fixed-length (128-word record), 3-3
   positioning, 3-68
   read, 3-52
type 2 files
   fixed-length (user-defined record length), 3-3
   positioning, 3-68
type 3 and above files positioning, 3-69
type 3 files variable-length records, 3-3
type 4 files ASCII data, 3-4
type 5 files relocatable binary code, 3-4
type 6 files memory image code, 3-4
type 7 files absolute binary code, 3-4
types greater than 7, user-defined data format, 3-4

U

ULEMA - unlock a shareable EMA partition, 4-7, 4-23
update mode, 3-6, 3-29
update open, 3-41
update SCB error mnemonic, 5-53
user buffer, 3-17, 3-23
user map, 1-7
user partitions, 1-11
utility programs, 1-31
   CLOAD, 1-31
   CMD, 1-34
   COMPL, 1-31
   DBUGR, 1-32
   DRREL, 1-34
   DRRPL, 1-34
   EDIT/1000, 1-32
   FC, 1-32
   GENIX, 1-34
   HELP, 1-34
   INDXR, 1-32
   KEYS, 1-33
   KYDMP, 1-33
   LGTAT, 1-33
   LINDX, 1-32
   LINK, 1-32
   LOADR, 1-32
   MERGE, 1-32
   MLLDR, 1-31
   PCOPY, 1-33

PRSTR, 1-33
PSAVE, 1-33
READT, 1-33
RT6GN, 1-32
SCOM, 1-33
SGMTR, 1-31
SXREF, 1-31
TF, 1-33
WHZAT, 1-33
WRITT, 1-33

V

virtual memory
backing store file, 4-2, 4-3
declaring VMA array, 4-3
I/O transfers, 4-20
initialized, 4-25
mapping segments (MSEG,VSEG), 4-2
page fault, 4-2
page table, 4-2, 4-3
programming, 4-12
size, 4-18
working set, 4-2, 4-3
Virtual Memory Area (VMA), 4-1, 4-2, 4-3
Virtual Memory Mapping Segment (VSEG), 4-2, 4-5
VMA subroutines
CLSVM, 4-32
CREVM, 4-26
example, 4-37
LKEMA, 4-23
OPNVM, 4-28
PSTVM, 4-31
PURVM, 4-30
ULEMA, 4-23
VREAD, 4-33
VWRIT, 4-35
VMA/EMA errors, 4-12
VMA/EMA mapping subroutines, B-1
.EMIO, B-14
.ESEG, B-11
.IMAP, B-4
.IRES, B-6
.JMAP, B-7
.JRES, B-8
.LBP, .LBPR, B-12
.LPX, .LPXR, B-13
MMAP, B-9
VMA/EMA programming, 4-12

VMA/EMA subroutines, 4-12, 4-15
   EIOSZ, 4-22
   EMAST, 4-16
   VMAIO, 4-20
   VMAST, 4-18
VMAIO subroutine, 4-20
VMAST subroutine, 4-18
VREAD subroutine, 4-33
VWRIT subroutine, 4-35

W

WHZAT - system status program, 1-33
Working Set, 4-2, 4-3, 4-5
   post, 4-31
write call - EXEC 2, 2-19
WRITF FMP call, 3-56
WRITT - disc cartridge save restore utility, 1-33

X

XLUEX subroutine, 2-8
XQPRG subroutine, 5-82
XREIO subroutine, 5-45

READER COMMENT SHEET

HP 1000 Computers

RTE-6/VM Programmer's Reference Manual

92084-90005 January, 1986

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please explain your answers under Comments, below, and use additional pages if necessary.

Is this manual technically accurate?	☐ **Yes**	☐ **No**
Are the concepts and wording easy to understand?	☐ **Yes**	☐ **No**
Is the format of this manual convenient in size, arrangement, and readability?	☐ **Yes**	☐ **No**

Comments:

This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

FROM: **Date** _____

Name _____

Company _____

Address _____

_____

_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 0141 CUPERTINO, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
Data Systems Division
11000 Wolfe Road
Cupertino, CA 95014-9974

FOLD FOLD

Part No. 92084-90005
Printed in U.S.A. January, 1985
E0185

HEWLETT PACKARD