HEWLETT PACKARD

RTE-A Programmer's

Reference Manual

Software Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804

Manual Part No. 92077-90007
E1193

Printed in U.S.A. November 1993
Seventh Edition

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

NOTICE

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

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THE 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.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARs 252.227.7013.

Copyright © 1983, 1985 - 1987, 1989, 1990, 1992, 1993 by Hewlett-Packard Company

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 Manual Numbering File or the Computer User's Documentation Index. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

Second Edition	Jun 1983	
Update 1	Dec 1983	FmOpenScratch, Program Examples
Reprint	Dec 1983	Update 1 Incorporated
Third Edition	Jan 1985	
Update 1	Jan 1986	
Reprint	Jan 1986	Update 1 Incorporated
Update 2	Oct 1986	
Reprint	Oct 1986	Update 2 Incorporated
Fourth Edition	Aug 1987	Rev. 5000 (Software Update 5.0)
Fifth Edition	Jan 1989	Rev. 5010 (Software Update 5.1)
Update 1	Jul 1990	Rev. 5020 (Software Update 5.2)
Sixth Edition	Dec 1992	Rev. 6000 (Software Update 6.0)
Seventh Edition	Nov 1993	Rev. 6100 (Software Update 6.1)

Preface

This manual describes the RTE-A Operating System services available to user programs. It explains the subroutine calls in detail and shows their formats. It also describes how to use Virtual Code Plus (VC+), product number HP 92078A, from user programs. This manual is the primary reference source for programmers who will write and maintain software under an RTE-A Operating System. If the programmer does not have RTE training or experience, Hewlett-Packard's customer training courses are recommended.

Chapter 1	Introduces the RTE-A services available to user programs, VC+, and system subroutine calls.
Chapter 2	Describes resource management and logical unit locks.
Chapter 3	Explains the standard I/O requests.
Chapter 4	Explains Class I/O requests.
Chapter 5	Describes scheduling and control of programs from within user programs.
Chapter 6	Explains time scheduling of programs from within user programs, and how to read the system time.
Chapter 7	Describes how to pass parameters between user programs, and how to use the data type conversion routines.
Chapter 8	Describes how to call FMP routines from user programs.
Chapter 9	Explains how to use the Virtual Memory Area (VMA), Extended Memory Area (EMA), and VMA Mapping Management subroutines.
Chapter 10	Describes how to use the Code and Data Separation (CDS) features of VC+.
Chapter 11	Explains how to use the spooling features of VC+.
Chapter 12	Describes Privileged Operation.
Chapter 13	Explains RTE-A Signals.
Chapter 14	Describes programmatic access of CI environment variables using the EXEC 39 call.
Appendix A	Explains the EXEC call error messages.
Appendix B	Describes how to convert FMGR calls to FMP calls.
Appendix C	Describes the FMGR routines.
Appendix D	Shows the HP Character set.
Appendix E	Lists the RTE-A program types and how they are handled by the operating system.
Appendix F	Describes how the RTE-A file system cleans up open files.
NOTE:	File management information in this manual is based on the File Management Package (FMP). Refer to the *RTE-A User's Manual*, part number 92077-90002, for details of the Command Interpreter hierarchical file system.

Table of Contents

Chapter 1
Introduction

Chapter 2
Resource Management

Chapter 3
Standard I/O

Chapter 4
Class I/O

Chapter 5
Program Control

Chapter 6
Time Operation Requests

Chapter 7
Parameter Passing and Conversion

Chapter 8
FMP Routines

Chapter 9
VMA and EMA Programming

Chapter 10
CDS Programming

Chapter 11
Programmatic Spooling

Chapter 12
Privileged Operation

Chapter 13
RTE-A Signals

Chapter 14
Programmatic Environment Variable Access

Appendix A
Error Messages

Appendix B
Converting FMGR File Calls

Appendix C
FMGR Calls

Appendix D
HP Character Set

Appendix E
Program Types for RTE-A

Appendix F
Cleaning Up Open Files

List of Illustrations

Tables

1

Introduction

RTE-A is the Real-Time Executive operating system for HP A-Series computers. The RTE-A Operating System coordinates requests for system services and resources and allocates them to the requesting programs as necessary. Requests for system resources can be made interactively, using the operator commands described in the *RTE-A User's Manual*, part number 92077-90002, or from your programs, using the operating system subroutine calls described in this manual. Located at the end of this chapter, is a functional listing of the subroutines available with RTE-A.

Overview

The RTE-A Operating System services described in this manual are summarized below:

- Executive Communication (EXEC) calls, the communication link between your programs and most system services.
- Program Segmentation: separation of a large program into several sections of code to allow it to execute in a memory partition smaller than its total size. Program segmentation can be implemented transparently by the Code and Data Separation (CDS) feature in VC+ or from a program using an EXEC call.
- Resource Management: a system that lets cooperating programs share system resources, such as files and peripherals.
- Class I/O: a scheme to permit a program to continue execution while its I/O requests are being processed and to facilitate program-to-program communication.
- Program Scheduling and control from within your programs, with multiprogramming, so several programs can be active at once, and time-slicing, so computation-heavy programs do not monopolize the CPU.
- Partitioned Memory that uses the hardware Dynamic Mapping System (DMS) to control access to as much as 24 Mbytes of memory.
- Extended Memory Area (EMA): allows your programs to store and access very large data arrays. Data in the EMA can be shared among programs.
- Demand-Paged Virtual Memory that produces programs that can access data structures as large as 128 MBytes.

These RTE-A features and services are accessed by your programs using EXEC, File Management Package (FMP), and System Library calls.

Executive Communication

EXEC calls are the line of communication between executing programs and system services. EXEC calls are coded directly into your programs. They have a structured format with optional parameters that define the function of the call.

Through EXEC calls, your program can:

- Perform input and output operations
- Terminate or suspend itself
- Load one of its segments
- Schedule other programs
- Time-schedule other programs
- Recover its scheduling string
- Read the system time
- Obtain partition status information
- Control swapping
- Obtain information about memory

File Management Package

The File Management Package (FMP) lets your programs manipulate I/O devices and disk files. There are two ways to access FMP: using the interactive Command Interpreter (CI), as described in the *RTE-A User's Manual*, part number 92077-90002, or using FMP calls from your programs, as described in Chapter 8 of this manual.

The FMP library contains routines that can be called from your programs to create, access, or purge disk or non-disk files. Even at the simplest level, FMP calls let your programs:

- Create disk files
- Open and close files
- Read from and write to files
- Move a pointer within a file record
- Purge disk files
- Obtain pointer position and file status
- Rename files
- Read a disk cartridge list
- Change the working directory
- Read file time stamps
- Perform masked file searches

You can access FMGR files (files from other RTE operating systems) interactively, using the CI or FMGR commands described in the *RTE-A User's Manual*; FMGR files can also be accessed from your programs, using the FMP calls described in Chapter 8 of this manual or the FMGR calls described in Appendix C.

System Library

The system library, part of the RTE-A Operating System, is a collection of relocatable routines that act as the interface between your programs and some system services. Some important RTE-A services are accessed by system library routine calls. The following system library routines are described in this manual:

- Resource management (Chapter 2)
- Parameter passing (Chapter 7)
- VMA and EMA programming (Chapter 9)

The system library also contains general-purpose routines that perform valuable services, such as:

- Re-entrant I/O processing
- Data conversion and string manipulation
- System status query
- Session environment services

Located at the end of this chapter, is a functional listing of the library routines available with RTE-A. Many of the routines listed there are described in detail in the *RTE-A • RTE-6/VM Relocatable Libraries Reference Manual*, part number 92077-90037.

EXEC and System Library Call Formats

All EXEC and system library calls must conform to the formats described in this section. The calls are shown in the format used in FORTRAN and Pascal.

Call Statement Conventions

Square brackets ([ ]) indicate optional parameters, as in the notation

[,*pram1*[,*pram2*[,*pram3*[,*pram4*]]]]

in which there are four optional parameters. Parameters are always separated by a single comma. Optional parameters that are not used can be left out of the call, unless one of the following optional parameters is used, as in the following example:

Call format: `CALL EXEC(`*ecode*`[,`*pram1*`[,`*pram2*`[,`*pram3*`[,`*pram4*`]]]])`

Actual call: `CALL EXEC(CODE,0,0,VALUE)`

The parameters *pram1* and *pram2* are not needed in the actual call, so their places are held by zeroes. The place of *pram4* need not be held.

All EXEC calls require the first parameter, *ecode*, to identify the function of the call. In RTE-A manuals, EXEC calls are identified by the *ecode* parameter, as in EXEC 6, which means an EXEC call with *ecode* = 6. Bits 15 and 14 of *ecode* are used for the no-abort and no suspend options described in the EXEC Error Processing section. All other bits are significant and should be set to zero to prevent unexpected results.

A-, B-, X-, and Y-Register Usage

The A-, B-, X-, and Y-Registers are not normally used by the high-level source code of programs. These registers are normally used by the macrocode generated by compilers. While the programmer can determine the current value of the A- and B-Registers with the ABREG subroutine, these registers should not be used to hold user data. Most system calls, such as EXEC, modify the A-, B-, X-, and Y-Registers.

A- and B-Register Return Values

EXEC calls return information to the A- and B-Registers. Information requested by EXEC calls is usually returned to the B-Register. In FORTRAN and Pascal, use the ABREG subroutine to read the A- and B-Registers into two variables in a program. The calling sequence is:

```
CALL ABREG(areg,breg)
```

areg and *breg* are one-word integer variables that receive the contents of the A- and B-Registers.

Note To receive correct data, the A- and B-Registers must not be changed between completion of the EXEC call and the ABREG call. The status is guaranteed to be correct if the ABREG call immediately follows the EXEC call. The variables being passed must be simple variables (no array or structure elements).

The A-, B-, X-, Y-, E-, and O-Registers may be altered by EXEC and system library calls and the values in these registers are undefined after execution of the call (unless specifically stated). If the values in these registers are to be preserved, then the registers should be saved by the programmer before using a system call.

EXEC Error Processing

Errors can occur in EXEC calls because illegal parameter values are passed, non-existent programs are scheduled, devices are not ready, security violations are detected, or parameters are left out.

Depending on the severity of the error, and upon the type of error processing selected by the calling program, the operating system may retry the failed request, suspend the calling program until the request can be completed, alert the program of the error, or abort the program.

The calling program is suspended when an I/O operation cannot be completed due to a temporary condition, such as a locked LU. The program is not aborted, and no error message is generated, but execution is suspended until the request can be completed.

The highest bits (14 and 15) of *ecode* select the no-suspend and no-abort error processing options. If no-suspend (bit 14) is set, the program is not suspended by a down device or a locked

LU; however, a program can still be I/O or buffer limit suspended. A code indicating the nature of the temporary error is stored in the A- and B-Registers, and the program continues. If no-abort (bit 15) is set, the program is not aborted by serious errors. Again, a code indicating the nature of the error is stored in the A- and B-Registers, and control is passed back to the calling program.

The contents of the registers can then be read with the ABREG subroutine. The code is four ASCII characters, stored in the two registers.

The following are examples of EXEC calls that request program error processing:

```
IMPLICIT INTEGER (A-Z)
  :
ECODE = 17 + 40000B     ! 40000B: Bit 14 = 1, no-suspend
CALL EXEC (ECODE, ...) ! 17 requests EXEC 17
  :
ECODE = 9 + 100000B     ! 100000B: Bit 15 = 1, no-abort
CALL EXEC (ECODE, ...) ! 9 requests EXEC 9
  :
  :
ECODE = 2 + 140000B     ! 140000B: no-abort and no-suspend
CALL EXEC (ECODE, ...) ! 2 requests EXEC 2
  :
```

When no-suspend or no-abort is selected, EXEC uses a special error return to indicate that an error occurred, and to permit error processing within the calling program. When an error occurs on an EXEC call with the no-abort or no-suspend bit set, the EXEC call returns to the statement immediately after the EXEC call. This instruction should be a branch instruction to an error handling routine that will service the error.

If no error occurs, EXEC returns to one location past the error return, usually with information returned by the EXEC call stored in the A- and B-Registers. This type of return occurs only when no-abort or no-suspend is specified, as shown in the following example:

```
        jsb exec
        def *+5
        def =b140001        ! Set no-abort and no-suspend bits
        def =d1
        def buffer
        def buflen
errrtn  jmp execerror       ! If error, jump to execerror
okrtn   ...                 ! If no error, continue
```

EXEC error handling is easy to use from FORTRAN, and works in the same way as other alternate returns do. The following sample program demonstrates the use of the no-abort error alternate return:

```
      IMPLICIT INTEGER (A-Z)
            :
      CALL EXEC (100001B,1,buffer,len,*777)
C   return here for no error
          :
C   exec error handling section
  777 CALL ABREG (A, B)
      IF (A .eq. 2hIO .and. B .eq. 2h04) then
         WRITE(1,*) 'IO04 error in EXEC call!'
```

FORTRAN recognizes all EXEC calls, and returns to the alternate return label number (the last parameter of the EXEC call) when an error occurs. Other routines can use this alternate return convention for errors, but FORTRAN does not recognize the alternate return label unless the $ALIAS compiler directive is used. Refer to the *FORTRAN 77 Reference Manual*, part number 92836-90001, for more information. Pascal programmers can also use the $ALIAS compiler directive to make no-abort calls; refer to the Pascal manual for more information.

The following EXEC errors always abort the calling program, regardless of the error processing options selected. The most common EXEC error is RQ. Appendix A describes the EXEC error messages.

Error Code	Error Type
LD	Segment load failed
MP	Memory Protect
PE	CPU Memory Parity Error
RQ	Request Code
SR	EXEC call executed in a privileged subroutine

If Class I/O is being used, the no-wait bit (bit 15) of the CLASS parameter can be used with the no-suspend bit of *ecode* to control error processing. Refer to the Class I/O Operation section, Chapter 4, for more information.

Functional Grouping of Library Routines

RTE-A is delivered with a collection of relocatable subroutines. This group of subroutines interfaces user programs with system services. The following pages contain a listing of those subroutines. The detailed description of each subroutine can be found in this manual or in the *RTE-A • RTE-6/VM Relocatable Libraries Reference Manual*, part number 92077-90037. The functional listing given here indicates the page number on which the subroutine is documented. The subroutines documented in the *RTE-A • RTE-6/VM Relocatable Libraries Reference Manual* are indicated in the listing by the mnemonic "*rel*", for example "*rel*-7-16" refers you to page 7-16 of the *RTE-A • RTE-6/VM Relocatable Libraries Reference Manual.*

The subroutines listed in this chapter are organized into the following functional groups:

- ASCII/Integer Conversion
- Bit Map Manipulation
- Buffer and String Manipulation
  - Integer Buffer Routines
  - Character String Routines
  - HpCrt and HpZ Buffer Routines
- Character Buffer Manipulation
- Command Stack
- Error Handling
- Interprocess Communication
  - Class I/O
  - Parameter Passing
  - Programmatic Environment Variable Access
  - Signals
- I/O
- Machine-level Access
- Math
  - Absolute Value Subroutines
  - Complex Number Arithmetic Subroutines
  - Double Integer Utilities
  - Exponents, Logs, and Roots
  - HP 1000/IEEE Floating Point Conversion Subroutines
  - Number Conversion Subroutines
  - Real Number Arithmetic Subroutines
  - Trigonometry Subroutines
  - VIS Subroutines
  - Miscellaneous Subroutines
- Multiuser
- Parsing Routines
- Privileged Operation
- Program Control
- Resource Management
- System Status
- Time Operations

ASCII/INTEGER CONVERSION SUBROUTINES

.FMUI	ASCII digit to internal numeric conversion	*rel*-5-32
.FMUO	Numeric to ASCII conversion	*rel*-5-32
.FMUP	Internal to normal format conversion	*rel*-5-32
.FMUR	Rounding of digit string produced by .FMUO	*rel*-5-34
CNUMD	Convert unsigned 16-bit integer to ASCII decimal	7-11
CNUMO	Convert unsigned 16-bit integer to ASCII octal	7-11
DecimalToDint	ASCII to double integer	*rel*-7-21
DecimalToInt	ASCII to single integer	*rel*-7-22
DintToDecimal	Double integer to ASCII	*rel*-7-22
DintToDecimalr	Double integer to ASCII	*rel*-7-23
DintToOctal	Double integer to octal	*rel*-7-23
DintToOctalr	Double integer to octal	*rel*-7-24
HexToInt	ASCII hexadecimal to single integer	*rel*-7-29
HpZBinc	Convert a number to binary	*rel*-12-42
HpZBino	Convert value to its binary ASCII representation	*rel*-12-42
HpZDecc	Convert a number to ASCII numerals	*rel*-12-43
HpZDeco	Convert an integer*2 number to ASCII decimal representation	*rel*-12-42
HpZDecv	Convert an integer*2 number to ASCII decimal representation	*rel*-12-42
HpZDicv	Convert double integer value to ASCII decimal representation	*rel*-12-44
HpZDParse	Parse the next occurring token in the input buffer	*rel*-12-45
HpZGetNumB2	Convert number in input buffer to integer*2 decimal or octal	*rel*-12-53
HpZGetNumB4	Convert number in input buffer to integer*4 decimal or octal	*rel*-12-53
HpZGetNumD2	Convert number in input buffer to integer*2 decimal	*rel*-12-53
HpZGetNumD4	Convert number in input buffer to integer*4 decimal	*rel*-12-53
HpZGetNumO2	Convert number in input buffer to integer*2 octal	*rel*-12-53
HpZGetNumO4	Convert number in input buffer to integer*4 octal	*rel*-12-53
HpZGetNumX	Convert digits to internal representation	*rel*-12-55
HpZHexc	Convert a number to hexadecimal	*rel*-12-56
HpZHexi	Parse hexadecimal ASCII integers	*rel*-12-57
HpZHexo	Convert an integer*2 number to hexadecimal	*rel*-12-58
HpZOctc	Convert a value to its octal ASCII representation	*rel*-12-68
HpZOctd	Convert a double integer value to its octal ASCII representation	*rel*-12-68
HpZOcto	Convert the passed value to its octal ASCII representation	*rel*-12-69
HpZOctv	Convert the passed value to its octal ASCII representation	*rel*-12-69
HpZParse	Parse routine for 16-character parameters	*rel*-12-71
HpZRomanNumeral	Convert a value to its Roman numeral equivalent	*rel*-12-77
HpZUdeco	Convert integer*2 number to unsigned decimal representation	*rel*-12-78
HpZUdecv	Convert integer*2 number to unsigned decimal representation, suppressing leading zeros	*rel*-12-79
INPRS	Inverse parse; convert buffer to original ASCII form	7-8
IntString	Integer to ASCII	*rel*-7-31
IntToDecimal	Integer to ASCII	*rel*-7-32
IntToDecimalr	Integer to ASCII	*rel*-7-32
IntToHex	Integer to ASCII hexadecimal	*rel*-7-33
IntToHexR	Integer to ASCII hexadecimal with right justification	*rel*-7-33
IntToOctal	Integer to octal	*rel*-7-34
IntToOctalr	Integer to octal	*rel*-7-34
KCVT	Convert positive integer to base 10; return last 2 ASCII digits	7-11
OctalToDint	ASCII digit to internal numeric conversion	*rel*-7-37
OctalToInt	ASCII to integer	*rel*-7-38
PARSE	Parse ASCII input buffer	7-7

BIT MAP MANIPULATION

BUFFER AND STRING MANIPULATION

Character String Routines

HpCrt and HpZ Buffer Routines

Integer Buffer Routines

COMMAND STACK

ERROR HANDLING

I/O

INTERPROCESS COMMUNICATION

Class I/O

See "Class I/O" chapter

Parameter Passing

Programmatic Environment Variable Access

Signals

MACHINE-LEVEL ACCESS

..MAP	Compute address of specified element of a 2- or 3-dimensional array	*rel*-5-41
.ENTC and .ENTN	Transfer true address of parameters from calling sequence into a subroutine; adjust return address to true return point	*rel*-5-28
.ENTP and .ENTR	Transfer true address of parameters from calling sequence into a subroutine; adjust return address to true return point	*rel*-5-29
.GOTO	Transfer control to the location indicated by a FORTRAN computed GOTO statement	*rel*-5-35
.MAP	Return actual address of a particular element of a two-dimensional FORTRAN array	*rel*-5-36
.MPY	Replace the subroutine call with the hardware instructions to multiply by integer i and j	*rel*-3-102
.PCAD	Return true address of parameter passed to a subroutine	*rel*-5-39
$SETP	Set up a list of pointers	*rel*-5-42
%SSW	Set sign bit of A-Register according to bit n of switch register	*rel*-5-43
ABREG	Obtain contents of A- and B-Registers	*rel*-5-2, 12-1
AddressOf	Return direct address	*rel*-7-1
IGET and IXGET	Read contents of a memory address	*rel*-5-10
ISSR	Set S-Register to value n	*rel*-5-15
ISSW	Set sign bit of A-Register according to bit n of switch register	*rel*-5-16
OVF	Set sign bit of A-Register according to overflow bit	*rel*-5-21
ReadA990Clock	Read the calendar clock of the A990	*rel*-7-40
WriteA990Clock	Set the calendar clock on the A990	*rel*-7-54

MATH

Absolute Value Subroutines

.ABS	Absolute value (double real)	*rel*-3-62
%ABS	(Call-by-name) IABS	*rel*-3-146
%BS	(Call-by-name) ABS	*rel*-3-150
ABS	Absolute value (real)	*rel*-3-2
CABS	Absolute value (complex)	*rel*-3-12
DABS	Absolute value (extended real)	*rel*-3-20
DIM	Positive difference (real)	*rel*-3-27
IABS	Absolute value (integer)	*rel*-3-43
IDIM	Positive difference (integer)	*rel*-3-45

Complex Number Arithmetic Subroutines

..CCM	Complement (complex)	*rel*-3-132
.CADD	Complex add	*rel*-3-66
.CDBL	Extract the real part of a complex number in extended real form	*rel*-3-67
.CDIV	Complex divide	*rel*-3-68
.CMPY	Complex multiply	*rel*-3-72
.CSUB	Complex subtract	*rel*-3-76
AIMAG	Extract imaginary part of complex (real)	*rel*-3-3
CMPLX	Combine real and imaginary complex	*rel*-3-15
CONJG	Form conjugate of complex	*rel*-3-16
REAL	Extract the real part of a complex	*rel*-3-53

Decimal String Arithmetic Subroutines

JSCOM	Compare two substrings	*rel*-10-7
SA2DE	Convert substring in A2 format to decimal	*rel*-10-31
SADD	Perform a decimal add of two substrings	*rel*-10-17
SCARY	Examine D2 decimal substring for carries	*rel*-10-33
SD1D2	Convert substring in D1 format to D2	*rel*-10-37
SD2D1	Convert substring in D2 format to D1	*rel*-10-38
SDCAR	Examine D1 decimal substring for carries	*rel*-10-34

SDEA2	Convert substring in decimal format to A2	*rel*-10-36
SDIV	Perform a decimal division of two substrings	*rel*-10-19
SEDIT	Edit data in substring array	*rel*-10-28
SFILL	Fill area in a substring array with a specified character	*rel*-10-9
SGET	Get a specified character from a substring	*rel*-10-10
SMOVE	Move data from one string to another	*rel*-10-11
SMPY	Perform a decimal multiply of two substrings	*rel*-10-22
SPUT	Put a specified character in a substring	*rel*-10-13
SSIGN	Find the sign of a number	*rel*-10-40
SSUB	Perform a decimal subtraction of two substrings	*rel*-10-26
SZONE	Find the zone punch of a character	*rel*-10-14

Double Integer Utilities

.DADS	Double integer add and subtract	*rel*-4-3
.DCO	Compare two double integers	*rel*-4-4
.DDE	Decrement double integer	*rel*-4-5
.DDI	Double integer divide	*rel*-4-6
.DDIR	Double integer divide	*rel*-4-6
.DDS	Double integer decrement and skip if zero	*rel*-4-7
.DIN	Increment double integer	*rel*-4-8
.DIS	Double integer increment and skip if zero	*rel*-4-9
.DMP	Double integer multiply	*rel*-4-10
.DNG	Negate double integer	*rel*-4-11
.FIXD	Convert real to double integer	*rel*-4-12
.FLTD	Convert double integer to real	*rel*-4-13
.TFTD	Convert double integer to double real	*rel*-4-14
.TFXD	Convert double real to double integer	*rel*-4-15
.XFTD	Convert double integer to extended real	*rel*-4-16
.XFXD	Convert extended real to double integer	*rel*-4-17
FLTDR	Convert double-length record number to real	*rel*-4-2

Computer Museum

Exponents, Logs, and Roots

.CTOI	Raise complex to integer power	*rel*-3-78
.DTOD	Raise extended real to extended real power	*rel*-3-83
.DTOI	Raise extended real to integer power	*rel*-3-84
.DTOR	Raise extended real to real power; extended real result	*rel*-3-85
.EXP	Raise e to double real power	*rel*-3-86
.FPWR	Raise real to integer power	*rel*-3-91
.ITOI	Raise integer to integer power	*rel*-3-96
.LOG	Natural log (double real)	*rel*-3-97
.LOG0	Base 10 log (double real)	*rel*-3-98
.PWR2	Multiply a real by 2 raised to integer power	*rel*-3-105
.RTOD	Raise real to extended real power; extended real result	*rel*-3-106
.RTOI	Raise real to integer power	*rel*-3-107
.RTOR	Raise real to real power	*rel*-3-108
.RTOT	Raise real to double real power	*rel*-3-109
.SQRT	Square root (double real)	*rel*-3-112
.TPWR	Raise double real to unsigned power	*rel*-3-120
.TTOI	Raise double real to integer power	*rel*-3-121
.TTOR	Raise double real to real power	*rel*-3-122
.TTOT	Raise double real to double real power	*rel*-3-123
$EXP	Raise e to extended real power; no error return	*rel*-3-141
$LOG	Natural log (extended real); no error return	*rel*-3-142
$LOGT	Base 10 log (extended real); no error return	*rel*-3-143
$SQRT	Square root (extended real); no error return	*rel*-3-144
%LOG	Natural log (real; call-by-name)	*rel*-3-156
%LOGT	Base 10 log (real; call-by-name)	*rel*-3-157
%QRT	Square root (real; call-by-name)	*rel*-3-162
%XP	Raise e to real power; call-by-name	*rel*-3-165

/EXP	.EXP with no error return	*rel*-3-169
/EXTH	2**n*2**z (small double real z)	*rel*-3-170
/LOG	.LOG with no error return	*rel*-3-171
/LOG0	.LOG0 with no error return	*rel*-3-172
/SQRT	.SQRT with no error return	*rel*-3-174
ALOG	Natural log (real)	*rel*-3-5
ALOGT	Base 10 log (real)	*rel*-3-6
CEXP	Raise e to complex power	*rel*-3-13
CLOG	Natural log (complex)	*rel*-3-14
CSQRT	Complex square root (complex)	*rel*-3-19
DEXP	Extended real e (extended real)	*rel*-3-26
DLOG	Natural log (extended real)	*rel*-3-28
DLOGT	Base 10 log (extended real)	*rel*-3-29
DSQRT	Square root (extended real)	*rel*-3-36
EXP	Raise e to real power	*rel*-3-41
SQRT	Square root (real)	*rel*-3-59

HP 1000/IEEE Floating Point Conversion Subroutines

DFCHI	HP 1000 double precision to IEEE	*rel*-11-1
DFCIH	IEEE double precision to HP 1000	*rel*-11-2
FCHI	HP 1000 single precision to IEEE	*rel*-11-2
FCIH	IEEE single precision to HP 1000	*rel*-11-3

Number Conversion Subroutines

.BLE	Convert real to double real	*rel*-3-65
.CINT	Convert complex to integer	*rel*-3-71
.CMRS	Reduce argument for SIN, COS, TAN, EXP	*rel*-3-73
.CTBL	Convert complex real to double real	*rel*-3-77
.DCPX	Convert extended real to complex	*rel*-3-79
.DINT	Convert extended real to integer	*rel*-3-81
.DTBL	Convert extended real to double real	*rel*-3-82
.ICPX	Convert integer to complex	*rel*-3-92
.IDBL	Convert integer to extended real	*rel*-3-93
.IENT	Greatest integer no greater than given real x	*rel*-3-94
.ITBL	Convert integer to double real	*rel*-3-95
.NGL	Convert double real to real	*rel*-3-103
.PACK	Convert signed mantissa of real into normalized real format	*rel*-3-104
.TCPX	Convert double real to complex real	*rel*-3-116
.TDBL	Convert double real to extended real without rounding	*rel*-3-117
.TINT	Convert double real to integer	*rel*-3-119
%FIX	Convert real to integer; call-by-name	*rel*-3-151
%INT	Truncate real; call-by-name	*rel*-3-154
%LOAT	Convert integer to real; call-by-name	*rel*-3-155
%NT	Truncate real to integer; call-by-name	*rel*-3-158
/CMRT	Range reduction for .SIN, .COS, .TAN, .EXP, .TANH	*rel*-3-168
/TINT	Convert double precision to integer	*rel*-3-176
AINT	Truncate (real)	*rel*-3-4
AMOD	x modulo y (real x and y)	*rel*-3-9
DBLE	Convert real to extended real	*rel*-3-23
DDINT	Truncate (extended real)	*rel*-3-25
DMOD	x modulo y (extended real x and y)	*rel*-3-31
ENTIE	Greatest integer not greater than given real	*rel*-3-39
FLOAT	Convert integer to real	*rel*-3-42
IDINT	Truncate extended real to integer	*rel*-3-46
IFIX	Convert real to integer	*rel*-3-47
INT	Truncate real to integer j	*rel*-3-48
MOD	i modulo j (integer i and j)	*rel*-3-52
SNGL	Convert extended real to real	*rel*-3-56
SNGM	Convert extended real to real without rounding	*rel*-3-57
SPOLY	Evaluate the quotient of two polynomials in single precision	*rel*-3-58

Real Number Arithmetic Subroutines

..DCM	Complement (extended real)	*rel*-3-133
..DLC	Load and complement (real)	*rel*-3-134
..FCM	Complement (real)	*rel*-3-135
..TCM	Negate (double real)	*rel*-3-136
.DTBL	Convert extended real to double real	*rel*-3-82
.FDV	Real divide	*rel*-3-88
.FLUN	Unpack (real); place exponent in A-Register, lower mantissa in B-Register	*rel*-3-89
.FMP	Real multiply	*rel*-3-90
.MANT	Extract mantissa of real x	*rel*-3-99
.MAX1 and .MIN1	Find the maximum (or minimum) of a list of double reals	*rel*-3-100
.MOD	Double real remainder of real divide	*rel*-3-101
.SIGN	Transfer sign of one double real to another	*rel*-3-110
.TADD	Double real add	*rel*-3-113
.TDBL	Convert double real to extended real with rounding	*rel*-3-117
.TDIV	Double real divide	*rel*-3-113
.TINT	Convert double real to integer	*rel*-3-119
.TMPY	Double real multiply	*rel*-3-113
.TSUB	Double real subtract	*rel*-3-113
.XCOM	Complement extended real unpacked mantissa in place	*rel*-3-125
.XDIV	Extended real divide	*rel*-3-126
.XMPY	Extended real multiply	*rel*-3-128
.XPAK	Normalize, round, and pack with the exponent an extended real mantissa	*rel*-3-129
.YINT	Truncate fractional part of double real	*rel*-3-131
%IGN	Transfer sign of real or integer to real	*rel*-3-152
/ATLG	Compute (1−x)/(1+x) (double precision)	*rel*-3-166
DSIGN	Transfer sign of one extended real to another	*rel*-3-34
ENTIX	Greatest integer no greater than given extended real; result is extended real	*rel*-3-40
SIGN	Transfer sign of real or integer to real	*rel*-3-54

Trigonometry Subroutines

.ATAN	Arctangent (double real)	*rel*-3-63
.ATN2	Arctangent double real quotient	*rel*-3-64
.COS	Cosine (double real)	*rel*-3-74
.SIN	Sine (double real)	*rel*-3-111
.TAN	Tangent (double real)	*rel*-3-114
.TANH	Hyperbolic tangent (double real)	*rel*-3-115
$TAN	DTAN with no error return	*rel*-3-145
%AN	Tangent (real); call-by-name	*rel*-3-147
%ANH	Hyperbolic tangent (real); call-by-name	*rel*-3-149
%IN	Sine (real); call-by-name	*rel*-3-153
%OS	Cosine (real); call-by-name	*rel*-3-160
%TAN	Arctangent (real); call-by-name	*rel*-3-164
/COS	.COS with no error return	*rel*-3-167
/SIN	.SIN with no error return	*rel*-3-173
/TAN	.TAN with no error return	*rel*-3-175
ATAN	Arctangent (real)	*rel*-3-10
ATAN2	Arctangent (real)	*rel*-3-11
COS	Cosine (real)	*rel*-3-17
CSNCS	Complex sine or cosine (complex)	*rel*-3-18
DATAN	Arctangent (extended real)	*rel*-3-21
DATN2	Arctangent (extended real x, double real y)	*rel*-3-22
DCOS	Cosine (extended real)	*rel*-3-24
DSIN	Sine (extended real)	*rel*-3-35
DTAN	Tangent (extended real)	*rel*-3-37
DTANH	Hyperbolic tangent (real)	*rel*-3-38
SIN	Sine (real)	*rel*-3-55
TAN	Tangent (real)	*rel*-3-60
TANH	Hyperbolic tangent (real)	*rel*-3-61

VIS Subroutines

DVABS	Absolutevalue routine (double precision)	*rel*-8-13
DVADD	Vector add (double precision)	*rel*-8-9
DVDIV	Vector divide (double precision)	*rel*-8-9
DVDOT	Vector dot product routine (double real)	*rel*-8-17
DVMAB	Vector largest value (absolute) (double real)	*rel*-8-20
DVMAX	Vector largest value (double real)	*rel*-8-20
DVMIB	Vector smallest value (absolute) (double real)	*rel*-8-20
DVMIN	Vector smallest value (double real)	*rel*-8-20
DVMOV	Vector move routine (double real)	*rel*-8-24
DVMPY	Vector multiply (double real)	*rel*-8-9
DVNRM	Vector sum (absolute) routine (double real)	*rel*-8-14
DVPIV	Vector pivot routine (double real)	*rel*-8-18
DVSAD	Vector-scalar add (double real)	*rel*-8-11
DVSDV	Vector-scalar divide (double real)	*rel*-8-11
DVSMY	Vector-scalar multiply (double real)	*rel*-8-11
DVSSB	Vector-scalar subtract (double real)	*rel*-8-11
DVSUB	Vector subtract (double real)	*rel*-8-9
DVSUM	Vector sum routine (double real)	*rel*-8-14
DVSWP	Vector copy routine (double real)	*rel*-8-24
DVWMV	Vector non-EMA to EMA move routine (EMA double real)	*rel*-8-26
DWABS	Absolute value routine (EMA double real)	*rel*-8-13
DWADD	Vector add (EMA double real)	*rel*-8-9
DWDIV	Vector divide (EMA double real)	*rel*-8-9
DWDOT	Vector dot product routine (EMA double real)	*rel*-8-17
DWMAB	Vector largest value (absolute) (EMA double real)	*rel*-8-20
DWMAX	Vector largest value (EMA double real)	*rel*-8-20
DWMIB	Vector smallest value (absolute) (EMA double real)	*rel*-8-20
DWMIN	Vector smallest value (EMA double real)	*rel*-8-20
DWMOV	Vector move routine (EMA double real)	*rel*-8-24
DWMPY	Vector multiply (EMA double real)	*rel*-8-9
DWNRM	Vector sum (absolute) routine (EMA double real)	*rel*-8-14
DWPIV	Vector pivot routine (EMA double real)	*rel*-8-18
DWSAD	Vector-scalar add (EMA double real)	*rel*-8-11
DWSDV	Vector-scalar divide (EMA double real)	*rel*-8-11
DWSMY	Vector-scalar multiply (EMA double real)	*rel*-8-11
DWSSB	Vector-scalar subtract (EMA double real)	*rel*-8-11
DWSUB	Vector subtract (EMA double real)	*rel*-8-9
DWSUM	Vector sum routine (EMA double real)	*rel*-8-14
DWSWP	Vector copy routine (EMA double real)	*rel*-8-24
DWVMV	Vector EMA to non-EMA move routine (double real)	*rel*-8-26
VABS	Absolute value routine (single precision)	*rel*-8-13
VADD	Vector add (single precision)	*rel*-8-9
VDIV	Vector divide (single precision)	*rel*-8-9
VDOT	Vector dot product routine (single precision)	*rel*-8-17
VMAB	Vector largest value (absolute) (single precision)	*rel*-8-20
VMAX	Vector largest value (single precision)	*rel*-8-20
VMIB	Vector smallest value (absolute) (single precision)	*rel*-8-20
VMIN	Vector smallest value (single precision)	*rel*-8-20
VMOV	Vector move routine (single precision)	*rel*-8-24
VMPY	Vector multiply (single precision)	*rel*-8-9
VNRM	Vector sum (absolute) routine (single precision)	*rel*-8-14
VPIV	Vector pivot routine (single precision)	*rel*-8-18
VSAD	Vector-scalar add (single precision)	*rel*-8-11
VSDV	Vector-scalar divide (single precision)	*rel*-8-11
VSMY	Vector-scalar multiply (single precision)	*rel*-8-11
VSSB	Vector-scalar subtract (single precision)	*rel*-8-11
VSUB	Vector subtract (single precision)	*rel*-8-9
VSUM	Vector sum routine (single precision)	*rel*-8-14

VSWP	Vector copy routine (single precision)	*rel*-8-24
VWMOV	Vector non-EMA to EMA move routine (single precision)	*rel*-8-26
WABS	Absolute value routine (EMA single precision)	*rel*-8-13
WADD	Vector add (EMA single precision)	*rel*-8-9
WDIV	Vector divide (EMA single precision)	*rel*-8-9
WDOT	Vector dot product routine (EMA single precision)	*rel*-8-17
WMAB	Vector largest value (absolute) (EMA single precision)	*rel*-8-20
WMAX	Vector largest value (EMA single precision)	*rel*-8-20
WMIB	Vector smallest value (absolute) (EMA single precision)	*rel*-8-20
WMIN	Vector smallest value (EMA single precision)	*rel*-8-20
WMOV	Vector move routine (EMA single precision)	*rel*-8-24
WMPY	Vector multiply (EMA single precision)	*rel*-8-9
WNRM	Vector sum (absolute) routine (EMA single precision)	*rel*-8-14
WPIV	Vector pivot routine (EMA single precision)	*rel*-8-18
WSAD	Vector-scalar add (EMA single precision)	*rel*-8-11
WSDV	Vector-scalar divide (EMA single precision)	*rel*-8-11
WSMY	Vector-scalar multiply (EMA single precision)	*rel*-8-11
WSSB	Vector-scalar subtract (EMA single precision)	*rel*-8-11
WSUB	Vector subtract (EMA single precision)	*rel*-8-9
WSUM	Vector sum routine (EMA single precision)	*rel*-8-14
WSWP	Vector copy routine (EMA single precision)	*rel*-8-24
WVMOV	Vector EMA to non-EMA copy routine (single precision)	*rel*-8-26

Miscellaneous Math Subroutines

..TCM	Negate (double real)	*rel*-3-136
.CFER	Move four words from address x to address y (complex transfer)	*rel*-3-69
.CHEB	Evaluate Chebyshev series	*rel*-3-70
.FLUN	Unpack (real); place exponent in A-Register, lower mantissa in B-Register	*rel*-3-89
.MANT	Extract mantissa of real x	*rel*-3-99
.TENT	Find the greatest integer i less than or equal to a double real	*rel*-3-118
.XFER	Move three words from address x to address y (extended real transfer)	*rel*-3-127
%AND	Logical product (two integers); call-by-name	*rel*-3-148
%OR	Logical inclusive OR (two integers); call-by-name	*rel*-3-159
%OT	Complement (integer); call-by-name	*rel*-3-161
%SIGN	Transfer sign of real or integer z to integer i; call-by-name	*rel*-3-163
DPOLY	Evaluate quotient of two polynomials (double precision)	*rel*-3-32
IAND	Logical product (two integers)	*rel*-3-44
IOR	Logical inclusive OR (two integers)	*rel*-3-49
ISIGN	Transfer sign of real or integer z to integer i	*rel*-3-50
IXOR	Exclusive OR (integer)	*rel*-3-51
XPOLY and .XPLY	Evaluate extended real polynomial	*rel*-3-130

MULTIUSER

AccessLU	Check for LU access	*rel*-6-2
ATACH	Attach calling program to a session	*rel*-6-3
ATCRT	Attach to CRT	*rel*-6-4
CLGOF	Call LOGOF	*rel*-6-5
CLGON	Call LOGON	*rel*-6-6
DTACH	Detach from session	*rel*-6-7
FromSySession	Check system session	*rel*-6-8
GetAcctInfo	Access user and group accounting	*rel*-6-8
GetOwnerNum	Return owner ID	*rel*-6-10
GetResetInfo	Access and clear multiuser account	*rel*-6-10
GETSN	Get session number	*rel*-6-11
GPNAM	Return group name	*rel*-6-11
GroupToId	Return group ID number given group name	*rel*-6-12
IdToGroup	Return group name given group ID number	*rel*-6-12
IdToOwner	Return user name	*rel*-6-13

LUSES	Return user table address	*rel*-6-13
Member	Determine if user is in group	*rel*-6-13
OwnerToId	Return user ID	*rel*-6-14
ProgIsSuper	Determine if program is a super program	*rel*-6-14
ResetAcctTotals	Reset user and group accounting totals	*rel*-6-15
RTNSN	Return session number	*rel*-6-16
SessnToOwnerName	Return user name	*rel*-6-16
SetAcctLimits	Set user and group accounting limits	*rel*-6-17
SuperUser	Check for or if superuser	*rel*-6-18
SYCON	Write a message to the system console	*rel*-6-18
SystemProcess	Check for or if system process	*rel*-6-19
UserIsSuper	Check for or if superuser	*rel*-6-19
USNAM	Return user name	*rel*-6-19
USNUM	Return session number	*rel*-6-20
VFNAM	Verify user name	*rel*-6-20

PARSING

HpZDParse	Parse the next occurring token in the input buffer	*rel*-12-45
HpZParse	Parse routine for 16-character parameters	*rel*-12-71
INAMR	Inverse parse of 10-word parameter buffer generated by NAMR	*rel*-5-11
INPRS	Inverse parse; convert buffer to original ASCII form	7-8
NAMR	Read input buffer, produce 10-word parameter buffer	*rel*-5-18
PARSE	Parse ASCII input buffer	7-7
SplitCommand	Parse a string	*rel*-7-47
SplitString	Parse a string	*rel*-7-48

PRIVILEGED OPERATION

$LIBR	Go privileged (highest level)	12-3
$LIBX	Resume normal operation after calling $LIBR	12-3
DispatchLock	Prevent all other user programs from executing	12-2
DispatchUnlock	Remove lock set by DispatchLock	12-2
GOPRV	Go privileged; disable normal memory protect mechanism	12-1
UNPRV	Resume normal operation after calling GOPRV	12-1

PROGRAM CONTROL

CHNGPR	Change program priority	5-4
EXEC 6	Stop program execution	5-5
EXEC 7	Suspend program execution	5-8
EXEC 8	Load program overlay	5-2
EXEC 9	Immediate program scheduling with wait	5-8
EXEC 10	Immediate program scheduling without wait	5-8
EXEC 22	Lock program into memory so it cannot be swapped	5-13
EXEC 23	Queue program scheduling with wait	5-8
EXEC 24	Queue program scheduling without wait	5-8
EXEC 26	Return memory limits of the partition of calling program	5-14
EXEC 29	Retrieve ID segment of specified program	5-16
GetFatherIdNum	Return father ID segment number	*rel*-7-28
HpLowerCaseName	Change the name of the program that calls it to lowercase	*rel*-12-33
HpZMesss	Send a command to the operator interface section of the OS	*rel*-12-61
IdAddToName	Convert ID segment address to program name and LU number	*rel*-7-30
IdAddToNumber	Convert ID segment address to segment number	*rel*-7-30
IDCLR	Deallocate ID segment	*rel*-7-30
IDGET	Retrieve ID segment of specified program	7-14
IDINFO	Return ID segment information	7-15
IdNumberToAdd	Convert ID segment number to segment address	*rel*-7-31
MESSS	Process base set commands	7-12

MyIdAdd	Return segment address	*rel*-7-36
PNAME	Return program name	7-14; *rel*-5-23
ProgramPriority	Return program priority	*rel*-7-38
SEGLD	Load program overlay; allows use of SEGRT and debug	5-3
SEGRT	Return to main from overlay	5-4

RESOURCE MANAGEMENT

LIMEM	Return starting location and size of memory area between end of program or stack area and end of program partition	2-13
LuLocked	Report is passed LU is locked	*rel*-7-36
LURQ	Give program exclusive access to an I/O device	2-8
RNRQ	Allocate and manage resource numbers	2-1
SetPriority	Set the priority of the currently executing program	*rel*-12-84
WhoLockedLu	Return ID segment address of program that locked LU	*rel*-7-53
WhoLockedRn	Return ID segment address of program that locked the specified resource number	*rel*-7-53

SYSTEM STATUS

.OPSY	Determine which operating system is in control	*rel*-5-37
CPUID	Get CPU identification	7-9
HpRte6	Determine if calling program is running on RTE-6/VM	*rel*-12-34
HpRteA	Determine if calling program is running on RTE-A	*rel*-12-34
HpZMesss	Send a command to the operator interface section of the OS	*rel*-12-61
IFBRK	Test break flag and clear if set	7-11
MESSS	Process base set commands	7-12
SamInfo	Return number of free words in SAM or XSAM	*rel*-7-46

TIME OPERATIONS

DayTime	Return ASCII time string	*rel*-7-21
ElapsedTime	Number of milliseconds since last time recorded by ResetTimer	*rel*-7-25
ETime	Number of centiseconds since specified base time	*rel*-7-25
EXEC 11	Retrieve current time	6-1
EXEC 12	Schedule program at specified time interval	6-2
FTIME	Return ASCII message giving time and date	6-7
GetRteTime	Read the system clock in three-word format	*rel*-7-29
HMSCtoRteTime	Convert Hr-Min-Sec-Centisec to RTE time format	*rel*-7-29
InvSeconds	Perform conversion that is the inverse of the Seconds routine	*rel*-7-35
KillTimer	Cancel current timer	13-27
LeapYear	Test a given year to see if it is a leap year	*rel*-7-35
NumericTime	Return numeric ASCII time string	*rel*-7-37
QueryTimer	Return number of ticks remaining before timer signal is to be generated	13-27
ResetTimer	Reset timer used by ElapsedTime routine	*rel*-7-40
RteDateToYrDoy	Convert from RTE combined year/day format to year and day	*rel*-7-44
RteTimeToHMSC	Convert centiseconds since midnight to Hr-Min-Sec-Centisec	*rel*-7-46
Seconds	Convert a time buffer to seconds since January 1, 1970	*rel*-7-47
SetTimer	Establish a new timer or reset an existing timer	13-27
SETTM	Set system time	6-7
TIMEF	Format time	*rel*-7-51
TIMEI and TIMEO	Measure difference between time in and time out	*rel*-5-27
TimeNow	Number of seconds since midnight January 1, 1970	*rel*-7-52
YrDoyToMonDom	Convert year and day of year to day of month, month, and day of week	*rel*-7-54
YrDoyToRteDate	Convert year and cardinal day to RTE format	*rel*-7-55

2

Resource Management

RTE-A has several mechanisms to control system resources. Some resource management features are:

- Exclusive use by a program of a subroutine, an area of memory, a disk file, or even the operating system itself.
- Exclusive use of a peripheral device, such as a printer or HP-IB device.
- Access to the memory between the end of a program and the end of its partition.
- Synchronization of resource access by a group of cooperating programs.

Two of the main mechanisms for resource management are resource numbers (RNs) and LU locks. RNs indicate when one of several cooperating programs is accessing a system resource that cannot be shared, so that the programs can coordinate their accesses. RNs cannot stop programs from failing to cooperate. That is the job of LU locks. When a program needs exclusive access to an LU or a file, the program locks the LU or opens the file in exclusive mode. Until the locking program either terminates, or unlocks the LU or closes the file, the LU or file is unavailable to other programs.

Resource Sharing with RNRQ

The RNRQ call allocates and manages resource numbers (RNs). System resources are areas within files, I/O devices, areas of memory, even other programs or operating system routines; anything that several programs might share.

You can think of RNs as flags, like semaphore flags (on some operating systems, RNs are called semaphores). When a program possesses one of the flags, the programs with which it cooperates know that the shared resource is being used. Because the programs cooperate, they will not try to use the resource until it is free. RNs cannot prevent programs from failing to cooperate, but they do give programs a mechanism for cooperation.

It is important to remember that RNs are never associated with specific system resources. It is entirely a function of the cooperating programs. If the programs share one or more files, then the RN indicates which program is accessing the file or files. If the programs share an I/O device, an RN can be used to indicate which program is writing to the device.

Often, several programs need to share a resource, but only one at a time should use it. An example is a data base file, where several programs can write to it or read from it, but only one at

a time should access it. Reading programs should always read from a stable file, and writing programs should not try to write to the same area as another program.

RNs are most useful when:

- Two or more programs use the same subroutine or change the contents of a disk file or area in memory
- One or more programs make decisions based upon the contents of a file or memory area that can be changed by any other program

The following terms have special meanings when used to describe RN resource management:

Allocate: assign an RN to a program. RNs are allocated when a program makes an RNRQ allocation call. There are two kinds of allocation: local or global. A locally allocated RN belongs to the program that made the RNRQ call and can be deallocated only by this program. The system automatically deallocates locally allocated RNs when the calling program terminates. Other programs can use a locally allocated RN only if the original program gives them access to the RN; however, these programs cannot deallocate the RN. Globally allocated RNs can be used by any program and be deallocated by any program; they are not automatically deallocated.

Deallocate: free an RN to the system by ending the RN's association with a program.

Lock: prevent other cooperating programs from trying to use a resource. An RN can be locked to only one program at a time. There are two kinds of locks: local and global. A local lock can be unlocked only by the original locking program. A global lock can be unlocked by any program.

Unlock: release an RN to indicate to a cooperating program that a shared resource is now available.

RNs are managed by the RNRQ subroutine. RNRQ asks the operating system to allocate or lock an RN to the calling program, or to deallocate or unlock an RN. The calling program owns a locally allocated RN until the program terminates, or deallocates or unlocks the RN. The program can pass the resource number to other cooperating programs that need to use the resource.

The RNRQ Call

The calling sequence for RNRQ is:

```
CALL RNRQ(cntwd,rn,stat)
```

where:

cntwd is a one-word integer variable that specifies the function of the RNRQ call:

Function	Bit	Octal value
No-wait	15	100000
No-abort	14	40000
Deallocate RN	5	40
Allocate globally	4	20
Allocate locally	3	10
Unlock	2	4
Lock globally	1	2
Lock locally	0	1

rn is a one-word integer that returns the allocated RN on allocation requests, or specifies the RN for lock, unlock, or deallocation requests. A returned value of zero indicates that an RN was not allocated.

stat is a one-word integer that returns the status of the RN:

```
-1 = invalid request
 0 = RN deallocated as requested
 1 = RN allocated/unlocked as requested
 2 = RN locked locally to calling program as requested
 3 = RN locked globally as requested
 4 = no RN available now
 6 = RN not allocated or requested RN locked to another program
 7 = requested RN is locked globally
```

For example, the following program fragment contains RNRQ calls that allocate an RN locally, lock it locally so that the program can let its cooperating programs know that the shared resource is being used, unlock the RN to free it for the other programs, and deallocate the RN.

```
CNTWD = 10B                ! allocate locally
CALL RNRQ(CNTWD,RN,STAT)
CNTWD = 1                  ! lock locally
CALL RNRQ(CNTWD,RN,STAT)
   :
use the resource
   :
CNTWD = 4                  ! unlock the RN
CALL RNRQ(CNTWD,RN,STAT)
CNTWD = 40B                ! deallocate the RN
CALL RNRQ(CNTWD,RN,STAT)
```

If the RN is locked to another program when the second allocate request is made, the program is suspended because the no wait bit is not set. If the no wait bit is set, the program continues, with the value 6 returned in the STAT parameter, and the value 0 returned in the RN parameter. Note that if the RN is already locked to the calling program, the call returns successfully.

The no-abort bit alters the error return point of the call when an error occurs:

```
$ALIAS  RNRQ,NOABORT
CALL   RNRQ(CNTWD,RN,STAT,*888)
```

The no-abort error return is established by setting bit 14 to 1 in the request control word (CNTWD). This causes the system to take the error branch if an error occurs. If no error occurs, the system returns normally, and the calling program continues. The $ALIAS directive is necessary to use the error return with system library calls.

Order of Precedence

More than one bit in *cntwd* can be set at a time. When several bits are set, the selected functions are executed in the following order:

1. Local allocate (bit 3)
   or
   Global allocate (bit 4)
2. Deallocate (bit 5) and return
3. Local lock (bit 0)
   or
   Global lock (bit 1)
4. Unlock (bit 2) and return

Only one of each of the choices for steps one and three is executed — if local locking or allocation is selected, global allocation or locking is not allowed. A single call may allocate, deallocate, lock, and unlock an RN.

The number of available RNs is fixed at system generation. If a program requests an RN when no more are available and the no wait bit is not set, the program is suspended until an RN is free. If the no wait bit is set, an RN is not allocated; therefore, the system returns a zero in the RN parameter, and the value 4 in the STAT parameter.

If the allocation is successful, the system returns the number of the allocated RN in the RN parameter. The allocated RN then is used in lock and unlock requests by cooperating programs to indicate when the shared resource is being used.

Resource Number Considerations

The three most important facts to remember when using RNs are as follows:

1. The association between an RN and a shared resource exists only in the programs that use the resource. The RN is associated only with the program to which it is allocated.

   Upon request, the system allocates RNs, locks them to programs and prevents other programs from locking the same RN, unlocks and deallocates RNs, suspends programs that request RNs when there are none left and reschedules them when RNs are available, and enforces the local and global allocation rules. The system does not enforce the association of RNs and resources.

2. All programs that access a shared resource must cooperate with each other using the RNs. The programs must agree on the RN-to-resource associations and must not access the resource when another program has locked an RN. The programs must lock an RN before accessing its resource to let cooperating programs know that the resource is being used and unlock the RN when finished to make the resource available to the other programs.

3. Unless the program is terminated saving resources, the system automatically unlocks all RNs that are locally locked when the calling program is aborted or terminated.

RNs can control access to a single file or to a group of files. To use as few RNs as possible, you can use one RN to control access to a group of files. While this method does make RNs available to other programs, it can be inefficient because the RN for the entire group of files is locked when any of the files is accessed. Programs may have to wait for the RN to be unlocked even if the file the program needs is not busy. If a separate RN is used for each file, less time is lost waiting because only when two programs need the same file is there a conflict. If such a conflict seems likely, the system manager should make a large number of resource numbers available at system generation to make the use of RNs more efficient.

You should decide whether execution speed or the availability of RNs is more important when deciding whether to use one or several RNs for a group of files. A single RN should always be used for a group of files if all of files must be updated before any of the files are accessed by other programs.

The system does not always allocate RNs in the same order from one series of RNRQ calls to another. The RNs that are assigned can be changed by other programs that use the RNs, by modifications in the system initialization, and by a new system generation. Therefore, your programs should always check with the program that makes the RNRQ calls to find out which RNs represent which resources. A program cannot assume that RN 6, for example, always represent the same file.

For each application that uses RNs, an RN initialization program can be run to allocate RNs for the application. The RNs can be stored in a file or a memory common block, or the RNs can be passed directly to the application programs.

Race Conditions

Sometimes programs that share a file or system common variables execute correctly when run independently, but do not execute correctly when run simultaneously. The intermittent errors that can occur when the programs interfere with each other are called race conditions. Race condition errors are often difficult to solve because they are sensitive to timing, and, therefore, hard to reproduce.

The following example shows how a race condition can occur:

```
PROGRAM A                    PROGRAM B
COMMON J                     COMMON J
IF (J .EQ. 2) J = J + 1      IF (J .EQ. 2) J = J + 3
:                            :
:                            :
```

Programs A and B share the system common variable J. J is set to 2 before A or B execute. Program A begins execution but is interrupted by the higher-priority program B after executing the IF statement but before executing the statement J = J + 1. Because J is still 2, program B executes its IF statement and J = J + 3. Program B continues with its other tasks and terminates. Program A then continues, and executes the statement J = J + 1.

If program A ran alone, it would terminate with J = 3. If program B ran alone, it would terminate with J = 5. Together, programs A and B terminate with J = 6. Any other program that uses J (perhaps the program that initialized J to the value of 2 and scheduled programs A and B) does not get the expected results.

Program priority cannot be relied upon to avoid race conditions. There are many conditions under which low-priority programs and higher-priority programs can interfere with one another. The correct way to avoid race conditions is to use RNs to help the programs coordinate their accesses to data.

Deadly Embrace

There is a condition related to race conditions that is known as a deadly embrace. The following sequence describes the steps that cause a deadly embrace:

1. Program A successfully locks RN 5 to indicate that it needs to access a data file.
2. Program B successfully locks RN 7 to indicate that it needs to access the printer.
3. Program A tries to lock RN 7 to indicate that it needs to access the printer.
4. Program B tries to lock RN 5 to read data from the file, but is suspended because RN 5 is locked to program A.

Deadly embrace conditions are easier to discover and repeat than race conditions. Deadly embrace is also easily avoided:

- If several files are to be updated together, all of their RNs should be successfully locked before any file is updated.
- When two or more RNs are to be simultaneously locked, use lock requests with no wait for the second and subsequent requests. This prevents the calling program from being suspended by the system with one or more RNs locked to it. It is best to avoid having RNs locked to a suspended program.
- The calling program should continue to try the lock until the RN is free. The calling program can access the resources when all necessary RNs are locked.
- In many applications, it is important to keep as many RNs free as possible. While the calling program waits for a locked RN to become available, the program should unlock any RNs that it had locked, so the RNs can be used by other programs. When the RN the calling program needs is free, the RNs released by the program can be locked again.

In summary, if a program must lock more than one RN and finds one or more of the RNs already locked, the program should release the RNs that it has already locked (if necessary), wait for the RN to become available, and again try to lock all needed RNs. The program must not fully or partially update any files, unless all the RNs locked that control access to the file and any related files that must be updated simultaneously are locked.

LURQ (Logical Unit Lock)

The LURQ subroutine lets a program have exclusive access to an I/O device. The calling sequence is:

```
CALL  LURQ (option [ , luary [ , numlus [ , keynum ] ] ] )
```

where:

option is a one-word integer variable that determines whether the LURQ call locks or unlocks LUs.

luary is an integer array of LUs to be locked or unlocked.

numlus is a one-word integer variable that specifies the number of LUs in LUARY.

keynum is a one-word integer variable that returns a key number to let the calling program share its locked LU with other programs.

Returns:

A-Register: 0 = lock successful.
1 = one or more LUs are already locked.

This request gives the calling program exclusive access to an LU until the program removes the lock. Any program that tries to use or lock the locked LU is suspended until the lock is removed. The locking program can pass its key number to other programs to let them use the locked LU. The other programs can then access the device by specifying the key number in their standard and class I/O calls.

LU locks remain in effect until the calling program terminates or unlocks the LUs, unless the program terminates saving resources, in which case the locks remain in effect. The LU locks only can be removed by running the calling program again to unlock the locked LUs.

Caution It is legal to lock the disk that contains the swap file and program files. If the disk is locked, RTE-A cannot swap programs out of memory, or load programs or program segments from disk into memory. All programs that need to use the disk are LU lock suspended until the disk LU is unlocked.

LURQ Parameters

The *option* parameter can be set to any of the following values to select the lock or unlock functions:

0XX001B — Lock with wait. Lock the LUs specified in *luary* and *numlus*, but suspend the calling program if any of the LUs in *luary* are locked to other programs. As the other programs unlock the needed LUs, the system locks them to the calling program. When all of the requested LUs are locked, the calling program is automatically rescheduled. If omitted, *numlus* defaults to 1, meaning that only the first LU in *luary* is locked.

Note If the calling program has any LUs already locked, the lock with wait call unlocks them, and then locks the LUs in *luary*. To prevent locks from being lost, use the lock without wait option.

1XX001B — Lock without wait. Lock the LUs specified by *luary* and *numlus*, unless any of the specified LUs are already locked to another program. If any are, return without locking any of the LUs. The A-Register contains the value 1 after such a call to indicate that at least one LU was already locked.

0XX000B — Unlock the LUs specified in the *luary* and *numlus* parameters. If omitted, *numlus* defaults to 1 meaning that only the first entry in *luary* is unlocked.

1XX000B — Unlock all LUs locked by the calling program. The *luary* and *numlus* parameters are ignored; therefore, they can be left out of the call.

In all of the above, the value of XX depends on the setting of bits 10, 12, and 14. The meaning of these bits is described below.

Bit 14: No-abort option. If an error occurs in processing the LURQ call, return to the program for error processing.

```
$ALIAS  LURQ,noabort

   LUARY(1)  =  6
   OPTION  =  040001B
   CALL  LURQ  (OPTION,LUARY,1,*100)
```

The $ALIAS compiler directive (required in FORTRAN programs for no-abort error processing), the no-abort bit, and the *100 special error return label cause a return to the main program at the statement labeled 100 for error processing.

The program is aborted if the no-abort bit is not set and an error occurs in processing the LURQ call.

Bit 12: LU 1 override option. If this bit is set, the system LU 1 is locked instead of the session LU 1.

Bit 10: Spool node lock override option. Spooling or I/O redirection in effect for any LUs is overridden and the specified LUs are locked. If your system does not have spooling, bit 10 is undefined.

When bit 10 is not set, spooling or redirection is observed. If LU 6 is redirected to LU 8, then a lock or unlock request for LU 6 locks LU 8. If LU 6 is spooled, then the lock or unlock request for LU 6 locks the spool file, not LU 6. LU 6 is not affected.

luary is an integer array of LU numbers to be locked or unlocked.

numlus specifies the number of LUs in *luary*. If omitted, *numlus* defaults to 1 meaning that only the first entry in *luary* is locked or unlocked. The value of *numlus* must not exceed the size of *luary*.

keynum returns the key number for the lock to the calling program. The returned value can be passed to other programs to allow access to a locked device.

Unlike RNs, LU locks do not require program cooperation. The system establishes direct associations between locks and LUs, and does not permit programs to ignore the LU locks.

Deadly Embrace

When two or more programs employ LU locking, a condition known as a deadly embrace or deadlock can occur. The following example shows how a deadly embrace can occur:

1. Program A locks LU 8, the magnetic tape, and begins to read from it. Program A is I/O suspended while the read proceeds.
2. Program B locks LU 6, the line printer, and tries to lock LU 8, which is already locked by program A. The system suspends program B to wait for LU 8 to be unlocked.
3. The read request to the magnetic tape completes. Program A is rescheduled, and tries to lock LU 6 to print the data from the magnetic tape. LU 6 is already locked by program B, so the system suspends program A to wait for LU 6 to be unlocked.
4. The programs are now in a deadly embrace, because each is suspended, waiting for the other to unlock an LU. Neither can unlock the LUs, so both are hung until you intervene.

Deadly embrace can be avoided easily by performing all LU locks in a single LURQ call before accessing the devices. With this technique, there is no time that the program is suspended with locked LUs; either it is suspended or it is running with all its LUs locked.

Figures 2-1 through 2-3 show programs A and B, and their deadly embrace.

```
          PROGRAM A
          IMPLICIT INTEGER (A-Z)
          DIMENSION LUARY (2), INBUF(40), NAME(1)
          DATA NAME /'B '/

C    Read data from the mag tape, and
C    display the data on the printer.
C
C    (1) Lock the mag tape (LU8) and
C        schedule program B
C    (2) Read the data
C    (3) Lock the line printer (LU 6)
C    (4) Print the data

          OPTION = 100001B
          LUARY(1) = 8
C (1)
          CALL LURQ (OPTION, LUARY(1), 1)
          CALL EXEC (10, NAME)
C (2)
          CALL EXEC (1, LUARY(1), INBUF, 40)
C (3)
          LUARY(2) = 6
          CALL LURQ (OPTION, LUARY(2), 1)
C (4)
          WRITE (LUARY(2), 6) INBUF
 6        FORMAT ("Mag tape info is "/,40A2)

          END
```

Figure 2-1. Program A, Deadly Embrace Example

```
                PROGRAM B (3,70)
                IMPLICIT INTEGER (A-Z)
                DIMENSION LUARY(2), INBUF(40)

C     Read data from the mag tape and
C     display the data on the printer.
C
C     (1) Lock the line printer (LU 6)
C     (2) Lock the mag tape (LU 8)
C     (3) Read the data
C     (4) Print the data

                OPTION = 100001B
                LUARY(1) = 6
C (1)
                CALL LURQ (OPTION, LUARY(2), 1)
C (2)
                LUARY(2) = 8
                CALL LURQ (OPTION, LUARY(2), 1)
C (3)
                CALL EXEC (1, 8, INBUF, 40)
C (4)
                WRITE (LUARY(2), 6) INBUF
 6              FORMAT ("Mag tape info is"/,40A2)

                END
```

Figure 2-2. Program B, Deadly Embrace Example

```
CI> wh
Program                    DataPartition  CodePartition
Name       Pr      PC Seg  Size Status    Size Status   Program Status
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Session    46    Superuser MANAGER
CI         51    24301      32 in                      waiting for WH
WH          5     6336      12 in                      scheduled
B          70        0       4 in                      dormant
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Tue Mar 1, 1983 11:42 am
CI> xq a
CI> wh
Program                    DataPartition  CodePartition
Name       Pr      PC Seg  Size Status    Size Status   Program Status
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Session    46    Superuser MANAGER
CI         51    24301      32 in                      waiting for WH
WH          5     6336      12 in                      scheduled
B          70    43372       4 in                      lockd dev susp on lu 8
    LU 8 is locked to A
A          99    45273       4 in                      lockd dev susp on lu 6
    LU 6 is locked to B
    **Deadlock**
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Tue Mar 1, 1983 11:42 am
```

Figure 2-3. WH During Deadly Embrace

LIMEM (Find Memory Limits)

LIMEM returns the starting location and size of the memory area between the end of the program or its stack area and the end of the program partition. LIMEM lets the program use this spare memory. Note that the size of the memory area returned by LIMEM does not include any EMA memory that the program may have.

For programs that use the code and data separation features of VC+ (CDS programs), the memory area reported by LIMEM begins after the stack area in the data segment partition. The size of the area is determined by the size of the data partition selected when the program was linked, not by the physical size of the memory partition in which the program is actually executing.

For non-CDS programs, the area reported by LIMEM begins after the end of the calling program. Its size is also determined by the size of the partition selected when the program was linked.

For both CDS and non-CDS programs, the physical memory partition in which the program executes can be larger than the program size set when the program was linked.

LIMEM Calls

For CDS programs, the format is as follows:

```
CALL LIMEM(code,fwam,words)
```

where:

code is ignored.

fwam is a one-word integer variable that returns the address of the first word after the end of the stack area in the data partition.

words is a one-word integer variable that returns the size of the area between *fwam* and the end of the data partition, expressed in words.

For non-CDS programs without overlays, the format is as follows:

```
CALL LIMEM(code,fwam,words)
```

where:

code is ignored.

fwam is a one-word integer variable that returns the address of the first word after the end of the program code.

words is a one-word integer variable that returns the size of the area between *fwam* and the end of the program partition, expressed in words.

For non-CDS programs with overlays, the format is as follows:

```
CALL LIMEM(code,fwam,words[,curnt[,cwrds]])
```

where:

code is ignored.

fwam is a one-word integer variable that returns the address of the first word after the end of the longest program overlay.

words is a one-word integer variable that returns the size of the area between FWAM and the end of the program partition, expressed in words.

FWAM and WORDS describe the largest area that is guaranteed to be free at all times of the program's execution.

curnt is a one-word integer variable that returns the address of the first word after the end of the current overlay.

cwrds is a one-word integer variable that returns the size of the area that begins at *curnt*, expressed in words.

curnt and *cwrds* describe the area of memory that is guaranteed to be free only while the current overlay is executing.

Do not use the same variable for any of the following parameters: *fwam*, *words*, and *curnt*. If the same variable is used in any combination of the three, then incorrect results occur. The calculation of *cwrds* is dependent on all three.

Do not use the LIMEM subroutine in Pascal programs. The LIMEM subroutine interferes with Pascal's use of the Heap/Stack area.

LIMEM Details

The size of the area of memory reported by LIMEM is determined by the size of the partition that was specified or defaulted when the program was linked. The size of the partition in which the program actually executes does not determine the amount of available memory. The memory that exists between the end of the partition size set at link time and the end of the actual partition is not available to the program.

LIMEM does not change the size of the program or its partition. LIMEM only reports the location and size of the available memory. For CDS programs, the program size is set by the LINK HE command, and can be changed by the DT operator command after the program has been linked and restored by the RP operator command.

LIMEM uses an EXEC 26 call to determine the memory information.

3

Standard I/O

Standard I/O Requests

In general, all program requests for standard I/O operations are coded according to formal call sequences shown in Chapter 1. These EXEC and system library calls provide for nonbuffered or buffered data transfers as well as I/O control and status requests. The following is a summary of requests used in the standard I/O function group.

Note All parameters are single-word integers or, when specified in the call description, integer arrays (buffers).

Standard I/O requests can be made in one of two forms: buffered or nonbuffered. Nonbuffered I/O requests allow the calling program to operate synchronously with its requests. This allows all available I/O status associated with a particular request to be returned to and acted upon by the calling program. Programs issuing requests to nonbuffered devices are suspended for the duration of the request. Suspended programs cannot be swapped. This is always true for read requests because the data transfer is from the device to your program buffer.

A program can establish nonbuffered operation for any I/O requests it issues. This request is made through a special control parameter bit, (bit 14 in *cntwd*), which overrides whether or not the device was specified as a buffered device when the system was generated. (Note that this does not affect the buffered operation of requests from other programs.)

When a program is guaranteed a nonbuffered operation, bit 13 in the control word can be set to allow the total error handling/recovery procedures to be defined and handled by your program.

It is helpful to use bits 13 and 14 together because status information and device error conditions are not available using buffered operations. By setting bits 13 and 14 at the same time, you ensure nonbuffered operation and can obtain device status and error information.

As a general rule, buffered operation on a device is used unless any of the following conditions are true:

- The request is a read.
- The device was specified as nonbuffered when the system was generated.
- Bit 14 in the control word parameter is set on the request.
- There will never be enough SAM available for the request.
- There is not enough SAM currently available and the no-suspend bit was set in the ECODE for the request.

Buffered write and control requests offer the advantage of continuing execution of the calling program in parallel with the I/O operation. In addition, the program remains swappable while the buffered operation progresses. This means that the calling program is capable of continued execution without a need for returned status from these I/O requests.

Returned device status often includes transmission log, error status, and available device status.

You should understand buffer limits when using buffered I/O, especially if large buffers are used; buffer limits are discussed in the *RTE-A System Design Manual,* part number 92077-90013.

Handling Device Errors

You can use Standard I/O in two ways to handle device errors. The first way is to let the operating system take care of the device error. When the operating system handles device errors, the following actions are taken:

1. The device is downed.
2. I/O is suspended.
3. You are alerted.

After the device problem has been corrected, you can up the device, which reissues the suspended request and resumes I/O operations. Using buffered I/O, your program is allowed to continue its execution during this I/O error recovery process.

The second way to handle device errors is to set bits 13 and 14 in the control word parameter. In this case, your program can interrogate the device. The device is not downed, the I/O request is not suspended, and you are not alerted. Programs using this method to handle device errors should be designed to perform some type of device diagnostics. In general, your program needs to check the A- and B-Registers to determine if an error occurs. Note that the error return, even if specified, will not be taken.

I/O and Swapping

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

- The request is a control request with no buffer.
- The device is buffered, the request is for output or control.
- The device is down.
- The LU is locked to another program.

EXEC 1 and 2 (Read and Write)

EXEC 1 and 2 allow the reading or writing of a specified number of words or characters to a device.

```
CALL EXEC(ecode,cntwd,bufr,bufln[,pram3[,pram4[,0,0,keynum]]])
```

where:

ecode is the request code: 1 for read, 2 for write.

bufr is the buffer. For read operations (*ecode*=1), this is the array where the system returns data. For write operations (*ecode*=2), *bufr* is the array containing the data to be written.

bufln is the buffer length. A positive value indicates the number of words; a negative value is the number of characters in *bufr*. When an array of type REAL is transmitted, the buffer length must be the total number of two-byte words required, which is two times the array length for standard precision or four times the array length for double precision. *bufln* must be set to zero when *pram3* and *pram4* are used to pass commands for direct I/O.

pram3 is an optional parameter or optional buffer.

pram4 is an optional parameter or optional buffer length.

keynum is the key number of the locked LU, corresponding to the *keynum* parameter returned by LURQ. See LURQ in Chapter 2 for more details.

cntwd is the control word:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BB	NB	UE	Z	X	TR	X	EC	X	BI	LU					

BB is the device driver bypass bit
NB forces nonbuffered operation
UE allows your program to handle device errors
Z specifies optional parameters *pram3* and *pram4* as a buffer and buffer length
TR establishes transparency mode in effect
EC establishes echo of input
BI selects ASCII or binary data transfer
LU selects device by logical unit number in the range 0-63. For LUs > 63, use XLUEX.
X is defined for the appropriate driver

Read/Write Parameters

In the *cntwd* control word:

LU (bits 0-5) is the logical unit number of the device where data is read from or written to. The range is from 0 through 63, inclusive. (LU 0 is the bit bucket.) For logical units greater than 63, refer to the XLUEX call later in this chapter.

BI (bit 6) indicates whether binary (BI=1) or ASCII information (BI=0) is to be transmitted.

EC (bit 8) indicates if echo is in effect. For echo mode (EC=1), keyboard input is displayed as received. This is the normal mode of operation for terminal type devices. For non-echo mode (EC=0), keyboard input is not displayed.

TR (bit 10) indicates whether transparency mode is in effect. When TR=0, the transparency mode is turned off, and terminators and/or embedded control characters may be removed or added by the driver on input or output (for example, a carriage return/linefeed may be added on a write to a terminal). When TR=1, the transparency mode is turned on and driver addition or removal of information is prohibited.

Z (bit 12) when set, indicates that optional parameters *pram3* and *pram4* specify a buffer and buffer length where special additional driver/device information can be supplied. If the Z bit is set, the program is not swappable. See description of *pram3* and *pram4* below.

UE (bit 13) is the user error-handling bit. When set (UE=1), it overrides normal operating system handling of device errors. If a device error occurs, the calling program is allowed to resume execution, the device is not downed, and a message is not displayed on the system console. If not set (UE=0) and a device error occurs, the device is downed, IO is suspended, and a message is displayed on the system console.

NB (bit 14) is the nonbuffered bit. If set (NB=1), the associated request operates in nonbuffered mode. It should be set whenever the UE bit (bit 13 in the control word) is set.

BB (bit 15) is the device driver bypass bit. If set (BB=1) the device driver is bypassed. In general, you should not set this bit.

X (bits 7, 9, 11) are defined for the appropriate driver. Refer to the *RTE-A Driver Reference Manual*, part number 92077-90011, for details.

Optional parameters are:

pram3 and *pram4* supply additional information depending upon the requirements of the driver. When the Z bit is set in the control word, *pram3* and *pram4* are used to define the Z-buffer. *pram3* defines the Z-buffer and *pram4* defines its length. If the request is to a disk-type device (driver types 30 through 37), *pram3* and *pram4* must be specified even if the device driver is bypassed; otherwise, an IO01 error occurs. Note that when *pram3* and *pram4* are used to pass commands, *bufln* must be set to 0. Refer to the *RTE-A Driver Reference Manual* for details on the use of these parameters.

0 (zero) in an EXEC 1 or 2 call sequence is a formal placeholder that must be supplied when the *keynum* parameter is needed.

keynum is a key number assigned by the system to a locked LU when a request is made using the LURQ call. This value is returned from the LURQ request and can be supplied in EXEC 1 and 2 requests to allow access to a locked device. Thus a locked LU can be shared among cooperating programs. Note that the program that issued the LURQ request is never required to supply this parameter when making its own I/O request. (See the LURQ section in Chapter 2.)

Read/Write Requests

Requests for data transfers to any device always require the first four parameters, thus identifying the transfer direction, the device and its control details, and the data buffer and length. The remaining parameters are optional. The need and use of parameters *pram3* and *pram4* are dictated by the specific device or driver called upon to perform the I/O operation. For example, a disk driver can be expected to use *pram3* and *pram4* as simple parameters defining disk track and sector. Access to the HP-IB driver might use *pram3* and *pram4* as a command buffer, and buffer length defining general HP-IB commands. For any read/write request, *pram3* and *pram4*, as needed, always supply information from the calling program to the I/O device driver. Specific details will be found in the *RTE-A Driver Reference Manual*.

A- and B-Register Returns

The operating system puts end-of-operation information for reads and nonbuffered writes in the A- and B-Registers. The A-Register contains word 6 of the DVT (see the EXEC 13 status section for details of this word).

The B-Register contains a positive number that is the number of words or characters (depending upon program specification) actually transmitted. Thus, you can find the number of words or characters received on any input request by getting the contents of the B-Register. The number in the B-Register is always positive.

If, in the EXEC call, *bufln* is a negative number, the content of the B-Register is the number of characters received. If *bufln* is the positive number, the B-Register contains the number of words entered.

The registers are meaningless in output requests to a buffered device. If the system reads a record that is shorter than the original buffered request length, the unused portion of the user

buffer may have been modified by the request. (The extent and nature of the modification depends on which device and driver are used.) This implies that a buffer should not be pre-initialized unless the device type and driver characteristics are known.

EXEC Examples

This example illustrates an EXEC 1 read. The various options used are:

1. Setting the Z bit in the control word to make the request a write/read, or write/write request.
2. Setting the echo bit in the control word so that the information typed in will be echoed at the terminal.
3. Setting the LU (bits 0-5) where the read operation will take place.

Write/read is useful for writing a prompt to the display terminal before reading the response into the data buffer. Both of these operations can be accomplished in the same call.

When the EXEC request executes, the word "PROMPT>>" appears on the terminal. The driver then echoes the characters typed in and, upon receiving a carriage return, reads up to 80 characters.

```
      IMPLICIT INTEGER (A-Z)
      DIMENSION BUFR (40), PRMT (4)
      DATA PRMT /'PROMPT>>'/
      .
      .
C SET THE INPUT BUFFER LENGTH TO ALLOW FOR 80 CHARACTER
      BUFLN = -80

C SET THE OUTPUT BUFFER LENGTH (PRMTLN) TO PRINT 8 CHARACTER
      PRMTLN = -8

      CALL EXEC (1, 010401B, BUFR, BUFLN, PRMT, PRMTLN)
      .
      .
      END
```

The example below illustrates the use of EXEC 2 write. The routine writes a buffer of 40 words maximum to LU 11. The program reports the number of words that were actually written to the device and checks for errors and reports them.

```
      PROGRAM EX2
      IMPLICIT INTEGER (A-Z)
      DIMENSION BUFR(40)

C SET THE NO-ABORT BIT TO CATCH ERRORS
C WRITE TO LU 11
C SET THE nonbuffered BIT TO MAKE TRANSMISSION LOG RETURN VALID

      WRITS = 2 + 100000B
      CNTWD = 11 + 40000B
```

```
      CALL  EXEC(WRITS,CNTWD,BUFR,40,*9999)

C CHECK STATUS....(A REG CONTAINS DVT WORD 6, B TRANSMISSION LOG
      CALL ABREG (A,B)
      WRITE(1,1) B
1     FORMAT("NUMBER OF WORDS ACTUALLY WRITTEN ",I3)
      .
      .
      .
9999                        CALL ABREG(A,B)
      WRITE(1,2) A,B
2     FORMAT("ERROR ON WRITE = ",A2,A2)
      END
```

SYCON (Write Message to System Console)

The SYCON subroutine writes a message to the system console (System LU 1).

CALL SYCON(*ibuf*,*ilen*)

where:

ibuf is the buffer that contains the message to be written.

ilen is the length of *ibuf*; a positive value indicates the number of words and a negative value indicates the number of characters.

This routine overrides LU mapping and writes directly to system LU 1.

The Macro calling sequence is as follows:

```
        EXT     SYCON
         .
         .
        JSB     SYCON
        DEF     RTN
        DEF     IBUF
        DEF     ILEN
RTN      .
         .
```

EXEC 3 (I/O Device Control)

The EXEC 3 control call carries out various I/O control operations, such as backspace, write end-of-file, and rewind.

Various device control operations are accomplished through specific I/O control requests. This includes, for example, such operations as clear or reset device, or backspace record. All control requests require the first two parameters (*ecode* and *cntwd*) to identify the device and the desired control functions.

The remaining parameters (*pram1* through *pram4*) are optional, depending upon the needs of a device. If the I/O device is not buffered, the program is placed in the I/O suspend list until the control operation is complete.

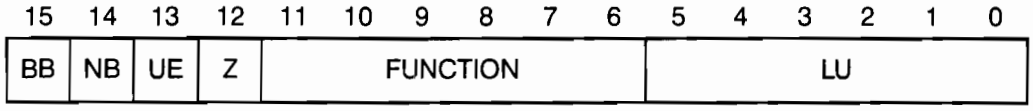
For specific device control information, refer to the *RTE-A Driver Reference Manual*, part number 92077-90011.

```
CALL EXEC(ecode,cntwd[,pram1[,pram2[,pram3[,pram4[,0,0,keynum]]]]])
```

where:

ecode is 3 for an I/O device control request.

cntwd is the control word:



BB	is the driver bypass bit
NB	forces nonbuffered operation
UE	allows your program to handle device errors
Z	specifies optional parameters *pram3*/4 as the Z-buffer and Z-buffer length respectively.
FUNCTION	specifies desired control function; see comments below.
LU	selects device by logical unit in the range of 0-63. For logical units greater than 63, use XLUEX call.

pram1, *pram2* are optional parameters.

pram3, *pram4* are optional parameters or the optional buffer and buffer length.

0,0 are formal placeholders required if parameter *keynum* is used.

keynum is the key number of the locked LU.

Computer Museum

I/O Device Control Parameters

cntwd (the control word) is essentially the same as in the EXEC 1 and 2 read and write call. The Z bit and optional parameters *pram1* through *pram4* are driver-dependent. If needed by the driver, *pram1* and *pram2* are sent. If the Z bit is set, *pram3* and *pram4* specify the Z buffer. *pram3* defines the buffer and *pram4* is the buffer length passed to the driver as required. Refer to the *RTE-A Driver Reference Manual* for more information.

If the device or request is nonbuffered and the Z bit is set, the program is not swappable.

The exception to EXEC 3's similarity with EXEC 1 and 2 is the FUNCTION field (bits 6-11) in *cntwd*. To determine the function codes defined for a particular device see the appropriate driver in the *RTE-A Driver Reference Manual.*

Optional parameters are:

pram1 - pram4 are driver-dependent parameters. Their use can vary considerably. With a control request of function code 23, for instance, *pram1* indicates enabling or disabling of an asynchronous interrupt response. See the appropriate section of the *RTE-A Driver Reference Manual* for details of the contents of each optional parameter.

keynum is a key number assigned by the system to a locked LU when a request is made via the LURQ call. This value is returned from the LURQ request and can be supplied in the above I/O requests to allow access to a locked device. Thus a locked LU can be shared among cooperating programs. Note that the program that issued the LURQ request is never required to supply this parameter when making an I/O request. See the LURQ section in Chapter 2.

A- and B-Register Returns

End of operation status for a nonbuffered request is returned in the registers as follows:

A-Register = Device status found in DVT word 6 (see EXEC 13 status section)

B-Register = Device status found in DVT word 17

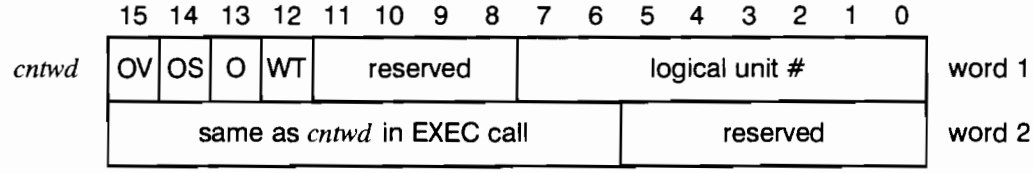
REIO (Buffered I/O)

REIO allows a program to be swapped while waiting for input or output. It is used exactly the same way that EXEC is used for request codes 1, 2, and 3, with the same parameters and return values. The advantage of REIO over standard EXEC I/O calls is that REIO uses Class I/O to force buffering on input from a buffered or nonbuffered device. Note that output requests to a nonbuffered device are not changed into a buffered request.

If the Class option is not included as part of the operating system or if there is not enough SAM available for the request, REIO calls become standard EXEC calls and the standard EXEC call restrictions apply.

XLUEX (I/O Extended Logical Unit EXEC)

In RTE-A, there may be up to 255 logical units. With standard EXEC calls, logical unit numbers greater than 63 may not be specified. The XLUEX (extended logical unit EXEC) subroutine solves this problem, 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.

If the OS bit is set, spooling to the logical unit is overridden; therefore, even if spooling to an LU is active, all writes to that LU go directly to the logical unit.

If the WT bit is set, the I/O operation writes through any pending read operation on the device. The read operation is aborted and restarted after the write operation completes. This allows a program to output urgent messages immediately and not wait until the read operation completes. When using write-through-pending-reads on buffered devices, you should understand how to use buffer limits, especially if large buffers are used or if numerous requests are sent to the devices. Buffer limits are discussed in the *RTE-A System Design Manual*, part number 92077-90013.

Bits 6-15 of *cntwd* word 2 are identical to bits 6-15 of the *cntwd* for the EXEC call.

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, OS, or WT bit is set.

When programming in FORTRAN and 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 *ecode* equal to 1, 2, 3, 13, 17, 18, 19, or 20. It is suggested that all new programs be written using XLUEX for these EXEC calls in order to access logical units greater than 63.

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)

      END
```

XREIO (Extended REIO)

XREIO allows for up to 255 logical units by having an extended (double word) *cntwd*. All other parameters and function of XREIO are identical to REIO. The format of the *cntwd* is the same as XLUEX.

AbortRq (Abort Current Request)

This subroutine aborts the current request (at the head of the queue) on the specified LU. AbortRq has the same effect as the "CN,*lu*,AB" command.

```
CALL AbortRq(lu)
```

EXEC 13 (Device Status)

The status of I/O operations is maintained by the operating system within the actual device and interface tables for each device in the system. Generally this information can be viewed as a momentary view of an I/O operation, in that the state of I/O conditions is constantly changing. The returned status consists of operating system maintained information and driver supplied information.

The operating system information is dynamic and reflects the state of events at the precise instant that the status request is executed. The driver-supplied information, on the other hand, normally reflects conditions as they existed at the time of the previous request. The calling program is not I/O suspended when the call is made, as this call is not an actual request to the driver.

If you want to keep the dynamic status from constantly changing, the device should be locked. See the LURQ section in Chapter 2 for more information on locking an LU.

The following steps might be done to ensure stable device status:

1. Lock the device
2. Check the device status
3. Perform the appropriate action on the device
4. Unlock the device

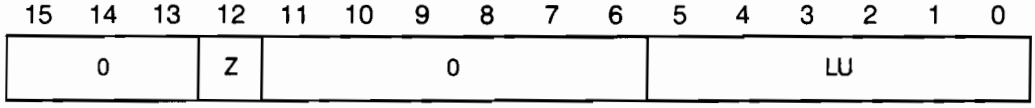
The EXEC 13 status request calling sequence is:

```
CALL EXEC(ecode,cntwd,stat1[,stat2[,stat3[,stat4]]])
```

where:

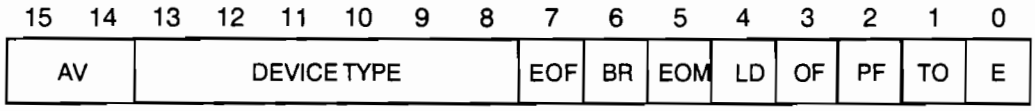
ecode is 13 for a device status request.

cntwd is the control word:

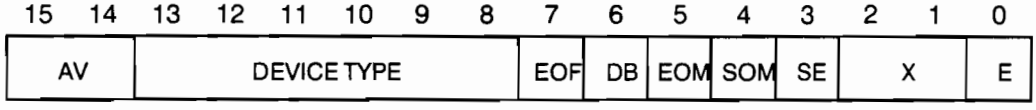


stat1 is the returned device status (DVT word 6):

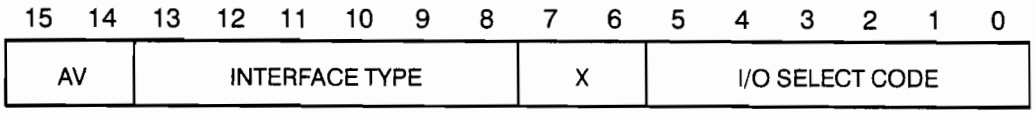
For DDC00/DDC01 Serial Drivers DVT6 is:



For all other drivers DVT6 is:



stat2 is the returned interface status (IFT word 6)



stat3 and *stat4* are:

If Z = 0, returned driver parameter words 1 and 2.

If Z = 1, buffer and length where information from the driver parameter area is returned.

Device Status Parameters

Two areas of *cntwd* (the control word) are valid for this call. Bits 0-5 specify the LU from which to receive status information. Bit 12, the Z bit, defines the usage of *stat3* and *stat4*.

If the Z bit is zero (0) and if *stat3* and *stat4* are included in the call, they respectively contain driver parameter words 1 and 2 from the DVT area.

If the Z bit is set to one (1) and if *stat3* and *stat4* are included in the call, they respectively define a buffer and buffer length. In this way, any number of driver parameters can be obtained from the DVT area.

All other bits in the control word have no meaning for this call.

stat1 is the returned device status word (DVT word 6) which provides information on the particular device's state. Note that this same information is provided at the end of any nonbuffered READ, WRITE, or CONTROL request in the A-Register. The fields of this word have the following meanings:

For DDC00/DDC01 Serial Drivers:

AV is the device availability:

0 = driver is free to process a new request
1 = device has been set down by you or the driver
2 = driver is currently processing an I/O request
3 = down but busy with a request (normally only occurs if you down an active device)

DEVICE TYPE

is a logical 6-bit value used to describe the type of device associated with the current DVT. All device type values are initially established at generation. Shown below are the device types for the devices which may be included in a typical system:

Type Number	Device Type
00-07	Terminal
12	Line Printer
20	264x Minicartridge
23	Magnetic Tape
24	Streaming Tape Drive
26	CS/80 Tape Drive
30	Floppy Disk
31	Mirrored Disk
32	MAC Disk
33	CS/80 Disk
34	Short Cylinder MAC Disk
35	Multi-CPU MAC Disk
36	PROM
37	HPIB
41	Writable control store
50-60	A/D or Parallel card
64	Multipoint slave
66	DS/1000-IV PSI card
70-77	Instrument

The remaining device status bits are as follows:

EOF is set if end-of-file condition has been detected (set by CTU drivers only).

BR is set if a break character is detected on the received data line.

EOM is set if end-of-medium has or will position past the physical media, such as attempting to write 2 disk tracks with only 1 remaining (set by detection of EOT in normal ASCII read).

LD is set if the communication line is down, valid for modem lines after first connect. Also, it is set if there is a speed sensing failure for ID400/ and ID800/801.

OF is set if there is an overflow error; the application is losing data.

PF is set if there is a parity error or frame error in the data.

TO is set if there is a timeout by the device driver.

E is an error flag that is set if any DVT16 error bits are set.

For all other drivers:

AV as described previously for the DDC00/DDC01 Serial Drivers.

DEVICE TYPE
as described previously for the DDC00/DDC01 Serial Drivers.

The remaining device status bits are as follows:

EOF is set if end-of-file condition has been detected.

DB is set if the device is busy. This would indicate that the device is performing some function which prevents other operations from starting, such as cartridge tape rewind.

EOM is set if end-of-medium has or will position past the physical media, such as attempting to write 2 disk tracks with only 1 remaining.

SOM is the start-of-medium indicator, set to indicate the media is at the start of the recording area.

SE is a "soft" error indicator, set when some difficulty has been encountered during the eventual successful completion of a request. This bit would be set, for example, if a successful disk read had been performed but one or more retries had been necessary to read the data.

E is an error flag, set whenever a driver signifies a "hard" error condition which prevents the completion of a request (such as timeout, not ready, and write protected). For any error of this nature, the appropriate error message is displayed on the system console (unless the UE bit is set). See the section on RMPAR. Whether the device has also been set down or not can be determined by examining the AV bits (15 and 14), described previously.

X is where driver or device dependent status information is returned.

Optional parameters are:

stat2 is the returned interface status which provides the following information:

AV is the interface availability:

0 = free, no operation is in progress
1 = locked to a device driver for future (momentary) operation
2 = busy with a device driver request
3 = locked to a device driver and busy with a request

INTERFACE TYPE
is a logical 6-bit value used to describe the I/O interface card that a device or a set of devices are connected to. All interface type values are initially established at generation. The interface types for a typical system are:

Type Number	Interface Type
00	Asynchronous serial interface card
27	Integrated disk
36	PROM
37	HP-IB
50	Parallel interface card
66	Network interface card

I/O SELECT CODE
is the particular interface card to be addressed. This is set on the card by switches and is an input to the generator.

X is reserved.

stat3, stat4 are optional parameters that can be used to return specific device configuration information from the $DVTP area of the device table (DVT). When these parameters are supplied in the status call sequence and Z=0, then the first word of $DVTP area is returned at *stat3*, and the 2nd word of $DVTP is returned at *stat4*. If Z=1, then *stat3* and *stat4* describe the Z-buffer and buffer length. *stat4* number of DVTP words are returned in the Z-buffer area *stat3*.

A- and B-Register Returns

The returned contents of the A- and B-Registers are undefined if the request is successful. Error information is returned to the A- and B-Registers for unsuccessful calls.

EXEC Status Examples

The following examples show legal formats for some status calls.

```
IMPLICIT  INTEGER(A-Z)
   :
CALL  EXEC(13,1,STAT1,STAT2)
   :
CALL  EXEC(13,6,STAT1,STAT2,STAT3)
   :
CALL  EXEC(13,10007B,STAT1,STAT2,BUFR,BUFLN)
```

The next two examples are illegal call formats. This is because the Z bit is set in the control word and thus the driver expects a buffer and buffer length to be defined.

```
CALL  EXEC(13,10007B,STAT1)

CALL  EXEC(13,10007B,STAT1,STAT2,BUFR)
```

RMPAR (Extended Status)

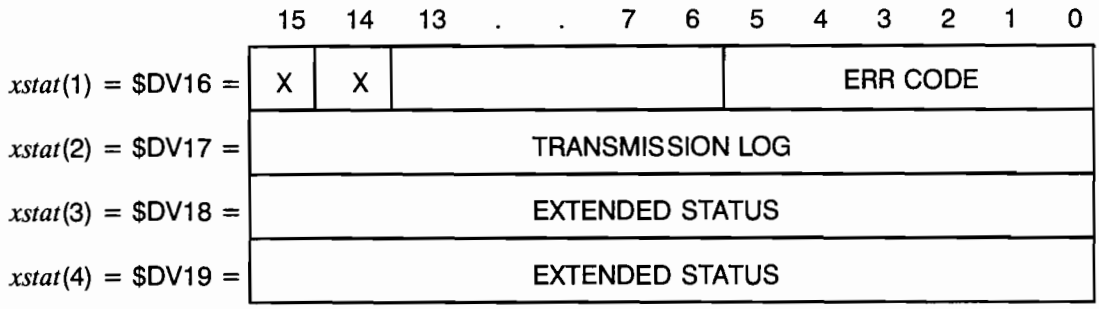
In addition to the general status provided through EXEC 13 requests, special extended driver/device status is sometimes available depending upon the nature of the device driver. When applicable, this information can be recovered by a call to the subroutine RMPAR immediately after a nonbuffered READ, WRITE, or CONTROL request.

At completion of a nonbuffered I/O request, the system moves driver-provided information from words 16 through 19 of the device table (DVT) to the temporary word locations in the ID segment of the calling program. Because these temporary words are used for many purposes, the validity of this information is maintained only as long as the calling program does not cause the temporary words to be overwritten. The words will not be overwritten if the call to RMPAR immediately follows the I/O request.

Call sequence is as follows:

```
IMPLICIT INTEGER(A-Z)
DIMENSION xstat(5)
:
CALL EXEC(1 or 2 or 3, ...)
CALL RMPAR(xstat)
```

The information returned in the *xstat* array is as follows:



xstat(5) is not used

X is a reserved driver/system interchange area.

ERR CODE
is a 6-bit error indication describing the particular type of device error detected. Note that any time ERR CODE is set to a non-zero value, the E-bit returned in the device status word (in the A-Register after every I/O request or *stat1* from the EXEC 13 status request) is set. Error codes are:

Error	Meaning
0	No Error
1	Illegal Request
2	Not Ready
3	Timeout

4	End of Tape
5	Transmission Error (Parity)
6	Write Protected
7	Addressing Error (HP-IB)
8	Serial Poll Failure (HP-IB)
9	Group Poll Failure
10	Fault (Such as Disk)
11	Data Communication Error
12	Generation Error (Check DVTX or DVTP)
13-19	Reserved
20-59	Driver Definable Error Condition (Refer to the *RTE-A Driver Reference Manual*)

TRANSMISSION LOG
: is the number of actual words or characters transmitted for a read or write request. TRANSMISSION LOG is meaningless for a control request. This is the same information returned in the B-Register after any nonbuffered I/O request.

EXTENDED STATUS
: is up to 32 bits of device-dependent information as available from a driver or device. See the *RTE-A Driver Reference Manual* for specific details.

Extended Status Example

Extended Status is always device/driver dependent and reflects the device/driver status as a result of an operation. For example, if an EXEC 1 read request to a terminal is followed immediately by a call to RMPAR, as in:

```
CALL EXEC(1, ...)
CALL RMPAR(XSTAT)
```

Then the returned status might be:

`XSTAT(1)` = 0 (no error)

`XSTAT(2)` = TRANSMISSION LOG equals 40 meaning that 40 words or 80 characters were transmitted to the computer.

`XSTAT(3)` = Reflects the asynchronous serial interface card status. This status is valid at the completion of any control request, unsuccessful read or write request, and timeout or abort. The format of this word can be found in the *RTE-A Driver Reference Manual.*

`XSTAT(4)` = Contains a copy of the last ASIC control word transmitted to the ASIC card.

`XSTAT(5)` = Not used.

4

Class I/O

The Class I/O feature of the operating system is implemented by a special set of EXEC I/O and system library calls. Class I/O calls provide programs with I/O and communication capabilities that are not available with the Standard I/O EXEC calls. The features provided by Class I/O are:

I/O without wait	Allows a program to continue executing concurrently with its own input operation (Class Read) or output operation to any device (Class Write).
Mailbox I/O	Allows cooperating programs to communicate by controlled access to a 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 buffer rethreading	Allows you to move class buffers from one completed class queue to another from within your program without using additional SAM or memory-allocation or word-moving overhead.

Class I/O uses a buffer with an associated access key, called a class number.

Class I/O uses SAM and not system or local common when performing standard program-to-program communication.

Class I/O is double-call I/O because one call is necessary to initiate the operation and another is necessary to complete it. The initiation call (Class Read, Write, Write/Read, or Control) places request parameters, plus data if required, in the class buffer in SAM. The completion call (Class Get) retrieves the data, if data exists, and optionally releases the request.

The class number must be used as a parameter in the Get call, thereby ensuring only authorized programs (programs that know the 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 using system or local common, a command string, or an EXEC call. 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.

A class number is allocated when a program issues an EXEC 17, 18, 19, or 20, or a CLRQ request and requests a class number by setting the CLASS parameter to zero. The class number remains allocated until it is implicitly deallocated or until the program that allocated it terminates. The

class number should always be deallocated when it is no longer needed, freeing it for use by other programs. The maximum number of class numbers, 1 through 255, is established at system generation (refer to the *RTE-A System Generation and Installation Manual*, part number 92077-90034). Programs can allocate more than one class number.

A buffer in SAM is allocated each time a Class I/O operation is initiated. The buffer contains the request and optional data. When the operation is completed (using the Get call), the buffer is released or retained according to the setting of the save buffer bit in the CLASS parameter.

When a Class I/O request is made (such as Read or Write), it is associated with the specified class number and queued on the I/O device. This is 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 Get call. Note that this technique (pending and complete) allows more than one buffer to be associated with the same class number. In other words, a program can make multiple requests specifying the same class number, or a program can have more than one class number allocated to itself.

The following terms have special meanings when used to describe Class I/O:

Class Number:	The access key used to access class buffers on this class number.
Class Users:	Programs that use the class number.
Class Request:	An access to a logical unit number or mailbox I/O with a class number.
Class Members:	Logical unit numbers that are currently being accessed on behalf of a class number. Completion of access removes the association between class number and logical unit number. Completion of access is when the driver completes the request.
Pending Class Requests:	The set of incomplete Class I/O buffers referencing the class number and queued on I/O devices.
Class Queue (Completed):	The set of all completed class buffers on the class number. The structure is first in, first out.
Rethread Buffer:	The buffer that is being moved from one completed class queue to become completed on another class number.

For I/O without wait operations, data can be read from or written to an I/O device by transferring the data to the buffer in SAM or by locking the user data area into memory for the duration of the I/O operation. The calling program can continue execution of other tasks without waiting for the I/O transfer to complete, suspend until the data transfer is complete, or terminate itself (releasing system services to other waiting programs).

The program recovers the results of its Class I/O call by later issuing a Class I/O Get call. If the results are not present, the calling program can either wait or return to execute more code before re-issuing the Class Get call.

A simple example of I/O without wait would be a program that issues a Class I/O Read call in its code, followed by a series of other coded operations. While these following operations were being executed, the system simultaneously could be reading the data into the allocated keyed buffer. The calling program would issue a Class I/O Get call to determine if the I/O was complete and to fetch the data from the buffer, if necessary.

Class I/O Operation

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

1. When the class user issues a Class I/O call (and the call is received), the system allocates a buffer from System Available Memory (SAM) and puts the call information in the header (first 16 words) of the buffer. The call is placed in the I/O queue on the device and the system returns control to the class user.

2. If this is the only call pending on the DVT, the driver is called immediately; otherwise, the system returns control to the class user and links the request on the DVT according to program priority.

3. If buffer space is not available, the class user is memory suspended unless the no-wait bit (bit 15) of the CLASS parameter is set. If the no-wait bit is set, control returns to the class user with the A-Register containing −2 (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.

4. If a program requests an amount of memory that is greater than the amount of System Available Memory (SAM), the program is aborted with an IO04 error return, unless the no-abort bit is set.

5. If a Class Number is not available or the I/O device is down, the class user is placed in the appropriate wait list until the condition changes.

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

If the request is buffered and is either a Write or Write/Read call, the buffer area furnished by the system is filled with data from the calling program. The buffer is then linked (pending) on the DVT initiation list specified by the logical unit number.

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

1. Release the data buffer portion of the request if a buffered Write call and the save buffer bit in the CLASS parameter is not set. The header is retained for the Get call.

2. Queue the class buffer in the completed class queue.

3. If a Get call is pending on the class number, reschedule the calling program. (This means that if the calling program issues a Class Get call or examines the completed class queue before the driver completes, you have effectively beat the system to the completed class queue.) Note that the program that issued the Class I/O call and the program that issued the Class Get call do not have to be the same program.

When you issue the Get call, the completed class queue is checked and only one of the following paths is taken:

1. If the completed class queue is non-empty, the data (if any) is returned. The calling program has the option of leaving the class buffer in the completed Class Queue so the header and any data in the buffer are not lost. In this case, a subsequent Get call with the same class number

obtains the same data. Optionally, the calling program can de-queue and release the class buffer and release the class number back to the system.

2. If the completed class queue is empty (for example, a Get is issued before the Class I/O operation is completed), the calling program is suspended in the Class I/O suspend list (status = CL) 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, any program waiting 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 Get call on the same class number before the first one has been satisfied, the second program is aborted (I/O error IO10). The programs involved can avoid being aborted by setting the no-abort bit (bit 15) in the *ecode* parameter of the Get call. from which it read the data.

Buffered and Nonbuffered Class I/O

It is possible to force a Class I/O request to be nonbuffered by setting the nonbuffered bit (bit 14) in the control word parameter of the request; however, to use all features of Class I/O, including program-program communication, the requests must be buffered. Also, a program that uses buffered I/O is swappable while the I/O is taking place. If nonbuffered I/O is used, the program's data space will be locked in memory to guarantee that the buffer will not be swapped out while the I/O is taking place. Nonbuffered Class I/O, however, uses less SAM and is faster because your program's data does not have to be copied to or from SAM.

You should be aware of the following differences between buffered and nonbuffered Class I/O:

When a buffered Class Read request is made, the buffer that is supplied in the call (*bufr*) is not used. Instead, the data obtained from the read is placed in the buffer that is specified by the Class Get call. When a nonbuffered Class Read is made, the buffer that is specified in the read request contains the data.

When a nonbuffered request is made, data is not returned by the Class Get associated with the request because the data came from or went directly to the buffer specified on the original request. This is true even if the SB (save class buffer) bit in the class parameter is set because there is no data in SAM to save. The Get is only used to determine when the I/O operation is complete, the status, and the transmission length.

Because the I/O is taking place directly to or from the user buffer when nonbuffered Class I/O is being done, the buffer should not be accessed until the I/O has completed.

Class I/O Programming Examples

The following four programs illustrate class I/O calls between two pairs of Pascal and FORTRAN programs. The first program schedules the second program in each of these program pairs with a unique class number so that the second program can access the buffer in SAM. RTE-A handles class I/O in Pascal and FORTRAN similarly. Pascal programs, however, require external EXEC calls; FORTRAN programs do not. The program pair that follows is written in Pascal.

Program 1

```
PROGRAM EXMP1(input,output);
TYPE
    int = -32768..32767;
    btype = array [1..10] of char;
    ntype = packed array [1..6] of char;
    string = packed array [1..12] of char;
VAR
    ibufr : btype;
    name : ntype;
    class, buflen, icode : int;
    i : integer;
    runstr : string;
PROCEDURE EXEC_20 $ALIAS 'EXEC'$
      (ICODE,ICNWD : INT; IBUFR : btype; BUFLN,IOP1,IOP2,class : int);
      EXTERNAL;
{Exec 20 call - class write/read}
PROCEDURE EXEC_9 $ALIAS 'EXEC'$
      (ICODE : int; NAME : ntype; CLASS,dum1,dum2,dum3,dum4 : int;
      runst : string; bufln : int); EXTERNAL;
{ Exec 9 call - schedule a program without wait }
BEGIN
    runstr := 'ru,exmp2,1,1';
    name := 'EXMP2';

    FOR I := 1 to 10 DO
      ibufr[i] := '1';  {Initialize the buffer}

    class := 0;{Set class to zero, so the system}
              {can allocate a unique class number}
    buflen := 10;
    icode := 20;
    {PLACE THE DATA OF 'IBUFR' INTO A BUFFER IN SAM,}
    {          ALONG WITH ITS CLASS NUMBER          }
    exec_20(icode,0,ibufr,buflen,0,0,class);
    {SCHEDULE 'EXMP2' AND SEND IT THE CLASS NUMBER}
    icode := 9;
    exec_9(icode,name,class,0,0,0,0,runstr,-12);
END.
```

Program 2

```
PROGRAM EXMP2(input,output);
TYPE
    int = -32768..32767;
    btype = array [1..10] of char;
    ptype = packed array [1..5] of int;
VAR
    ibufr : btype;
    class, buflen, icode : int;
    pram : ptype;
    i : integer;

PROCEDURE EXEC_21 $ALIAS 'EXEC'$
      (ICODE, CLASS : int; IBUFR : btype; BUFLN : int); EXTERNAL;
{Exec 21 call - class get}

PROCEDURE PARAMS $ALIAS 'RMPAR'$
      (pram : ptype); EXTERNAL;
{Pick up the parameters sent from the 'father' program}

BEGIN
    params(pram); {Pram[1] contains the class number}

    buflen := 10;
    icode := 21;
    class := pram[1];

    {GET THE DATA FROM SAM AND PUT IT INTO 'IBUFR'}

    exec_21(icode,class,ibufr,buflen);

    {PRINT THE DATA AND CHECK FOR CORRECTNESS}

    write( 'The buffer read-in is  ');
    FOR I := 1 to 10 DO
      write(ibufr[i]);

END.
```

The program pair that follows is written in FORTRAN.

Program 3

```
FTN7X,L
       PROGRAM CLIO1
       IMPLICIT INTEGER(A-Z)
       DIMENSION IBUFR(10),NAME(3)
       DATA NAME/'CLIO2'/
C
C   INITIALIZE THE BUFFER
C
       DO 10,I=1,10
              IBUFR(I) = 1
10     CONTINUE
C
C   SINCE WE SET CLASS TO ZERO, THEN THE SYSTEM WILL
C   ALLOCATE A UNIQUE CLASS NUMBER, IF ONE IS AVAILABLE
C
       CLASS = 0
       BUFLEN = 10
C
C   CLASS READ/WRITE CALL PLACES THE DATA IN 'IBUFR'
C   INTO A BUFFER IN SAM ALONG WITH ITS CLASS NUMBER
C
       CALL EXEC(20,0,IBUFR,BUFLEN,0,0,CLASS)
C
C   SCHEDULE 'CLIO2' WITHOUT WAIT, PASSING THE CLASS NUMBER
C
       ICODE = 9
       CALL EXEC(ICODE,NAME,CLASS)
C
       END
```

Program 4

```
FTN7X,L
       PROGRAM CLIO2
       IMPLICIT INTEGER(A-Z)
       DIMENSION IBUFR(10),PRAM(5)
C
C   PICK UP THE PARAMETERS PASSED FROM 'CLIO1'
C   PRAM(1) CONTAINS THE CLASS NUMBER
C
       CALL RMPAR(PRAM)
       BUFLEN = 10
C
C   GET THE DATA FROM SAM AND PLACE IT IN 'IBUFR'
C
       CALL EXEC(21,PRAM(1),IBUFR,BUFLEN)
C
C   WRITE OUT THE BUFFER TO SEE IF WE PICKED UP THE CORRECT
C   DATA
C
       WRITE(1,*)(IBUFR(I),I=1,10)
C
       END
```

CLRQ (Class Management Request)

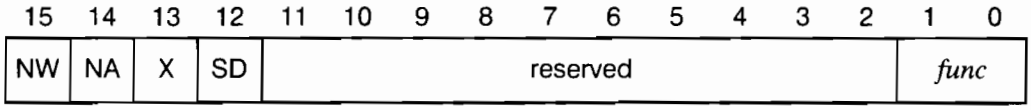
The CLRQ subroutine allows the assignment of ownership to a class number. If the calling program terminates or aborts without cleaning up the class numbers and class buffers assigned to it, the system deallocates these resources.

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(func,class[,pram1])
```

where:

func is the class management control function word; its values are 1, 2, or 3 with optional bits set:



NW is the no-wait bit.

NA is the no-abort bit.

SD is the save data bit.

class is the class number.

pram1 is a call-dependent parameter used to describe a program name or LU.

Class Management Parameters

Two bits in the *func* parameter can be set to allow more control over the process and to obtain error information from the registers.

Bit 15 is the no-wait bit. If set, the program is not suspended if a class number is not available when the CLRQ request is made. The A-Register is set to −1 if a class number is not available, or set to 0 (zero) if the request completed without error.

Bit 14 is the no-abort bit. It operates similarly to the no-abort bit in the *ecode* parameter of the EXEC calls. If this bit is set, the program continues if an error is made in the calling sequence of CLRQ, or some other programming error occurs. If set, the A- and B-Registers contain an ASCII error message. The A-Register contains the first two ASCII characters and the B-Register contains the second two ASCII characters of the four-character message.

Bit 12 is the save data bit. It provides for the graceful handling of data read from a MUX when a CLRQ (3) is executed to deallocate (flush) a class request on an LU. If bit 12 is set in the CLRQ (3) request (*func* = 010003B) and the request is the current request on the LU, RTE-A tests whether DMA is in progress for the request. If it is, the CLRQ (3) request is ignored because valid type-ahead data would be lost if the request's class buffer is flushed. When the current request is completed normally, all other pending requests are deallocated (flushed). By setting bit 12, you can prevent valid type-ahead data from being flushed. Setting bit 12 allows true Write-thru-Pending capability with no data loss on the MUX.

func values and their meanings are:

func = 1 means class ownership is assigned. If *pram1* contains the name of a program, the program is assigned ownership of the class specified in *class*. If *pram1* is zero, no ownership is assigned. If *pram1* is defaulted (omitted from the call), the calling program is assigned ownership. If *class* is zero, a new class number is allocated by the call. The system deallocates the class number and its associated buffers when the program owning the class number becomes dormant.

func = 2 flushes class requests and deallocates the class specified in *class*. All non-active pending requests are deallocated. Abort requests are issued by the system for all active I/O requests, and the buffer is deallocated at the completion of the abort processing. All previously completed requests are immediately deallocated. The class table entry is flagged so that no new requests will be issued on the class. An I/O error (IO00) is 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 deallocates the class. Note that *pram1* is not used.

func = 3 flushes class requests on the LU designated by *pram1*. The system looks at the class table entry specified in *class*. Non-active requests on *class* that are pending on the LU specified in *pram1* are deallocated. If a request is active, an abort request is issued by the system. The buffer is deallocated when the active EXEC request completes. The class number is not deallocated nor are completed class buffers affected.

class 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 *func* and *pram1*. For example, when this parameter is zero and *func* is 1 then a new class number is assigned to the calling program. This new class number is returned in *class*.

pram1 is an optional parameter that works with *func* and *class* in a number of ways. See the above *func* and *class* descriptions for details.

CLRQ Processing

The system checks all terminating and aborting programs for class ownership. If ownership exists, all completed class request buffers are 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 are allowed to complete normally. If I/O is to be aborted, all non-active pending requests are flushed, and the drivers are issued an abort request for all active requests. In the latter case, the buffer and the class are automatically deallocated by the system when abort processing has completed.

Example

The following example allocates two class numbers, assigning the first to the calling program and the second to the program called P2. (P2 must have an ID segment or an SC05 error will result.) The no-abort bit is set in the function parameter to prevent the program from being aborted. An error return routine should always be specified if the no-abort or no-suspend bits are set.

```
$ALIAS CLRQ, NOABORT
   PROGRAM ALLOC
   IMPLICIT INTEGER (A-Z)
   DIMENSION PRAM1 (3)
   DATA PRAM1/'P2'/

   CLAS1 = 0
   FUNC  = 1
C ALLOCATE FIRST CLASS NUMBER TO THE CALLING PROGRAM
   CALL CLRQ (FUNC + 40000B, CLAS1,*100)
   CALL ABREG(ERROR, B)

C CHECK ERROR
   IF (ERROR .NE. 0) GO TO 100
C NOW ALLOCATE THE 2ND CLASS NUMBER, ASSIGNING IT TO P2
   CLAS2 = 0
   CALL CLRQ ( FUNC + 40000B, CLAS2, PRAM1,*100)
   CALL ABREG(ERROR, B)

C CHECK ERROR
   IF (ERROR .NE. 0) GO TO 100
   .
   .
   .
100 CONTINUE
C ERROR PROCESSING HERE
   .
   .
   .
   END
```

EXEC 17, 18, 20 (Class Read, Write, Write/Read)

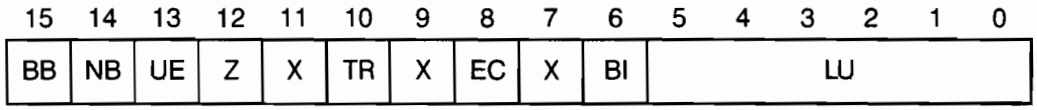
EXEC 17, 18, and 20 transfer information to or from an external I/O device or another program. Depending upon parameter specifications, the calling program is not suspended while the call completes.

```
CALL EXEC (ecode , cntwd , bufr , bufln , pram3 , pram4 , class [ , uv [ , keynum ] ] )
```

where:

ecode is 17 for Class Read, 18 for Class Write, and 20 for Class Write/Read.

cntwd is the control word. For all three calls, the format is as follows:



BB	is the device driver bypass bit
NB	forces nonbuffered operation
UE	allows your program to handle device errors
Z	indicates *pram3* and *pram4* are buffer and buffer length respectively
TR	indicates transparency mode
EC	allows echo of input
BI	indicates ASCII or binary data transfers
LU	selects the device by LU number from range 0-63. An LU of zero indicates a program-to-program data transfer. For LUs greater than 63 use XLUEX call.
X	is defined for the appropriate driver

bufr is the user-defined integer array used as a read/write buffer.

bufln is the length of *bufr*. A positive value indicates the number of words; a negative value is the number of characters.

pram3, *pram4* are two parameters retrieved by the Class I/O Get call, which can be acted upon by the device driver.

class is the class number, which is initially set to zero to inform the system that the program wants a class number issued.

uv is a user-defined variable maintained by the system and returned in the Class Get call. If the request is a class rethread, this variable represents the *oclas* parameter.

keynum is the key number for a locked LU.

Read, Write, Write/Read Parameters

ecode is described in the ECODE section of Chapter 1.

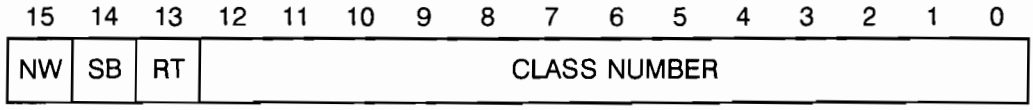
cntwd is the control word, and is exactly the same as *cntwd* in EXEC 1 and 2 (standard read/write). The EXEC 17, 18, and 20 calls can only reference LU numbers up to 63. The XLUEX call described in Chapter 3 should be used in new applications because it can reference LU numbers up to 255, inclusive.

bufr specifies the user buffer where data will be transferred for a Class Write or Write/Read request. For a buffered Class Read, this parameter is not used, but must be supplied in the call sequence. See the example at the end of this section.

bufln specifies the length of the transfer, which must not exceed the size of *bufr*. If positive, *bufln* indicates the number of words in *bufr*; if negative, *bufln* specifies the number of characters in *bufr*.

pram3, *pram4* are parameters whose format depends upon the state of the Z-bit in *cntwd*. If Z = 0, *pram3* and *pram4* are integers; if Z = 1, *pram3* is the name of a buffer, and *pram4* is its length. When the class request is made to an actual device (non-zero LU), these parameters may provide special additional information about the nature of the I/O operation to the driver. As in standard I/O requests, the details defined here are described in the *RTE-A Driver Reference Manual*. When these are not used by a driver or when program-to-program communication is being used (LU=0), these parameters can be used by the calling program to pass information between a Class I/O and a Class Get request.

class is the class word parameter used by a program or programs to coordinate the various Class I/O and Get operations. A class number is assigned by the system to the program, generally on the first Class I/O or CLRQ request issued. This is controlled by the calling program according to the content of the *class* parameter:



NW is the no-wait bit. If the no-wait bit is set and SAM or a class number is unavailable, the calling program is not suspended. Information returned in the A-Register indicates the action taken by the system. The A-Register should be checked for the status of the call if the no-wait bit is set.

A-Register value	Meaning
0	Request successful (no error)
−1	No class number currently available
−2	No memory now, buffer limit exceeded, or pending count is already 255

SB is the save class buffer bit. When set, the system saves the data buffers allocated by a Class Write request for future processing by a Class Get. For Read and Write/Read calls, this bit has no significance.

RT is the rethread class bit which allows Class I/O buffer rethreading on the class buffer at the head of the class completion queue. When RT is set to 1, the call becomes a rethread request rather than the standard call specified by *ecode*. This bit is not examined unless the *uv* parameter is supplied. Normal class users do not require this ability, so normal requests have RT equal to zero. Refer to the Class Rethread Request section for details on the use of this option.

CLASS NUMBER
returns a class number assigned by the system. If a class number is to be allocated, this parameter must be set to zero in the EXEC call. The returned class number is used in later class calls.

uv is an optional parameter which, in general, provides for user-defined information passing between a Class I/O request and a Class Get request. The system only maintains this information and does not act upon it. An exception exists, however, when a class rethread request occurs (bit RT=1 in *class*). In this case, *uv* is used to define one of the pair of class queues. See the section on Class Rethread Requests.

keynum is an optional parameter that contains a key number assigned by the system to a locked LU when a request is made via the LURQ call. This value is returned from the LURQ request and can be supplied in the above I/O requests to allow access to a locked device. Thus a locked LU can be shared among cooperating programs. Note that the program that originally issued the LURQ request is not required to supply this parameter when making an I/O request to the locked device. Refer to the LURQ section in Chapter 2.

A- and B-Register Returns

When a program issues a Class I/O call, the system allocates a buffer from System Available Memory and puts the call in this buffer. The call is queued and the system returns control to the program. If memory is not available, three possible conditions exist:

1. The program is requesting more memory space than will ever be available. In this case, the program is aborted with an IO04 error unless the no-abort bit was set in the *ecode* parameter. If the no-abort option was used, the A-Register contains "IO" and the B-Register contains an ASCII "04".

2. The program is requesting a reasonable amount of memory but the system must wait until memory is returned before it can satisfy the calling program. 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 in the *class* parameter is set, the program is not suspended, and the A-Register is set to −2.

If the pending count is already 255, the program will be suspended until this condition clears. If the no-wait bit in the *class* parameter is set, the program is not suspended and the A-Register is set to −2.

When a program issues a Class I/O call, the system uses LU 0 (the "bit bucket"). If LU 0 is not in the session LU access table, the program is aborted with an SC03 error, unless the no-abort bit is

set in the *ecode* parameter. If the no-abort option is used, the A-Register contains "SC" and the B-Register contains ASCII "03".

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.

The returned content of the B-Register is meaningless. Error information is returned to the A- and B-Registers for unsuccessful calls.

Class Write

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

1. Your program places data in *bufr*, specifies data length in *bufln*, and issues an EXEC 18 call specifying a previously allocated class number in *class* (if *class* = 0, a class number is allocated, if available).
2. The system allocates a buffer in SAM (if available) large enough for the header information. If the request is buffered (NB = 0), additional space is allocated to contain *bufr*, and *bufr* is copied into SAM. If the request is nonbuffered, the program's data area is locked into memory. The calling program continues executing or suspends itself with a Class Get call to the class number.
3. The request is linked (according to program priority) on the DVT associated with the LU number specified in *cntwd*.
4. When the driver completes the control operation, the data portion of the class buffer, if any, is released to the system (SAM) unless the save-buffer bit in the *class* parameter is set. The buffer is linked into the completed class queue. Any program suspended by a previous Get call (EXEC 21) is rescheduled. Refer to EXEC 21 Class Get section for details associated with the Get call.

Class Read

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

1. Your program issues an EXEC 17 call specifying a previously allocated class number in *class* (if *class* = 0, a class number will 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 *bufln*.
2. The system allocates a buffer in SAM (if available) large enough for the header information. If the request is buffered (NB = 0), additional space is allocated to contain *bufr* and then *bufr* is copied into SAM. If the request is nonbuffered, the program's data area is locked into memory. The calling program continues executing or suspends itself with an EXEC 21 Class Get call to the class number.
3. The request is then queued (according to the calling program's priority) on the DVT associated with the LU specified in *cntwd*.
4. When the driver completes the transfer of data from the external I/O device to the buffer in SAM, the buffer is linked into the completed class queue. A program suspended by a previous Get call (EXEC 21) will be rescheduled. Refer to EXEC 21 Class Get for details concerning the 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 earlier. A Write/Read call functions like a Class Write except that the driver receives a Read request code and the buffer is not released on I/O completion, but instead is saved with its header in the completed class queue.

The buffered Class Write/Read call with LU equal to 0 is used for program-to-program communication. The data is transferred from the program's buffer into the class buffer and written to LU 0 (the bit bucket). The data is retained in the completed class queue to be recovered by an EXEC 21 (Get) call from another program. Note that a nonbuffered Class Write/Read with LU equal to 0 is not practical because only the header is in the buffer in SAM. The program's data cannot be recovered.

Figure 4-1 is a sample of communication between two programs. The sequence of events shown in the figure are described below:

1. Program PROGA issues a Class I/O call with the class number parameter set to zero and the logical unit number portion of the control word parameter set to zero. This causes the system to allocate a class number (if available) and the request to immediately be placed on the class completion queue. (Logical unit zero implies immediate completion.)

2. When the Write/Read call completes, PROGA's data will have been placed in the class buffer.

3. PROGA then schedules PROGB (the program receiving the data) and passes PROGB the class number as a parameter.

4. When PROGB executes, it picks up the class number by calling the system library routine RMPAR. Then using this class number, it issues a Class I/O Get call to the class. PROGA's data is then passed from the class buffer to PROGB's buffer.

Note that if PROGA terminates before PROGB issues the Get call, the request will have already been flushed. This means that PROGB's class number is no longer valid.

Figure 4-2 is an example of Class I/O to a terminal.

```
      PROGRAM PROGA
      INTEGER BUFR(32),NAME(3),CLASS
      DATA NAME/'PROGB'/
        .
        .
C     DO CLASS WRITE/READ TO LU ZERO, PROGRAM TO PROGRAM
C     COMMUNICATION
C
      CLASS=0
      CALL EXEC(20,0,BUFR,-64,0,0,CLASS)
C
C     SCHEDULE PROGB AND PASS CLASS NUMBER.
C
      CALL EXEC(9,NAME,CLASS)
        :
      END



      PROGRAM PROGB
      INTEGER BUFR(32),PRAM(5)

C     GET THE CLASS NUMBER BY CALLING SYSTEM SUBROUTINE
C     RMPAR AND SAVE THE CLASS NUMBER IN PRAM(1)
C
      CALL RMPAR(PRAM)
C
C     ACCEPT DATA FROM PROGA USING CLASS GET CALL
C     AND RELEASE CLASS NUMBER AND CLASS BUFFER.
C
      CALL EXEC(21,PRAM(1),BUFR,32)
        :
      END
```

Figure 4-1. Program-to-Program Communication

```
FTN7X,L
      IMPLICIT NONE

*     THIS EXAMPLE PROGRAM FOR CLASS I/O TO A TERMINAL WILL:
*        1) ISSUE A CLASS READ FROM THE USERS TERMINAL
*        2) LOOP ON A CLASS GET
*           A) IF THE GET FAILS, INCREMENT A COUNTER
*           B) IF THE GET COMPLETES, WRITE OUT THE COUNTER AND QUIT

*     THIS DEMONSTRATES THAT THE PROGRAM IS CARRYING OUT OTHER TASKS
*     WHILE THE READ IS PENDING. WHEN THE READ COMPLETES, THE PROGRAM
*     DETECTS IT AND FINISHES.

      INTEGER BLEN, NOWAIT, ECHO
      PARAMETER (BLEN=32, NOWAIT=100000B, ECHO=400B)

      INTEGER INBUFF(BLEN), CLASS, AREG, BREG
      INTEGER*4 COUNT

      DATA COUNT/1/, CLASS/0/

*     ISSUE A CLASS READ TO THE USERS TERMINAL. SYSTEM WILL RETURN
*     THE CLASS NUMBER IN CLASS.

      CALL EXEC(17, 1+ECHO, INBUFF, BLEN, 0, 0, CLASS)

      DO WHILE(.TRUE.)

*     ISSUE A CLASS GET WITHOUT WAIT. CHECK A-REG FOR COMPLETION.

          CALL EXEC(21, CLASS+NOWAIT, INBUFF, BLEN)
          CALL ABREG(AREG, BREG)

*     WHEN THE READ COMPLETES, THE GET WILL RETURN 0 IN BIT 15 AND
*     WRITE OUT THE COUNT. OTHERWISE, INCREMENT COUNT, CONTINUE LOOPING.

          IF (AREG .GE. 0) THEN
               WRITE(1,*)"COUNT = ', COUNT
              STOP
          ENDIF

          COUNT = COUNT + 1
      ENDDO
      END
```

Figure 4-2. Class I/O to a Terminal

EXEC 21 (Class I/O Get)

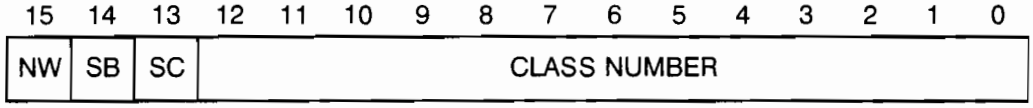
EXEC 21 Class Get completes the data transfer (between the system and a program) that was previously initiated by a Class Read, Write, Write/Read or Control request.

```
CALL EXEC(ecode,class,bufr,bufln[,rtn1[,rtn2[,rtn3[,uv[,to]]]]])
```

where:

ecode is 21 for a Class I/O Get.

class is the class number. Its format is:



NW is the no-wait bit
SB is the save class buffer bit
SC is the save class number bit
CLASS NUMBER is the class number returned from CLRQ with *class*=0, or EXEC 17, 18, 19, or 20

bufr is the user-defined integer array where information will be transferred.

bufln is the length of *bufr*. If positive, length is number of words; if negative, length is number of characters.

rtn1 corresponds to *pram3* from a read or write call, and *pram1* from a control call.

rtn2 corresponds to *pram4* from a read or write call. *rtn2* corresponds to DVT word 17, after driver modification, for a control call.

rtn3 is the request code passed to the driver on an initial read, write, or control call. See below.

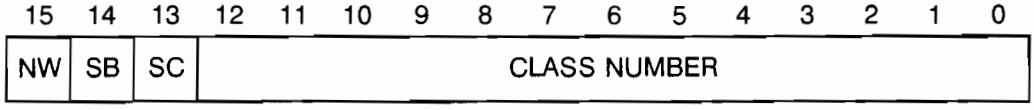
uv is the user-defined variable returned from a previous read, write, or control call.

to is the class timeout or wait time.

Class Get Call Parameters

ecode is described in the ECODE section of Chapter 1.

class coordinates various Class I/O and Get operations. A class is assigned by the system to the program, generally on the first Class I/O or CLRQ request issued. This is controlled by the calling program as follows:



NW is the no-wait bit. When set, the calling program is not suspended if the completed class queue is empty. Information describing the action taken by the system is returned in the A- and B-Registers.

If the no-wait bit was set and no requests were completed, the A-Register will have bit 15 set to 1 to indicate that the queue was empty, and will contain the negative of the number of pending requests minus one.

SB is the save class buffer bit. When set, it saves the class buffer at the head of the completed class queue for future processing by a Class Get. As long as the SB bit is set, the program receives the same data on subsequent Gets as it received in the first SB-set Class Get. If the SB option is not set, the buffer and class header will be released back to the system when the Class Get completes. If the original class request was nonbuffered, then setting the SB bit in a Class Get is not very useful. The header is saved, but the data is not present for the Class Gets.

SC is the save class number bit. When used in a Class Get call, setting SC to 1 will save the class number for use in future class requests. If SC is 0, the class number will be deallocated when the Get call completes if there are no pending or completed buffers on the Class. If SB is set to 1, the class number will not be deallocated regardless of the state of SC.

CLASS NUMBER
is the access key number that must be used in a Class Get to access the class buffer.

bufr is the buffer to receive data. There are several buffer considerations when using the Class Get call:

If the original request was nonbuffered, then data is not returned by the Class Get. Instead, the data has already been placed in the buffer that was specified on the original request.

If the original request was buffered, then the number of words returned to *bufr* is the lesser of:

- The number requested (*bufln* specified in the Get call).
- The number in the completed class data buffer being retrieved (*bufln* specified in the original request).

If the original request was made with the Z-bit set in *cntwd*, the returned value of *rtn1* is undefined.

The Z-buffer is returned only if the original request was a Read, Write/Read, or Write request (with the SB bit set in the *class* parameter). Note that *bufln* must allow for the length of the Z-buffer; that is, *bufln* = length of original data + length of Z-buffer.

The remaining words in *bufr*, if any, past the number indicated by the transmission log (B-Register) or *bufln*, whichever is smaller, are undefined. If a Z-buffer is also returned, the words remaining past the end of the Z-buffer are undefined.

bufln defines the length of the data record to be retrieved; allow for the data type. If the data record contains type REAL values, two words per data item are required. If the data contains double-precision data, four words per data item are required. If the amount of data to be retrieved is greater than *bufln*, only the amount defined by *bufln*

is returned. The transmission log, however, will indicate the actual amount of data transmitted. This might be useful if the SB bit is set.

Optional parameters are:

rtn1 is an optional parameter to obtain data passed in the *pram3* optional parameter of a previous Class I/O Read, Write, Write/Read or Control call. Generally this data is the information passed to the driver in the original request.

rtn2 is an optional parameter to obtain data passed in the *pram4* optional parameter of a previous Class I/O Read, Write, or Write/Read call. Generally this data would be information passed to the driver in the original request. *rtn2* is used to obtain DVT word 17, after driver modification, for a previous class control call. Note that there is not a standard for the value a driver puts in DVT word 17 (see the *RTE-A Driver Reference Manual*, part number 92077-90011).

rtn3 contains one of three values if included in the Get call. It can be used to inform the program making the Get call what type of operation was done to obtain the data that the program is now receiving via the Get call. The three values possible are:

1 if call was 17 or 20	(Read or Write/Read)
2 if call was 18	(Write)
3 if call was 19	(Control)

uv is an optional parameter that provides for user-defined information passed between a Class I/O request and a Class Get request. The system maintains this information and does not act upon it. (An exception exists, however, when a class rethread request occurs with RT=1 in *class*. In this case, *uv* defines one of the pair of class queues. Refer to the section on Class Rethread Requests for details.)

One example of its use in a Get call is that program A reads data from an LU and needs to pass this LU to program B. Program A places in *uv* the destination LU number of the Write/Read call; program B retrieves the LU number from *uv*.

to is an optional parameter that provides a timeout of the Class Get. If no data is available after *to* x 10ms, the Class Get times out. The timeout is indicated in the A-Register on return. *to* may be any value from 1 to 65535.

A- and B-Register Returns

The A- and B-Registers after the return from a successful Get call contain the following.

If a return is made with class completion data, then:

If A=−32768 (100000B), then the Get has timed out.

Bit 15, 14 of the A-Register = 0

A-Register = device status word 6 of the Device Table (DVT); refer to the *RTE-A Driver Reference Manual*, part number 92077-90011.

B-Register = transmission log indicating the positive number of words or characters transmitted to the system buffer during the Class Read, Write or Write/Read. That is, the B-Register contains the number of words transferred in the Class Read, Write, or Write/Read, but not the Class Get, unless *bufln* is large enough to hold all the data transmitted.

If a return is made without class completion data, then:

Bit 15, 14 of the A-Register = 1,0

A-Register = the ones complement (−n−1) of the number of requests made to the class but not yet serviced by the driver (pending class requests).

B-Register = meaningless

Class I/O Get Call Comments

One of the features of the Get call is that one or more programs waiting for system resources can suspend themselves without CPU overhead or program overhead such as polling. A program can perform a Get on a class number associated with a device or another program and put itself to sleep. The program will only be awakened when there is something to process. The desired data will be resident in the program's buffer. After the data is processed, the program can put itself to sleep again with another Get.

When the calling program issues a Class Get call, the program is telling the system that it is ready to accept returned data from a Class Read call or remove a completed Class Write or Control call from the completed class list. If the driver has not yet completed (the Get call got to the completed class before the system did), the calling program is suspended (status = CL) and a marker stating this 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 in the *class* parameter to avoid suspension.

Note that if a Get call is made before the Class Read, Write, or Control call is made, the Get deallocates the class number and immediately returns. If your application requires this type of call to suspend and wait until a request is made and completes, you must set the Save Class bit.

EXEC 19 (Class I/O Device Control)

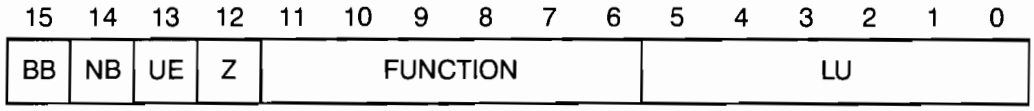
An EXEC 19 call carries out various I/O control operations such as backspace, write end-of-file, and rewind. The calling program does not wait for the operation to be completed.

```
CALL EXEC(ecode,cntwd,pram1,class[,pram2[,pram3[,pram4[,uv[,keynum]]]]])
```

where:

ecode is 19 for Class I/O device control.

cntwd is the control word. Its format is:



BB bypasses the device driver
NB forces nonbuffered operation
UE allows your program to handle device errors
Z specifies optional parameters *pram3* and *pram4* as a buffer and buffer length respectively
FUNCTION
is the set of device functions
LU selects device by logical unit

pram1 is a user-defined parameter.

class is the class number.

pram2 is an optional parameter for certain control functions.

pram3, *pram4* are optional parameters to be used as an optional buffer and buffer length if the Z-bit is set.

uv is an optional user-defined parameter that can be retrieved in a future Class Get.

keynum is an optional parameter representing the locked LU's key number.

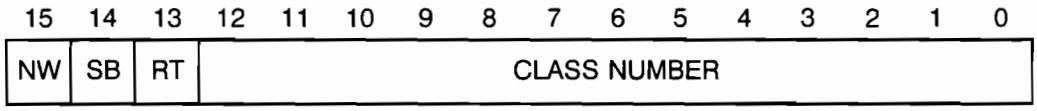
Class I/O Control Parameters

Note that with the exception of the *class*, *pram1* and *pram2* parameters, this call format is identical to the standard I/O Control Call. Refer to Class I/O Get for additional information.

ecode is described in the ECODE section of Chapter 1.

cntwd for this call is essentially the same as in the standard I/O EXEC 3 call. The EXEC 19 call can reference LU numbers up to 63. Use XLUEX to access LU numbers up to 255. The Z-bit and optional parameters *pram1* through *pram4* are driver-dependent. If needed by the driver, *pram1* and *pram2* are sent. If the Z-bit is set, parameters *pram3* and *pram4* specify a buffer and buffer length (the Z-buffer) to be passed to the driver as required; refer to the *RTE-A Driver Reference Manual*. The exception in the control word is the FUNCTION field (bits 6-11). Specific details on available functions are found in the pertinent driver section of the *RTE-A Driver Reference Manual*.

class is the class word parameter used by a program or programs to coordinate various Class I/O and Get operations. A class number is assigned by the system to the program, generally on the first Class I/O or CLRQ request issued. This is controlled by the calling program as follows:



NW is the no-wait bit. If set, the calling program does not become suspended if SAM or a class number is unavailable. Information returned in the A-Register indicates the action taken by the system. The A-Register should be checked for the status of the call:

A-Register value	Meaning
0	Request successful (no error)
−1	No class number currently available
−2	No memory now, buffer limit exceeded, or pending count is already 255

SB is the save class buffer bit.

RT is the rethread class bit which allows Class I/O buffer rethreading on the class buffer at the head of the class completion queue. When RT is 1, the call becomes a Rethread Request rather than the standard call specified by *ecode*. This bit is not examined unless the *uv* parameter is supplied. Although the SB and RT bits can be set on class control requests, the bits are ignored.

CLASS NUMBER
is set to 0 to obtain a class number. The system then allocates a class number (if one is available) to the calling program. The number is returned in this same parameter when the request completes and is used thereafter, bits 12-0 unaltered, for later class calls.

Optional parameters are:

pram1 - *pram4* are driver-dependent parameters. Their use can vary considerably. For example, by specifying a control request with a FUNCTION code of 23, *pram1* indicates enabling or disabling of an asynchronous interrupt response. Refer to the *RTE-A Driver Reference Manual* for specific details on the contents of each optional parameter.

uv is an optional parameter which, in general, provides user-defined information passed between a Class I/O request and a Class Get request. The system maintains this information and does not act upon it. For example, program A reads data from an LU and must pass the LU to program B. Program A sets the value of *uv* equal to the LU number in the Write/Read call that it made. Program B obtains the LU number in the *uv* parameter. Note that this parameter takes on a different meaning if RT=1 in the *class* parameter. Refer to the Rethread section.

keynum is a key number assigned by the system to a locked LU when a locking request is made via the LURQ call. This value is returned from LURQ and can be supplied in the above I/O requests to allow access to a locked device. Thus a locked LU can be shared among cooperating programs. Refer to the LURQ section in Chapter 2 for more detail. The program that originates the LURQ request need not supply this parameter when making an I/O request to the locked device.

A- and B-Register Returns

When a program issues a Class I/O call, the system allocates a buffer from System Available Memory and puts the call in this buffer. The call is queued and the system returns control to the program. If memory is not available, three possible conditions exist:

1. The program is requesting more memory space than will ever be available. In this case, the program is aborted with a IO04 error, unless the no-abort bit was set in the *ecode* parameter. If the no-abort option was used, the A-Register contains "IO" and the B-Register contains an ASCII "04".

2. The program is requesting a reasonable amount of memory but the system must wait until memory is returned before it can satisfy the calling program. 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 in the *class* parameter is set, the program is not suspended and the A-Register is set to −2.

If the pending count is already 255, the program will be suspended until this condition clears. If the no-wait bit in the *class* parameter is set, the program is not suspended and the A-Register is set to −2.

When a program issues a Class I/O call, the system uses LU 0 (the "bit bucket"). If LU 0 is not in the session LU access table, the program is aborted with an SC03 error, unless the no-abort bit is set in the *ecode* parameter. If the no-abort option is used, the A-Register contains "SC" and the B-Register contains ASCII "03".

The A-Register contains −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 contains zero if the request was successful.

If the no-wait bit is clear then the A-Register will contain the class number.

The returned content of the B-Register is meaningless. Error information is returned to the A- and B-Registers for unsuccessful calls (refer to EXEC Error Processing in Chapter 1).

Class I/O Rethread Request

As previously indicated, any of the Class I/O requests (except Class Get) can be used for the special class rethread operation. Under this circumstance a prior class I/O request that is completed (in the completed class queue) can be removed from one class queue and added to another. This can save considerable overhead where request re-transmission or the broadcasting of a request is desirable within a program. The rethread request modifies the normal Class I/O call sequence as described in the Class I/O Rethread Parameters section.

Class Rethread Uses

Class buffer rethreading is a convenient way to handle class buffers without the cost of excessive system overhead for allocating memory and moving words. Some possible uses include the following:

- Re-using buffers passed via program-to-program communication (EXEC 20).
- A convenient method to "cycle through" the buffers in their respective completed class queue(s). Essentially, the program rethreads the buffers to its own queue (*oclas* = *class*); rethreading places the rethread buffer specified via *oclas* at the end of the completed class queue specified in *class*.
- "Broadcasting" class-buffered messages to multiple LUs without having to allocate SAM and move words on each request.

```
CALL  EXEC (ecode , cntwd , bufr , bufln , pram3 , pram4 , class , oclas [ , keynum ] )
```

where:

ecode is 17, 18, or 20 for a class rethread request.

cntwd is the control word. Its format is described in the following section. Its Z-bit condition must be the same in the original and rethread request.

bufr is the user buffer; part or all of the class buffer may be overwritten with new data that is placed in this buffer.

bufln is the above buffer's length. *bufln* must not exceed the rethread buffer length. To leave the buffer in the completed class queue unaltered, set *bufln* to zero.

pram3, *pram4* are parameters that replace those in the previously-defined buffer header if the Z-bit is zero. If the Z-bit equals 1, these parameters can be used to overwrite part or all of the Z-buffer. To prevent the Z-buffer in the completed class queue from being overwritten, set *pram4* to zero and the Z-bit to 1.

class is the class number. Its RT bit (bit 13) is set, indicating rethreading is desired.

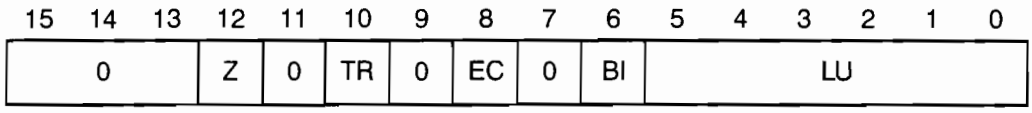
oclas is the old class word identifying the completed class queue where the rethread buffer will be removed.

keynum is the key number of the locked LU; it is used to access a locked LU.

Class I/O Rethread Parameters

ecode is described in the ECODE section of Chapter 1. For the rethread request, *ecode* should generally match the type of the request that was INITIALLY used to obtain the rethread buffer.

cntwd is exactly the same as in the Class Read, Write, Write/Read calls.



If the Z-bit is 0, parameters *pram3* and *pram4* replace those that were defined previously in the rethread buffer header. If the Z-bit is 1, these parameters specify a buffer and buffer length respectively and can be used to overwrite all or part of the Z-buffer that was originally allocated in the initial call.

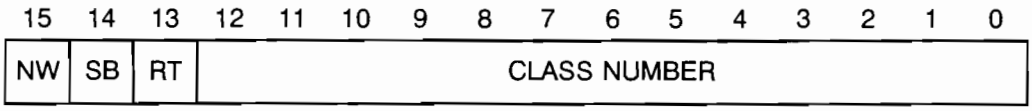
If the Z-bit setting does not match the setting in the original request, an IO04 error results.

bufr contains data to overwrite the rethread buffer. The rethread data buffer is unaltered if the *bufr* length parameter (*bufln*) is zero; if it is zero, *bufr* is ignored. If *bufln* does not equal zero, *bufln* words or characters will be overwritten from *bufr* to the rethread buffer.

bufln determines whether the rethread buffer will be overwritten. If *bufln* is zero, the rethread buffer is moved unaltered. A non-zero *bufln* specifies how much of *bufr* will overwrite the rethread buffer: a positive amount specifies the number of words, and a negative amount indicates the number of characters.

If the Z-bit is set and *pram4* is zero, the driver buffer specified in *pram3* will be transferred unaltered. If *pram4* does not equal zero, then *pram4* words or characters will be overwritten from *pram3* to the Z-buffer part of the rethread buffer.

class is the class word parameter used by a program or programs to coordinate various Class I/O and Get operations. A class number is assigned by the system to the program, generally on the first Class I/O or CLRQ request issued. When the RT bit is set to 1, it indicates rethreading is desired. The class number (bits 0-12) within this parameter identifies the class queue on which the rethread buffer will be attached. This information, in conjunction with the information in *oclas*, enables rethreading to be accomplished. Note that if the class number is zero, a new class number is allocated by the system.



NW is the no-wait bit. If set, the calling program does not become suspended if SAM or a class number is unavailable. The A-Register should be checked for the status of the call:

A-Register value	Meaning
0	Request successful (no error)
−1	No class number currently available

SB is the save class buffer bit. When set, the write buffers issued from a Class request are saved for future processing by a Class Get. Refer to the Class I/O Get Call section.

RT is the rethread class bit which allows Class I/O buffer rethreading on any Class I/O request. It MUST be set to one to indicate a rethread operation.

To obtain a class number from the system, the CLASS NUMBER field (bits 0-12) is set to 0. The system then allocates a class number (if one is available) to the calling program. The number is returned in this same parameter when the request completes and is used thereafter, bits 0-12 unaltered, for later class calls.

This allows the moving of a previously issued class request buffer from one class queue to another. This very sophisticated request requires that special attention be given to the overall program(s) activity with class requests.

It also modifies the normal Class I/O request call sequence requiring strict adherence to special parameter details. Normal class users would not require this ability and would operate with RT=0.

Note The NW, SB, and RT bits are modified by the system after a class rethread request, and, therefore, must be set up before each call is made.

---

oclas is an old class word identifying the completed class queue from which the class buffer will be removed. Note that if a *uv* parameter was supplied in a previous request in the rethread buffer, it will also be maintained on the new class queue (*class*), and can be recovered later using a Class Get. Only the class number field of *oclas* is meaningful.

If the no-abort bit is not set and the completed class queue is empty, then the program will be aborted with an IO04 error.

keynum is a key number assigned by the system to a locked LU when a request is made via the LURQ call. This value is returned from the LURQ request and can be supplied in the above I/O requests to allow access to a locked device. Thus a locked LU can be shared among cooperating programs.

The program that originally issued the LURQ request is not required to supply the *keynum* parameter when making an I/O request to the locked device. Refer to the LURQ section in Chapter 2 for more details.

Class Rethread Procedures

The general flow of the Class rethread request is best shown in terms of a hypothetical example:

Programs A, B and C all have buffers in the completed class queue. Assume program A knows the class numbers programs B and C used to access their respective class buffers. Program A now wants to rethread B's and C's buffers to its completed class queue on its class number. Program A needs to do the following:

1. Set the RT bit in program A's class word to indicate rethreading is desired.
2. Set *oclas* to program B's class number. Bits 15, 14, and 13 are not checked.
3. Set *bufln* to zero to rethread the buffer exactly as it currently stands. To insert new information into program B's buffer, *bufr* should contain the new information and *bufln* should contain *bufr*'s length. *bufr*'s length must not exceed the length of program B's buffer which is now the rethread buffer. In other words, program A cannot overwrite more information in program B's buffer than was originally allocated to program B.
4. Issue the rethread call; program B's buffer (the rethread buffer) will be added to the end of program A's class number completed class queue.
5. Repeat steps 2 through 4, putting program C's Class Number in *oclas* and taking the appropriate action at step 4. Repeat until all desired buffers are rethreaded.

Class Rethread Example

The following three programs illustrate class rethreading.

```
      PROGRAM THRED(3,89)
      IMPLICIT INTEGER (A-Z)
      INTEGER BUFA (5), BUFB (5), BUFC (5), NUMB (15),
      MYNAM(3)

      DATA CLASS/0/, CLASA/0/, CLASB/0/, CLASC/0/
      DATA NUMB/' '/

      DATA BUFA/'MESSAGE 1'/
      DATA BUFB/'MESSAGE 2'/
      DATA BUFC/'MESSAGE 3'/

C ASSIGN CLASS NUMBERS TO PROGRAM GETEM

      CALL CLRQ (1,CLASS,6HGETEM )
      CALL CLRQ (1,CLASA,6HGETEM )
      CALL CLRQ (1,CLASB,6HGETEM )
      CALL CLRQ (1,CLASC,6HGETEM )

C PICK UP MY OWN NAME AND DISPLAY IT

      CALL PNAME (MYNAM)
      CALL EXEC (2,1,MYNAM,3)

      CALL CNUMO (CLASS,NUMB)
      CALL CNUMO (CLASA,NUMB(5))
      CALL CNUMO (CLASB,NUMB(9))
      CALL CNUMO (CLASC,NUMB(13))

C DISPLAY ALL FOUR CLASS NUMBERS

      CALL EXEC (2,1,NUMB,15)

C PASS THREE BUFFERS WITH "TAGGED" MESSAGES.  TAG IS
C FIRST PARAMETER (1,2 OR 3) AND INDICATES THE ULTIMATE
C DESTINATION (PROGA, PROGB OR PROGC).

      CALL EXEC (20,0,BUFA,5,1,0,CLASS)
      CALL EXEC (20,0,BUFB,5,2,0,CLASS)
      CALL EXEC (20,0,BUFC,5,3,0,CLASS)

C NOW SCHEDULE THE FOUR PROGRAMS.  PROGA, PROGB AND PROGC
C NEED KNOW ONLY THEIR OWN CLASS NUMBERS. GETEM KNOWS ALL
C THE NUMBERS. GETEM WILL RETHREAD THE BUFFERS IT GETS
C ON NUMBER CLASS TO CLASA, CLASB OR CLASC (ACCORDING TO
C THE FIRST PARAMETER).

      CALL EXEC (10,6HGETEM ,CLASS,CLASA,CLASB,CLASC)
      CALL EXEC (10,6HPROGA ,CLASA)
      CALL EXEC (10,6HPROGB ,CLASB)
      CALL EXEC (10,6HPROGC ,CLASC)
      END

      PROGRAM GETEM(3,88),Get Class Buffers and
      Reroute 'Em
      IMPLICIT INTEGER (A-Z)
```

```
      INTEGER CLAS(5), BUFR (10)
      EQUIVALENCE       (CLASS,CLAS(1)),
      >    (CLASA,CLAS(2)),
      >    (CLASB,CLAS(3)),
      >    (CLASC,CLAS(4))

C SC = SAVE CLASS BIT, WHEN USED WITH CLASS GET.
C SB = SAVE BUFFER.  IF NOT SAVED (I.E. DEALLOCATED FROM SAM
C     ON CLASS GET), THEN IT CAN'T BE RETHREADED.

      DATA SC/20000B/, SB/40000B/, RT/20000B/

      CALL RMPAR (CLAS)
10    CALL EXEC(21,SC+SB+CLASS,BUFR,10,R1,R2)
      CALL EXEC(20,0,BUFR,0,R1,R2,CLAS(R1+1)+RT,CLASS)
      GO TO 10

C IN THIS EXAMPLE, GETEM AND THE THREE PROGRAMS PROGA,
C PROGB AND PROGC WILL WAIT "FOREVER" FOR MORE MESSAGES
C TO ARRIVE ON CLASS.  NOTE: IF GETEM IS ABORTED, THEN
C PROGA, PROGB AND PROGC WILL BE ABORTED ALSO, SINCE
C GETEM OWNS ALL THE CLASS NUMBERS.

      END

      PROGRAM PROGA(3,88),Display Buffers Passed by Monitor
      IMPLICIT INTEGER (A-Z)
      INTEGER CLASS(5), BUFR (10)
      DATA BUFR/3*2H  ,2H: ,6*2H  /
      DATA SC/20000B/

C IN THIS EXAMPLE, PROGA, PROGB AND PROGC ARE IDENTICAL.
C ONCE PROGA IS A TYPE 6 FILE, DO THE FOLLOWING:
C
C     :RP,PROGA
C     :RP,PROGA,PROGB
C     :RP,PROGA,PROGC
C
      CALL RMPAR (CLASS)
      CALL PNAME (BUFR)
10    CALL EXEC (21,CLASS+SC,BUFR(5),10,R1,R2)
      CALL ABREG (A,LEN)
      CALL EXEC (2,1,BUFR,LEN+4)
      GO TO 10
      END
```

5

Program Control

Program Control Program control functions provided within the system consist of program loading and scheduling. These multiprogramming activities are controlled by you but supervised by the system. These requests offer a number of features:

- Program overlay loading.
- Program scheduling, with or without wait.
- Queued scheduling with or without wait.

These control functions provide programmatic loading and scheduling of real-time programs with fast response time.

The option of scheduling with or without wait allows programs to react to real-time events synchronously or asynchronously. Simply by selection of EXEC parameters, the scheduling program waits for the scheduled program to complete, or proceeds with its own processing while the system takes care of executing the scheduled program.

If the calling program requests a scheduling with wait, it does not continue executing until the scheduled program completes. If the calling program requests a scheduling without wait, it continues execution immediately, regardless of the action the system takes with respect to the request. If the scheduled program is available, it executes in conformance with the request. If the scheduled program is not available, the system issues an error message.

EXEC 8 (Overlay Load)

Note The EXEC 8 Overlay Load call is for use only with non-CDS programs. It does not apply to CDS programs, so an EXEC 8 call from such a program causes an SC06 error. Control returns to the instruction following the call only if the no-abort bit is set and an error that causes an abort occurs. See *ecode* in Chapter 1 for details on setting the no-abort bit.

An EXEC 8 request loads a program overlay (background or real-time) of the calling program from disk into memory, and transfers control to the overlay's entry point. The overlays occupy an area in memory provided by the program.

```
CALL EXEC(ecode,NAME[,pram1[,pram2[,pram3[,pram4[,pram5]]]]])
```

where:

ecode is 8 for an overlay load request.

NAME is a three-word integer array in which the overlay name must appear in all uppercase and as:

NAME(1) = 1st two characters
NAME(2) = 2nd two characters
NAME(3) = last character in upper 8 bits (the lower byte is not significant)

pram1 - *pram5* are optional user-defined parameters.

Completion of the request passes control over to the loaded program overlay.

Overlay Load Parameters

ecode is described in the ECODE section of Chapter 1.

NAME is the five-character name (in uppercase) of the overlay. If the overlay name has less than five characters, the name must be padded with blanks to fill out the remaining letters.

pram1 - *pram5* are optional user-defined parameters of type INTEGER, and are used to pass parameters to an overlay. Their use can vary considerably. Such diverse things as array addresses, LU numbers, or program names can be passed using one or more of these parameters. Their advantage is that Common (Program or System) is not used, nor are any special declarations necessary. To retrieve these parameters, a call in the overlay to the system library subroutine RMPAR is necessary. See the RMPAR section in Chapter 7 for details on calling RMPAR.

A- and B-Register Returns

On overlay entry, the registers are set as follows:

A = Address of the (overlay's) short ID segment

B = Address of the parameter list

If an overlay load attempt is made on a overlay that does not exist or does not have an ID segment, an SC05 error results and is contained in the A- and B-Registers as follows:

A = The two ASCII characters "SC" (51503 octal)

B = The two ASCII characters "05" (30065 octal)

If an overlay load attempt is made from a CDS program (on a system with the VC+ Enhancement Package), an SC06 error results. The A- and B-Registers will contain:

A = The two ASCII characters "SC" (51503 octal)

B = The two ASCII characters "06" (30066 octal)

SEGLD (Overlay Load)

SEGLD loads an overlay of the calling program from disk 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 that allows use of SEGRT and Symbolic Debug. It should not be used in CDS programs, or an error will occur.

```
CALL SEGLD(NAME,ierr[,pram1[,pram2[,pram3[,pram4[,pram5]]]]])
```

where:

NAME is 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 is an error return; its value is −6 if the overlay cannot be loaded or if the calling program is a CDS program.

pram1 - *pram5* are optional user-defined parameters of type INTEGER, and are used to pass parameters to an overlay. Their use can vary considerably. Such diverse things as array addresses, LU numbers, or program names can be passed using one or more of these parameters. Their advantage is that Common (Program or System) is not used, nor are any special declarations necessary. To retrieve these parameters, a call in the overlay to the system library subroutine RMPAR is necessary. See the RMPAR section in Chapter 7 for details on calling RMPAR.

SEGLD loads overlays via an executive request. Control returns to the calling routine if the segment cannot be loaded (*ierr* will be equal to −6) or if the overlay calls SEGRT (*ierr* will equal zero).

SEGRT (Return to Main from Overlay)

SEGRT allows an overlay that was called from the main program via a SEGLD request to return to the instruction after the SEGLD request.

Note SEGRT is provided for use in conjunction with SEGLD. Only non-CDS programs using SEGLD style segmentation should use these calls.

```
CALL SEGRT()
```

There are restrictions on the use of SEGRT:

1. SEGRT can only be used if the overlay was loaded by a SEGLD request.
2. The overlay must have been loaded from the main, not another overlay.
3. SEGRT can only return to the main, not another overlay.

CHNGPR (Change Program Priority)

CHNGPR sets the priority of the calling program to the value specified. For a description of priorities see the description of the PR command in the *RTE-A User's Manual*, part number 92077-90002.

ierr = CHNGPR(*prio*)

where:

ierr is an error return. If the value of PRIO is not between 1 and 32767, inclusive, *ierr* will be less than zero; otherwise *ierr* will be set to zero.

prio is the priority to be assigned to the calling program. It should be an integer variable with a value between 1 and 32767, inclusive.

EXEC 6 (Stop Program Execution)

EXEC 6 terminates the calling program or another program subordinate to the calling program.

```
CALL EXEC(ecode[,prog[,type[,pram1[,pram2[,pram3[,pram4[,pram5]]]]]]])
```

where:

ecode is 6 to stop program execution.

prog is zero or a three-word integer array. To terminate the calling program, set *prog* to zero. To terminate a subordinate (child) program, use the name of a three-word character array containing the name of the program to be terminated. The entire program name must be specified in uppercase.

type is an optional parameter indicating the type of termination. Possible values are:

```
-1 = terminate serially reusable
 0 = normal completion (default)
 1 = make the program dormant, save resources.
 2 = normal completion and remove program from time list.
 3 = same as 2, and also remove program's ID segment from system.
```

Note that types 2 and 3 are considered abnormal terminations by the parent program if the child was scheduled with wait. See the following discussion of EXEC 9, 10, 23, and 24.

pram1 - *pram5* are optional integer parameters that may be specified if the calling program is terminating itself, that is, *prog* is zero. These values become the default parameters to the program when it is next scheduled, if it is scheduled without parameters.

Stop Program Execution Parameters

ecode is described in the ECODE section of Chapter 1.

type specifies the type of termination to use. Zero is the normal (default) termination. Results of this option are:

1. The next time the program executes, it will start from its primary entry point (not the location following the EXEC call).
2. If the EXEC 6 call was made from a re-entrant subroutine, the subroutine is unlocked.
3. If a string was passed to the program and the program never requested the string with EXEC 14 call, the string's memory space is returned to SAM.
4. Any devices the program locked with the system library routine LURQ are unlocked (see the LURQ section in Chapter 2 for more information).
5. Any resource numbers that are locally locked to the program are unlocked (see the RNRQ section in Chapter 2).

6. Any resource numbers that are locally allocated to the program are deallocated (see the RNRQ section in Chapter 2).

7. If the program owns an I/O class number or numbers, its class requests are flushed and its class buffers and class number(s) are deallocated. From a user program, this is done with a CLRQ request (see the CLRQ section in Chapter 4).

8. Any programs that are queue suspended (QU) or wait suspended (WT) for this program are resumed. See the section on EXEC 9, 10, 23, and 24.

9. The memory occupied by the program is released and made available for any other use.

10. If the program was originally loaded from a memory-image (type 6) program file, it will be reloaded from the same file when it is scheduled again.

11. The ID segment and its program space are deallocated if the program was flagged as temporary by the RU or XQ command or by an IDCLR call. See the *RTE-A User's Manual* for details of the RU and XQ commands. See the IDCLR section in the *RTE-A/RTE-6/VM Relocatable Libraries Reference Manual*, part number 92077-90037, for more information on IDCLR.

A termination with *type* = 1 terminates saving resources and is the same as an EXEC 7 (pause) request, with the following exceptions:

1. The program is put in the dormant (OF) state, and the SV bit is set. This allows it to be scheduled by an XQ, RU, or AT command, an EXEC 9, 10, 12, 23, or 24 request, a driver, or an interrupt. Note that the SV bit is cleared when the program is rescheduled.

   FmpRunProgram calls can also be used to schedule the program. However, the :IH option should be used with the program's name (for example, PROG:IH) so that the program is not cloned.

2. If the program terminates itself, then any programs queue suspended (QU) or wait suspended (WT) for this program are resumed. See EXEC 9, 10, 23, and 24 for more information on queue scheduling and wait scheduling. If the program schedules a child program and then terminates that child program saving resources, then programs queue suspended or wait suspended on the child are not resumed.

3. No other program can be loaded into the memory area until the dormant program has been swapped out. When the dormant program is next scheduled, it will not be loaded into memory from the program file. Instead, it will be loaded from the swap area if not still in memory. It will continue execution at the next instruction after the EXEC 6 call with *type* equal to 1.

A termination with *type* = −1 terminates serially reusable. When rescheduled, the program is not loaded from disk if it is still in memory. Serially reusable completion should be used only with programs (if stored on disk) that can initialize their own buffers or storage locations. The program must be able to maintain the integrity of its data since its partition (and data) may be overlaid by another program. If the operating system needs the program's memory area for other purposes, the program is not swapped out to disk, but is simply overlaid.

A termination with *type* = 2 is identical to a termination with *type* = 0, except that the program is removed from the time list. A termination with *type* = 3 is identical to a termination with *type* = 2, except that the ID segment of the program is removed from the system. A termination with *type* =

2 or *type* = 3 is considered an abnormal termination by the parent program if the calling program was scheduled with wait. See the discussion of EXEC 9, 10, 23, and 24.

The parameters *pram1* through *pram5*, if given, are placed into the temporary words $TMP1 through $TMP5 of the calling program's ID segment. These values become the default parameters retrieved by a call to the RMPAR routine when the program next executes. This feature is typically used by a program in the time list to pass parameters to itself on its next execution. If the program is scheduled by a means that passes a new set of parameters then these default values are lost. The FmpRunProgram routine and both the CI and the system RU and XQ commands always pass a new set of parameters, even if no parameters are specified in the runstring. If new parameters are specified in an EXEC 9, 10, 23, or 24 call then the default parameters are lost. Note that these default parameters may be defined only when a program terminates itself.

Stop Program Execution Example

The following example illustrates the use of two of the options for EXEC 6.

```
      PROGRAM ENDD
      IMPLICIT INTEGER (A-Z)
      DIMENSION PRAM(5)
      .
      .
      .
C SUSPEND YOURSELF AND PICK UP PARAMETERS ON THE NEXT RUN.

      CALL EXEC (6, 0, 1)

C PROGRAM IS SUSPENDED AT THIS POINT. NEXT EXECUTION WILL
C CONTINUE FROM THIS POINT.

C PICK UP PARAMETERS
      CALL RMPAR( PRAM )
      .
      .
      .
C NOW TERMINATE AND RELEASE RESOURCES.
      CALL EXEC(6)
      END
```

EXEC 7 (Program Suspend)

EXEC 7 suspends execution of the calling program until it is restarted by a GO operator request.

```
CALL EXEC(7)
```

When the program is restarted by a GO command the B-Register contains the address of a five-word array set by the GO command. A call to the RMPAR system library subroutine can load these parameters, providing the RMPAR call occurs immediately following the EXEC 7 request. Another alternative is to call the system library routine GETST to retrieve the GO command string. For details on GETST, see Chapter 7.

If no parameters were passed to the program with the GO command, a call to RMPAR returns zero in every parameter.

The FORTRAN library subroutine .PAUS, which is automatically called by a FORTRAN PAUSE statement, generates the program suspend EXEC 7 call. In addition, it logs the pause and any supplied number on the scheduling terminal.

EXEC 9, 10, 23, 24 (Program Scheduling)

The EXEC 9, 10, 23, and 24 calls schedule a program for execution and pass up to five parameters and a buffer to the scheduled program.

Program schedule calls are similar to the RU or XQ operator commands. The programs to be scheduled must already have an assigned ID segment in order to execute. If a program lacks an ID segment, the FmpRunProgram call can be used to assign an ID segment and schedule the program.

```
CALL EXEC(ecode,NAME[,pram1[,...[,pram5[,bufr,bufln]]]])
```

where:

ecode is
- 9 for immediate schedule with wait
- 10 for immediate schedule without wait
- 23 for queue schedule with wait
- 24 for queue schedule without wait

NAME is a three-word integer array containing the five-character program name which must appear in all uppercase and as:

- *NAME(1)* = 1st two characters
- *NAME(2)* = 2nd two characters
- *NAME(3)* = last character in upper 8 bits (the lower byte is not significant)

pram1 - *pram5* are five optional integer parameters.

bufr is where data to be sent to or received from the child program is placed. (A program scheduled as a result of an EXEC call is called a child program.)

bufln is the length of *bufr*. A positive number is the number of words; a negative number indicates the number of characters.

Program Scheduling Differences

The program that issues the EXEC 9, 10, 23, or 24 request is known as the parent. The program scheduled as a result of the EXEC call is the child.

Assume that program PARNT schedules program CHILD:

With an EXEC 9, PARNT waits until CHILD is finished executing before continuing its own execution. PARNT is wait suspended (WT) until CHILD has finished. This is known as scheduling with wait.

With an EXEC 10, PARNT schedules CHILD and then immediately continues its own execution. This is known as scheduling without wait.

With an EXEC 23, if CHILD is already executing, PARNT waits in a queue until CHILD is finished, then schedules CHILD. PARNT waits again until CHILD has finished before continuing its own execution. PARNT is queue suspended (QU) until CHILD executes. This is known as queue scheduling with wait.

With EXEC 24, if CHILD is already executing, PARNT waits in a queue until CHILD is finished, then schedules CHILD and immediately continues its own execution. PARNT is queue suspended (QU) until CHILD executes. This is known as queue scheduling without wait.

Program Scheduling Parameters

The *ecode* parameter determines whether or not the calling program (parent) waits, and whether the parent's schedule request will be queued until the requested program (child) becomes dormant.

When a program that has been scheduled with wait (EXEC 9 or 23) completes, the parent program can recover the system's copy of optional parameter *pram1* to determine whether or not the child terminated normally.

Abnormal termination of the child is caused by any of the following conditions:

a. System abort of program.
b. An OF operator command.
c. Child program performed an EXEC (6,0,2) or EXEC (6,0,3) self-termination call. (Refer to "Exec 6 Stop Program Execution.")

Abnormal termination causes the system's copy of optional parameter 1 to be set to −32768 (100000 octal). This occurs even if the child program attempted to pass back parameters via PRTN. The parent can recover the system's copy of optional parameter 1 by calling RMPAR.

If the child program terminated normally and no parameters are passed back via PRTN, the value of all parameters returned by RMPAR are zero. Otherwise, the RMPAR-returned values will be those passed back from the child's PRTN call. The PRTN subroutine allows child programs to pass parameters back to their parent programs; it is described in Chapter 7.

An *ecode* of 9 (schedule with wait) causes the system to put the parent in a wait status. If required, the parent may be swapped by the system in order to run another program. The child runs at its

own priority, which may be greater than, less than, or equal to that of the parent. Only when the child terminates does the system resume execution of the parent at the point immediately following the program schedule call.

All schedule combinations are legal: a background (BG) program can schedule a real-time (RT) program, a RT program can schedule a BG program, a RT program can schedule a RT program, and a BG program can schedule a BG program.

An *ecode* of 10 (schedule without wait) also schedules the child according to its priority. The parent continues at its own priority without waiting for the child to terminate. Note that an *ecode* of 9 or 10 will not schedule a child program if it is busy (not dormant).

An *ecode* of 23 or 24 functions similar to an *ecode* of 9 or 10 except that the system places the parent program in a queue if the child is not dormant. The queue means that if the child is not dormant, the potential parent is suspended until the child may be scheduled by this parent. When the potential child can be scheduled, the request is reissued and execution proceeds as in EXEC 9 or 10.

Setting the no-suspend bit (bit 14) prevents the parent program from being SAM suspended when insufficient SAM is available.

As in all EXEC calls, the no-abort/no-suspend option is available by setting bit 15 or bit 14 respectively of the *ecode* parameter.

A- and B-Register Returns

Returns from EXEC 9 or 10 requests:

> If a child program is dormant, it is scheduled and a zero value is returned to the calling program in the A-Register. If the child is not dormant, it is not scheduled by these calls and the program status (a non-zero value) is returned to the calling program in the A-Register.

Returns from EXEC 23 or 24 requests:

> Zero is returned in the A-Register to indicate that the EXEC 23 or 24 request succeeded.

In each type of scheduling request, B-Register returns have no special meaning.

Optional Parameters

A call to RMPAR as the first executable statement in the child program transfers parameters *pram1* through *pram5* to a specified five-word array within the called program. For example:

```
PROGRAM CHILD
INTEGER PRAM (5)
CALL RMPAR (PRAM)
.
.
```

Note that PRAM is a maximum of five words.

For schedule with wait requests (*ecode* = 9 or 23), the child program may pass back five words to the parent by calling the PRTN library routine. For example:

```
PROGRAM DAUTR
INTEGER PASBAK(5)
.
.
CALL PRTN (PASBAK)
CALL EXEC (6)
END
```

The parent may recover these parameters by calling RMPAR immediately after the child call. If the parent program needs to examine the A- and B-Register values (for example, on a return from an EXEC 9 or 10 request), then immediately after the child call, the parent should issue a call to ABREG followed by a call to RMPAR.

If the optional buffer *bufr* is included in the parent program's scheduling call, the buffer is moved to System Available Memory and assigned to the child. The child can recover the buffer by using the GETST subroutine or the string passage EXEC 14 call.

Caution If the *bufr* string is picked up with a GETST call, make sure the string's structure conforms to that required by GETST. (The string should be preceded by two commas because GETST performs an EXEC 14 and returns everything after the second comma up to the end of the string.)

The parent program is memory-suspended if there is not enough System Available Memory to currently hold the buffer. The parent is aborted and a SC10 error is returned if there will never be enough System Available Memory for the buffer. The parent will not abort if the no-abort bit (bit 15 in *ecode*) is set. The length of the string is limited only by the amount of usable System Available Memory.

Program Scheduling Example

The following program will schedule a number of child programs, some with wait (EXEC 9 or 23), others without wait (EXEC 10 or 24).

```
PROGRAM PATER ()
IMPLICIT INTEGER (A-Z)
DIMENSION BACK(5), PRAM(5), NAME(3), BUFR(40)
DATA NAME/'P1'/

*   All programs to be scheduled have an
*   assigned ID segment already
C
C   Schedule without wait, EXEC 10
C   Set the no-abort bit in case the child program isn't dormant
*
     CALL EXEC (10 + 100000B, NAME, *1000)
50   CALL ABREG (A, B)                  ! A-reg = 0 for success
     IF (A .NE. 0) GOTO 100             ! if fail, queue schedule P1
C
C   Schedule with wait, EXEC 9
C
     NAME = 2HP2                        ! schedule P2 without wait
     CALL EXEC (9 + 100000B, NAME,*1000)
60   CALL ABREG (A, B)                  ! A-reg = 0 for success
     IF (A .NE. 0) GOTO 200
*
*   child P2 passes back 3 parameters and a buffer;
*   the third parameter is the length of the buffer.
*
     CALL RMPAR (BACK)                   ! get 3 parameters
     RD = 1                              ! enable EXEC 14 read
     CALL EXEC (14, RD, BUFR, BACK(3)) ! get buffer
     CALL ABREG (A, B)                   ! B-reg is transmission log
     WRITE (1, '("True buffer length = ", I2)') B

     CALL EXEC (6)                       ! normal termination here
C
C   P1 wasn't dormant; so queue schedule without wait,
C   EXEC 24
C
100  WRITE (1, '("Waiting for ", 3A2)') NAME
     CALL EXEC (24, NAME)                ! try scheduling P1 again
     GOTO 50                             ! until successful
C
C   P2 wasn't dormant; so queue schedule with wait, EXEC 23
C
200  WRITE (1, '("Waiting for ", 3A2)') NAME
     CALL EXEC (23, NAME)                ! try scheduling P2 again
     GOTO 60                             ! until successful
C
C   Error handler for initial scheduling attempts
C
1000 WRITE (1, '("Cannot schedule ", 3A2, /,
     >            "possibly no ID segment." )') NAME
END
```

EXEC 22 (Program Swapping Control)

An EXEC 22 request allows its calling program to lock itself into memory so it cannot be swapped, even by a program of higher priority.

A non-CDS program consists of one partition (referred to as a data partition); so only the options for the data partition are legal. A CDS program (only in the VC+ environment) consists of two partitions, a data partition and a code partition, so all swapping options are legal.

In the following list, options 0 and 1 apply to both RTE-A and the VC+ Enhancement Package. Options 2 and 3 apply only to VC+ Enhancement Package.

```
CALL EXEC(ecode,option)
```

where:

ecode is 22 to alter swapping.

option specifies the alteration:

0 indicates that the data partition can be swapped
1 indicates that the data partition cannot be swapped
2 indicates that the code partition can be overlaid
3 indicates that the code partition cannot be overlaid

A call to EXEC 22 sets the appropriate lock bit in the calling program's ID segment (word 16, bit 6 for the data partition; word 36, bit 14 for the code partition, and the lock bit in the partition's Memory Descriptor Status word). The dispatcher examines this status bit before overlaying a partition. If this bit is set then the partition will not be overlaid, even if a higher priority program would normally be swapped into its partition. If the option specified is not 0, 1, 2 or 3 or if a non-CDS program attempts to lock or unlock a code partition, an SC02 error results.

EXEC 26 (Memory Size Request)

EXEC 26 returns the memory limits of the partition the calling program occupies while executing. For CDS programs running in the VC+ environment, a call to EXEC 26 returns the memory limits of the area in the data partition between the upper stack limit (Z-Register) and the end of the partition (not including EMA or MSEG if used). Refer to the chapter on CDS programming for a description of the data partition. For a non-CDS program, the limits of the area in the program partition between the end of the program (main if not segmented, largest segment if segmented) and the end of the partition are returned (not including EMA or MSEG if used).

```
CALL EXEC(ecode,fword,nwords,lpart[,umap[,cmap]])
```

where:

ecode is 26 for a memory size request.

fword returns the address of the first available word behind the calling program. This is the address of the last word of the program, plus the length of the segment space, plus one.

For CDS programs, *fword* returns the address of the first available word beyond the area which is reserved for the stack in the data partition. This is the value of the current stack size limit (Z-Register) plus a stack overflow buffer zone of 265 words.

nwords returns the number of words available between the last word of the program and the last word of the program address space.

For CDS programs, *nwords* returns the number of words available between the stack bounds and the last word of the data partition, excluding any VMA/EMA area. This number can be changed by the operator DT command or the LINK HE command.

lpart returns the length in pages of the partition occupied by the program, including the user base page. For CDS programs, *lpart* is the length in pages of the partition occupied by the program's data segment, including page 0. This may include the EMA area or the working-set area, if the program uses VMA or non-shareable EMA.

umap for non-CDS programs, an optional 32-word array that returns a copy of the currently enabled user map. For CDS programs, *umap* contains the current user data map.

cmap an optional 32-word array that returns a copy of the current user code map. *cmap* is used only for CDS programs.

Parameter Relationships

The executive (EXEC) calculates *nwords* by subtracting *fword* from the address of the last word of the program's logical address space. The logical address space is determined at when the program is linked. It can be the size of the program, the size determined by the size override option in the LINK or operator SZ command. The partition in which the program resides, in turn, may be equal to or larger than the program's logical address space. In VC+ Enhancement Package, the EXEC calculates *nwords* by subtracting *fword* from the address of the last word of the data segment size. The size is determined at link time either by default or by the LINK HE command. The default is the stack bound value, rounded up to the next page boundary. If the HE command is used, the size of the data segment may be increased by word increments. The operator DT command may be used to change the data partition size by page increments rounded up to the next page boundary.

A- and B-Register Returns

For successful EXEC 26 calls, the A-Register and B-Register contents are unchanged. For unsuccessful calls, the A- and B-Registers return error information, described under Error Processing in Chapter 1.

Memory Size Request Example

The following FORTRAN example calls EXEC with *ecode* 26 to determine the size of the unused portion of its logical address space. The program passes the address of the first word and the size of the available space to a user-written routine (USER) that uses the space for temporary data storage.

```
PROGRAM DYALC
IMPLICIT INTEGER (A-Z)
   .
   .
DIMENSION UMAP(32)
   .
   .
ECODE=26
CALL  EXEC(ECODE,FWORD,NWORDS,LPART,UMAP)
   .
   .
CALL  USER(FWORD,NWORDS)
   .
   .
   .
```

EXEC 29 (Retrieve ID Segment Address)

EXEC 29 scans the ID segments in the system for a specified program.

```
CALL EXEC(ecode,name,session,idaddr[,search_flag])
```

where:

ecode is 29 to retrieve an ID segment address.

name is a three-word integer buffer that contains the name of the program whose ID segment address is to be retrieved. The first unoccupied ID segment may be found by setting each element of *name* to zero.

session is the number of the session with which *name* is associated. Unlike the IDGET routine, this parameter does not default to the caller's session. Setting *session* to 0 specifies the system session. This parameter is ignored if any of the following conditions are found to be true:

- *name* is all zeroes,
- *search_flag* is specified and has the sign bit set,
- *name* specifies a program that is a system process or is in the system session.

idaddr is set by EXEC 29 to the ID segment address of the program that was found. If a matching program is not found, *idaddr* is set to 0.

search_flag if specified, and the sign bit of this parameter is set, it indicates that the *session* parameter does not have to match for an ID segment to match.

6

Time Operation Requests

Time Operation Requests Time operation requests can be divided into two categories:

1. Obtaining the time from the real-time system clock (in two possible formats).
2. Scheduling the calling program or other programs by putting them in the time list. Programs are scheduled to execute once or a number of times with a specified time interval elapsing between each execution. The initial starting time for the program may also vary; it can be offset from the current time or it can be an absolute time such as 1:37:30 AM.

The real-time clock operates with 10-millisecond resolution, which allows that level of precision to be achieved in time operations. Initial time is established through the use of the TM operator command. Refer to the *RTE-A User's Manual*, part number 92077-90002, for details on TM.

A- and B-Register Returns

For EXEC 11 and EXEC 12 calls, the returned contents of the A- and B-Registers are meaningless if the call is successful. Error information is returned to the A- and B-Registers for unsuccessful calls.

EXEC 11 (Time-Retrieval Request)

EXEC 11 retrieves the current time from the real-time clock. The time is returned as a series of integers.

```
CALL EXEC(ecode,time[,year])
```

where:

ecode is 11 for a time-retrieval request.

time is a five-word integer array where the time value is returned.

year is an optional parameter where the year is returned.

ecode is described in the ECODE section of Chapter 1.

The array *time* contains the time on a 24-hour clock, with the day of the year in the last word. Values are returned as integers:

time(1) = Tens of milliseconds
time(2) = Seconds
time(3) = Minutes
time(4) = Hours (0 to 23)
time(5) = Day of the year (such as 117)

The hours are returned in 24-hour format: midnight is 0, 2 AM is 2, and 5 PM is returned as 17. Note that subtracting 12 from a 24-hour time (if it is greater than 12) yields its equivalent 12-hour PM value.

The optional parameter YEAR contains a four-digit year as set by the TM command (such as 1982).

The EXEC 11 call is similar in function to the TM operator command. Refer to the *RTE-A User's Manual* for more details on TM.

EXEC 12 (Initial Offset Scheduling)

An EXEC 12 call schedules a program for execution at specified time intervals, starting after an initial offset delay. The system places the specified program in the time list and then returns to the calling program.

```
CALL EXEC(ecode,NAME,units,often,delay)
```

where:

ecode is 12 to schedule a program after an initial offset.

NAME is a three-word array containing the name of the program (in uppercase) to be put in the time list.

units is the resolution code that specifies the time units; it requires the parameter *often*. The values of *units* mean:

0 = remove from time list
1 = tens of milliseconds
2 = seconds
3 = minutes
4 = hours

If parameter *units* is equal to zero, parameters *often* and *delay* are optional.

often is the execution multiple, specifying the time interval between runs of a program run repeatedly. It is used in conjunction with *units*. Its value may be 0 through 4095.

delay is the initial offset. It is a negative number indicating the starting time of the first execution (not zero). A delay of greater than 24 hours is reduced to modulo 24 hours; for example 36 hours is reduced to 12 hours.

Initial Offset Scheduling Parameters

ecode is described in the ECODE section of Chapter 1.

NAME is a three-word integer array containing the five-character name of the program (in uppercase) to be put in the time list. The program name occupies two ASCII

characters per word with the last byte as a blank (a word-for-word description of *NAME* is in the EXEC 8 section of Chapter 5). The calling program may put itself in the time list by setting *NAME* to zero. Any other program may be put in the time list as long as it has an ID segment already assigned to it.

If *NAME* is zero, the calling program is suspended and then resumed after the delay indicated by *delay* and *units*. While the program is suspended, it is put in the time-suspend list, and cannot be scheduled by an operator command or another program. When the delay has elapsed, execution resumes with the instruction after the EXEC 12 call. In this case, the *often* parameter is ignored.

units specifies the time element involved, or resolution code. *units* in conjunction with the parameter *often* specifies the time between each execution of the program indicated in the parameter *NAME*. For example, if *units* is 3 (indicating minutes) and *often* is 7, then the program specified in *NAME* will run every 7 minutes. If *units* is 0, the program specified in *NAME* is removed from the time list and the remaining EXEC 12 parameters are ignored.

often indicates the time between each execution of *NAME*. It is used in conjunction with *units*. For instance, if *units* is 2 (meaning seconds), an *often* value of 30 causes *NAME* to be executed every 30 seconds. An *often* value of zero causes *NAME* to be executed once.

delay is the initial offset, used in conjunction with *units*, to specify when *NAME* will initially execute. It is always a negative number.

Initial Offset Scheduling Examples

The Timed Execution EXEC 12 call is similar to the AT operator request. Refer to the *RTE-A User's Manual* for details on the AT command. This form of the EXEC 12 call schedules a program in various ways:

1. RUN ONCE — Execute the program in *NAME* once, 45 minutes from the current time, and then remove it from the time list.

   units = 3 (minutes)
   often = 0 (run once)
   delay = −45 (run after 45 minutes have elapsed from the current time)

2. RUN REPEATEDLY — 8 hours from the current time, when the program *NAME* is dormant, it will execute, go dormant, and then re-execute every 24 hours. The program *NAME* will remain in the time list indefinitely.

   units = 4 (hours)
   often = 24 (run every 24 hours)
   delay = −8 (run after 8 hours have elapsed from the current time)

3. REMOVE PROGRAM FROM THE TIME LIST — If *units* is zero, the program in *NAME* (or the calling program if *NAME* is zero) is removed from the time list. The remaining parameters are ignored.

Note that an EXEC 6 termination, *type*=2 or *type*=3, removes the calling program from the time list. If the system time is changed while one or more programs are time scheduled by initial offset, the execution time of the programs are altered to be consistent with the new system time.

EXEC 12 (Scheduling Absolute Start Time)

Another form of the EXEC 12 call schedules a program for execution at specified time intervals, starting at a particular absolute time. The system places the specified program in the time list and returns to the calling program.

The value of *hour* determines whether this call is an initial-offset EXEC 12 or an absolute-start-time EXEC 12. If *hour* (or *delay*) is negative, the call is interpreted as an initial offset call: *hour* or *delay* is used to specify the delay and any remaining parameters such as *min*, *sec*, and *msec* are ignored. If *hour* is positive, the call is interpreted as an absolute start time call: *hour*, *min*, *sec*, and *msec* define the starting time.

```
CALL  EXEC (ecode [ , NAME [ , units [ , often [ , hour [ , min [ , sec [ , msec ] ] ] ] ] ] ] )
```

where:

ecode is 12 to schedule a program after an absolute starting time.

NAME is a three-word array containing the name of the program to be put in the time list.

units is the resolution code specifying time units, and is used with *often*. Its possible values mean:

0 = remove from time list
1 = tens of milliseconds
2 = seconds
3 = minutes
4 = hours

often is the execution multiple, specifying the time interval between runs of a program run repeatedly. It is used in conjunction with *units*. Its value may be 0 through 4095.

The following parameters collectively specify the starting time:

hour is the starting hour, 0 to 23.
min is the starting minute.
sec is the starting second.
msec is the starting tens of milliseconds.

Parameters that are omitted default to zero.

Absolute Start Time Parameters

ecode is described in the ECODE section of Chapter 1.

NAME is a three-word integer array containing the five-character name of the program to be put in the time list. The program name occupies two ASCII characters per word with the last byte as a blank (a word-for-word description of *NAME* is in the EXEC 8 section of Chapter 5).

If *NAME* is zero, the calling program is suspended and then resumed at the time indicated by *hour*, *min*, *sec*, and *msec*. While suspended, the program is put in the time-suspend list and cannot be scheduled by an operator command or another program. At the specified time, execution resumes at the instruction after the EXEC 12 call. If *NAME* is zero, the *often* parameter is ignored.

Any other program may be put in the time list as long as it has an ID segment already assigned to it.

units specifies the time element involved, or resolution code. *units* in conjunction with the parameter *often* specifies the time between each execution of the program indicated in the parameter *NAME*. For example, if *units* is 3 (indicating minutes) and *often* is 7, then the program specified in *NAME* will run every 7 minutes. If *units* is 0, the program specified in *NAME* is removed from the time list and the remaining EXEC 12 parameters are ignored.

often indicates the time between each execution of *NAME*. It is used in conjunction with *units*. For instance, if *units* is 2 (meaning seconds), an *often* value of 30 causes *NAME* to be executed every 30 seconds. An *often* value of zero causes *NAME* to be executed once.

hour, *min*, *sec*, and *msec* together specify the absolute starting time for the program's execution. *hour* is in 24-hour time format; in other words, 9 in the morning is indicated with 9, and 2 in the afternoon is indicated with 14. To convert 12-hour times (such as 6 PM) to 24-hour format (hour 18), add 12 to 12-hour PM times. AM times are the same in either format. If the system time is changed while an absolutely-time-scheduled program is waiting to execute, its execution time is not changed. However, repeated executions of absolutely-time-scheduled programs are treated as offsets from the initial execution and such program execution times are adjusted if the system time is changed. The *hour*, *min*, *sec*, and *msec* parameters must be positive values. When RTE-A checks the range of these parameters, it issues an SCO2 error if they are negative or out of range.

Absolute Start Time Examples

The timed-scheduling EXEC call is similar to the CI command AT (refer to the *RTE-A User's Manual*). This call differs from the initial offset version only in that a specific future starting time is specified instead of a delay. For example, if the current time is 2 PM (1400 hours) and the program should run at 3:45 PM (1545 hours), use:

hour	=	15
min	=	45
sec	=	0
msec	=	0

This call can schedule a program in various ways:

1. RUN ONCE — After the program to be scheduled is dormant, execute it once and then remove it from the time list. *units* may be any valid non-zero integer in this case; its exact value is not meaningful. Use:

units	=	3	(any non-zero)
often	=	0	(run once)
hour	=	0	(midnight)
min	=	0	(absolute starting time
sec	=	0	is 100 milliseconds
msec	=	10	after midnight)

2. RUN REPEATEDLY — After the program to be scheduled is dormant, it will execute, go dormant, and then re-execute every 15 minutes. The program will remain in the time list indefinitely:

units	=	3	(minutes)
often	=	15	(run every 15 minutes)
hour	=	6	(program's
min	=	15	first run is
sec	=	50	50 seconds past
msec	=	0	6:15 AM)

   The program may be scheduled by the operator or another program while it is dormant and in the time list.

3. TIME SUSPEND — If *NAME* is zero, the calling program is immediately time (TM) suspended. The program is suspended until the absolute time specified elapses. Then it resumes execution at the instruction following the EXEC call. While the program is time suspended, that program cannot be scheduled by the operator or another program, and it is not put in the time schedule list. However, if the program is cloneable, then a copy of the same program can be scheduled.

4. REMOVE PROGRAM FROM TIME LIST — If *units* is zero, the program named in *NAME* (or the calling program if *NAME* is 0) is removed from the time list. Any remaining parameters are ignored.

FTIME (Formatted ASCII Time Message)

FTIME returns to the calling program a 15-word (30-character) ASCII message showing the time, day, and date.

```
CALL FTIME(bufr)
```

where:

bufr is a 15-word integer buffer to which FTIME returns the ASCII time string.

FTIME returns a string in the form:

```
1:27 PM MON., 15 FEB., 1982
```

Each word of *bufr* contains two ASCII characters.

All characters, including numbers, are in ASCII format. The day and month fields will contain 4 characters (such as in TUE. or SEPT).

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)
```

where:

hour is 0 to 23	*month* is 1 to 12
minute is 0 to 59	*day* is 1 to 31
second is 0 to 59	*year* is 1978 to 1999

SETTM and 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. Possible return values for *error* are as follows:

0	No error, time is changed
−1	Illegal parameter value

The system time can be changed even if there are time-scheduled programs in the system. Programs that 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 which were scheduled relatively (to run after a certain period of time passes) will still run when the time period has passed, regardless of how the system time changed. For example, if at 10:00 AM a program is scheduled to run in 1 hour, and then the system time is changed to 8:00 AM, the program will run at 9:00 AM, 1 hour after it was scheduled.

7

Parameter Passing and Conversion

The subroutines in this chapter pass and retrieve parameters between programs, or convert one type of data into another. In FORTRAN, routines called as functions should be declared as integers unless otherwise stated.

PRTN and PRTM (Parameter Return)

These two routines pass parameters to the program that scheduled it with wait (refer to EXEC 9 and 23 in Chapter 5 for more information on scheduling with wait). The scheduling program may recover these parameters with RMPAR. The PRTN subroutine passes five parameters; PRTM passes four parameters.

```
CALL PRTN(prams)
```

where:

prams is a five-word integer array. Up to five parameters can be passed in *prams*.

Note The PRTM routine is provided for compatibility with the RTE-6/VM Operating System. In RTE-6/VM, the wait flag is not cleared when you use PRTM. In RTE-A, the wait flag is not cleared for either PRTN or PRTM.

To pass parameters with PRTM, you place the values in array elements 1 through 4; however, these values will be parameters 2 through 5 when the parameters are recovered by RMPAR. The first parameter will be undefined.

RMPAR (Recover Parameters)

RMPAR is a general purpose request used to recover parameters passed to the calling program. These parameters may originate from a number of sources: operator run commands (RU, XQ, ON, GO), program scheduling requests (EXEC 6, 7, 9, 10, 23, 24), or information passed by some drivers after an nonbuffered read, write, or control (EXEC 1, 2, 3) request. For recovery of runstring or scheduling parameters, a call to RMPAR should be the first executable instruction of the program. This is necessary because its parameter storage area is used by each EXEC request.

```
CALL RMPAR(prams)
```

where:

prams is a five-word integer array where up to five parameters will be returned.

A- and B-Registers are meaningless upon return from RMPAR.

If at least one but fewer than 5 parameters are passed from the operator runstring or from the parent scheduling program and a RMPAR request is made, the non-supplied parameter positions return zero. If no parameters are passed, the first element in the array will contain the LU of the scheduling device.

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
```

Parsing of ASCII strings passed by the XQ, RU, ON, and GO commands (or a MESSS call) is according to the rules documented with the PARSE subroutine. Only the first two characters of an ASCII parameter are passed by RMPAR.

EXEC 14 (String Passage Call)

EXEC 14 retrieves the command string that scheduled the program or passes a buffer back to the scheduling program. For command string retrieval, the EXEC 14 call should be made before any EXEC program schedule requests and before any FMP calls. Command strings are discussed in the *RTE-A User's Manual*, part number 92077-90002.

```
CALL EXEC (ecode , rcode , bufr , bufln [ , prams ] )
```

where:

ecode is 14 for string passage.

rcode is the retrieve or write code:

1 = retrieve parameter string or buffer
2 = write buffer to "parent"
3 = write buffer to calling program.

bufr is the integer buffer where the string is placed.

bufln is the length of the above buffer, specified as either positive number of words or the negative number of characters. Recommended length is 128 words (256 characters).

prams is an optional five-integer array, used when writing a buffer to yourself, placed in the program's temporary area for later retrieval by a RMPAR call.

String Passage Parameters

ecode is described in Chapter 1.

rcode is the Retrieve or Write code. When the first option is used (*rcode* = 1), this parameter indicates that the command string passed by an RU, XQ, ON, or GO operator command, or the *bufr* passed by an EXEC 9, 10, 23, or 24 schedule call is to be retrieved by the calling program. When the second option is used (*rcode* = 2), this parameter indicates that the string information contained in the parameters *bufr* and *bufln* is to be passed to the "parent" program. The parent is the program that scheduled the calling program. The third option (*rcode* = 3) is identical to the second option, except the string is written to the calling program. Refer to the Procedures section below for more details.

bufr is the user-defined buffer where the string is either retrieved or placed. The action is determined by *rcode* above.

bufln contains the length of *bufr*, expressed in positive words or negative characters. If *rcode* = 1 and the string is longer than *bufln*, only *bufln* words or characters are transmitted. If an odd number of characters are requested in a retrieve operation, the right half of the last word is undefined.

A- and B-Register Returns

Upon return from a retrieve operation, the A-Register contains status information: 0 if the operation was successful or 1 if no string was found. The B-Register is a positive number giving the number of words (or characters) transmitted. If the string is longer than *bufr*, only *bufln* words or characters are transmitted.

String Passage Procedures

The command string retrieved is exactly like the string used in scheduling the program via RU, ON, GO commands, or EXEC 9, 10, 23, or 24. The block of System Available Memory (SAM) used to store the command string is released by this call or when the calling program goes dormant. Parsing of the returned string is left to the calling program. The system library routine GETST can be used to recover the parameter string portion of the command string.

If the write parameter string option is used, the call returns any block of system available memory associated with the program and allocates a new block for the program into which the string will be stored. If the string is being written to the parent, the parent has to have scheduled the child with wait option.

If no memory is currently available, the calling program is memory suspended. If there will never be enough memory and bit 15 of *ecode* is not set, the calling program is aborted with an SC10 error.

If there is no parent when *rcode*=2, execution continues at the return point with the A-Register equal to 1. If the write parameter operation was successful, the A-Register is set to 0.

Example:

```
RU,PROGX,ABCDSTRING
```

where `RU,PROGX,ABCDSTRING` is returned to `BUFR` by "`CALL EXEC(14,1,BUFR,10)`"; and `ABCDSTRING` is returned by "`GETST`".

GETST (Recover Parameter String)

GETST 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. For parameter string recovery, GETST must be called before any EXEC program schedule requests and before any FMP calls. Refer to the *RTE-A User's Manual* for a discussion of command strings.

```
CALL GETST(bufr,bufln,tlog)
```

where:

bufr is the user-defined buffer large enough to hold the parameter string.

bufln is the length of *bufr*, expressed as a positive number of words or negative number of characters. Recommended length is the maximum of 40 words or 80 characters.

tlog returns a positive integer giving the number of words (or characters) actually transmitted into *bufr*. *tlog* never returns an integer greater than the value of *bufln*.

The A- and B-Registers are undefined 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 A string 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.

Example:

This example and corresponding output shows the differences between EXEC 14 and GETST, as well as demonstrating the use of GETST.

```
      PROGRAM EXAM1
      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

      PROGRAM EXAM1
      IMPLICIT INTEGER (A-Z)
      DIMENSION BUFR2 (40)

C     GET THE PARAMETER STRING WITH GETST

      CALL GETST (BUFR2,40,TLOG)

      WRITE (1,2) (BUFR2(I),I=1,TLOG)
2     FORMAT("Parameter part is: ",40A2)
      END
```

Output:

```
The runstring is: RU,EXAM1,P1,P2,P3,P4,P5,P6
Parameter part is: P1,P2,P3,P4,P5,P6
```

PARSE (Parse Input Buffer)

PARSE allows a program to parse an ASCII string.

```
CALL PARSE(bufr, crcnt, rbufr)
```

where:

bufr is an integer user-defined buffer where the string to be parsed is placed.

crcnt is an integer variable that specifies the number of characters in *bufr*.

rbufr is the integer receiving buffer. The result of the parse of the string in *bufr* is stored in *rbufr*. *rbufr* is always 33 words long.

The result of parsing the ASCII string in *bufr* is stored in *rbufr*. A set of 4 words in *rbufr* is used to describe each parameter that is parsed. The set is:

Word	Entry	
1	FLAG WORD	0 = parameter is null 1 = parameter is a numeric 2 = parameter is ASCII
2	VALUE(1)	0 if null; parameter value if numeric; first 2 characters if ASCII.
3	VALUE(2)	0 if null or numeric, else the 3rd and 4th characters.
4	VALUE(3)	0 if null or numeric, else the 5th and 6th characters.

This subroutine can parse up to eight parameters.

The *rbufr* parameter is initialized to 0 before parsing the string *bufr*. Word 33 of *rbufr* will be set to the number of parameters in the string.

One or more non-digit characters in the parameter (except a trailing "B" or leading "−") makes a parameter ASCII. A leading "+" is considered a non-digit ASCII character, and makes the parameter ASCII.

The PARSE routine ignores all blanks and uses commas to delimit parameters. ASCII parameters are padded to six characters with blanks. If more than 6 characters are present, the leftmost 6 are returned. Numbers may be negative (with a leading "−") and/or octal (trailing "B"). If a parameter starts with a number and you want the parameter to be parsed as ASCII, then any letter except "B" should appear as the last character in the string, as follows:

```
123c, LU9, TEST
```

The result is that the first parameter is parsed as 4 ASCII characters, not as one hundred twenty-three. The application program must discard the "dummy" character.

INPRS (Inverse Parse)

INPRS converts a buffer of data back into its original ASCII form.

```
CALL INPRS(rbufr,pnum)
```

where:

rbufr is an integer buffer containing the parsed string.

pnum is the number of parameters parsed. If the parsed buffer *rbufr* (returned from PARSE) is used, then *rbufr*(33) contains the number of parameters parsed.

INPRS is the inverse of PARSE. The calling program passes to INPRS a buffer in the format returned by the PARSE routine. INPRS then reformats the buffer into an ASCII string that is syntactically equivalent (under the rules of PARSE) to a buffer that may have been passed to PARSE to form *rbufr*. The length of the ASCII string in characters will be eight times the number of parameters. For example, if there were 4 parameters in the string, then INPRS would return an ASCII string 32 characters in length.

The following example illustrates the use of PARSE and INPRS:

```
      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(2O10)
60    CONTINUE
C PRINT THE RESULT
C SYNTACTICALLY EQUIVALENT TO ORIGINAL
      CALL EXEC(2,1,PBUF2,PBUF2(33)*(-8))

      END
```

CPUID (Get CPU Identification)

CPUID returns a value indicating the type of CPU that is being used.

```
icpu = CPUID()
```

icpu will be set to one of the following values:

ICPU	CPU Type
2	A600
3	A700
4	A900
5	A600+
7	A400
10	A990

LOGLU (Get LU of Invoking Terminal)

LOGLU returns an LU number suitable for use in EXEC calls to the terminal. In RTE-A, this value is always 1; but in other RTE operating systems it may be different.

```
lu = LOGLU(reallu)
```

On return, *lu* contains a 1. *reallu* contains the real LU number of the scheduling terminal; this may be up to 255.

LUTRU (Returns True System Logical Unit)

The LUTRU subroutine returns the true system LU number associated with a session LU.

```
CALL LUTRU(seslu,syslu)
```

or

```
syslu = LUTRU(seslu)
```

where:

seslu is the session logical unit number to be checked.

syslu is the system LU is returned here.

If *seslu* is not equal to 1 (that is, not in session), *syslu* is set equal to *seslu*.

The Macro/1000 calling sequence is as follows:

```
      EXT LUTRU
       :
      JSB LUTRU
      DEF RTN
      DEF SESLU
      DEF SYSLU
RTN   :
```

EQLU (Interrupting LU Query)

EQLU returns the LU of the interrupting device that scheduled the program.

lu = EQLU(*zlu*)

where:

lu is set to the first LU number assigned to the Device Reference Table (DVT) if the LU was found, or zero if an LU referring to the DVT was not found.

zlu is the same as the LU parameter. It must be supplied when the call is made.

When a driver detects an interrupt from a device and schedules a program as a result of that interrupt, the driver provides the scheduled program an easy method of obtaining the LU of that device. The driver passes the LU as the first scheduling parameter. The system library routine EQLU will provide the LU of the interrupting device.

Note The *RTE-A Driver Reference Manual*, part number 92077-90011, explains how to enable or disable program scheduling on interrupt, as well as additional values that may be passed by certain drivers.

CNUMD, CNUMO, KCVT (Binary to ASCII Conversion)

CNUMD, CNUMO, and KCVT convert a positive integer binary number to ASCII.

```
CALL CNUMD(numb,bufr)      (for decimal)

CALL CNUMO(numb,bufr)      (for octal)

digits = KCVT(numb)
```

where:

numb is an unsigned 16-bit integer number that will be converted to ASCII format in the specified number base. Variables may also be used in *numb*.

bufr is a three-word integer array where the ASCII representation of *numb* will be stored (six characters maximum).

digits is the one-word destination (ASCII) buffer.

CNUMD converts an unsigned 16-bit integer to ASCII decimal (base ten) representation. CNUMO converts an unsigned 16-bit integer to ASCII octal (base eight) representation. KCVT is a function that converts a positive integer number to base ten and returns the last two equivalent ASCII digits:

```
J = 32767
DIGITS = KCVT(J)
```

DIGITS will contain the two ASCII characters "67".

The range of numbers that these routines accept are from 0 to 65535 decimal and 0 to 177777 octal. Leading zeros are converted to spaces.

IFBRK (Breakflag Test)

IFBRK tests the break flag and clears it if it is set. (Also refer to the BR operator command in the *RTE-A User's Manual*, part number 92077-90002.) IFBRK returns zero if flag not set. If flag is set, IFBRK returns −1. Two common uses are:

```
IF (IFBRK() .LT. 0) GOTO 10
  or
ibk = IFBRK()
```

where:

ibk is a one-word variable containing a returned value of 0 (if the break flag is not set) or −1 (if the break flag is set).

In the first format, 10 is the branch taken if the break flag is set. The flag will be cleared.

IFTTY (Interactive LU Test)

IFTTY ascertains whether or not a logical unit (LU) is interactive.

int = IFTTY(*lu*)

where:

lu is the number of the logical unit being tested.

The call returns the following:

int or A-Register = −1 if *lu* is interactive
= 0 if *lu* is NOT interactive

MESSS (Message Processor Interface)

MESSS processes all base set commands that can be entered at the system prompt. MESSS lets your programs access the base set commands; refer to the *RTE-A User's Manual* for a description of which commands are base set commands.

ic = MESSS(*bufr*, *count*[, *lu*])

where:

bufr is an integer buffer that must be at least 72 characters (36 words) in length. The command string is placed here before the call, and the string is overlaid by any returned error or status message. Parameters in the command string must be separated by commas.

ic is the negative character count of the returned message in *bufr* if it is returned here, or 0 if no returned message.

count is the integer value containing the character count for the above *bufr*.

lu is an optional parameter.

The command is placed in *bufr* and the number of characters in *bufr* is placed in *count*. Command formats and possible error and status messages are described in the *RTE-A User's Manual*.

The value of *ic* on return will be zero if there is no response, or the negative character count if there is a message from the system. Any message will be in *bufr*; note this overwrites the previous contents of *bufr*.

The *lu* parameter is optional. It has meaning only for the RU or XQ requests. When *lu* is included in the MESSS call, it is like adding "/*lu*" to the program name when issuing the command interactively.

The passed LU is returned as the LOGLU value for the scheduled program and the program runs in that session; the program gets the session LU for LOGLU in all cases. All EXEC calls to

LU 1 will be directed to this LU. When both /session and *lu* are passed, the /session overrides the LU.

The program is run in the specified session (or your session, if no session is specified) and must be available in the specified session. Otherwise, a message is returned that reads:

```
No ID segment for this program. Try RP,program.
```

If the first parameter in the runstring is defaulted (as in RU,PROGZ,,PRAM2,PRAM3,...), it is replaced by the value *lu*. If RMPAR is used to pick up the parameters, the first parameter will be the value of *lu*.

If the *lu* parameter is not supplied, the terminal LU of the calling program is used.

LOGIT (Send Logging Message)

LOGIT logs a message in the error logging file. The calling sequence is:

ierr = LOGIT(*string*,*string_len*)

where:

string	is an integer buffer up to 128 words long that contains the ASCII string to be logged.
string_len	is the integer number of words in the string.
ierr	is an integer variable that is set to a negative value if the call fails. Note that a positive value does not guarantee that the message was logged.

Note that the message is not logged unless the spool error logging option is enabled. See routine RteErrLogging to determine if error logging is enabled.

RteErrLogging (Is Error Logging On?)

This routine may be used to programmatically determine if error logging is on or off. The calling sequence is:

onoff = RteErrLogging()

logical *onoff*, RteErrLogging

where:

onoff	returns TRUE (negative value) if error logging is enabled, or FALSE (zero) if disabled.

PNAME (Retrieve Program Name)

PNAME returns the program's name into a three-word buffer. The name in the ID segment is normally the name given in the PROGRAM statement or NAM record, but may have been altered by renaming the program file, by using the rename option of the RP command, or through program cloning.

```
CALL  PNAME(bufr)
```

where:

bufr is a three-word integer buffer that returns the program's clone name.

IDGET (Retrieve ID Segment Address)

IDGET retrieves the ID segment address of a specified program.

```
idaddr = IDGET(name[,session_number])
```

where:

idaddr is set by IDGET to the address of the program's ID segment given in the parameter *name*, or to zero if the program does not have an ID segment in memory.

name is a three-word integer buffer with a program name in it. To find the ID segment address of the first unoccupied ID segment (the ID segment with the lowest address), set the *name* parameter to nulls, that is, each array element contains the value zero. In this case, the *session_number* parameter is ignored.

session_number if specified, designates the session number associated with the program. It makes it possible to distinguish between programs with the same name. This optional parameter defaults to the calling program's session number, and is only used in systems with the VC+ System Enhancement Package. Setting *session_number* to 0 specifies the system session. This parameter is ignored if either of the following conditions are found to be true:

- *name* is all zeroes,
- *name* specifies a program that is a system process or is in the system session.

The EXEC 29 call is an alternative method of obtaining the ID segment address of a specified program. It has the additional benefit of being able to scan the system for a given program name regardless of the session number. The EXEC 29 call is documented in Chapter 5.

IDINFO (Return ID Segment Information)

IDINFO returns information about the program whose ID segment address is passed to it. IDINFO also can return the state of the program, equivalent state in RTE-6/VM, and the parent's ID segment address.

ierr = IDINFO(*idaddr*,*pname*,*astat*,*mefstat*,*pidaddr*)

where:

ierr is set to a negative value if an invalid ID segment address is given or if the ID segment currently does not contain a program. If the address is valid, *ierr* is set to zero or a positive value.

idaddr specifies the address of the ID segment.

pname is a three-word integer array that is set to the name of the program. The name is padded with spaces.

astat is the state of the program. This value is returned only if *astat* is zero when IDINFO is called; otherwise, the value in *astat* remains unchanged.

mefstat is the state of the program if the program was running on an RTE-6/VM type system. This value is returned only if *mefstat* is zero when IDINFO is called; otherwise, the value in *mefstat* remains unchanged.

pidaddr is the address of the waiting parent's ID segment address. If the parent is waiting, the value returned in *pidaddr* is the negative of the parent's ID segment address. If the parent is not waiting, the value returned in *pidaddr* is positive. A value is returned in *pidaddr* only if the parameter is set to zero when IDINFO is called; otherwise, the value in *pidaddr* remains unchanged.

The list of possible values (in octal) for *astat* and *mefstat* follow:

Program State	***astat***	***mefstat***
Dormant	0	0
Dormant saving resources	0	140000
Dormant and in time list	0	100000
Program abort in process	1	−1
I/O suspend	2	2
Program wait suspend	3	3
Operator suspend	6	6
Pause	7	6
Waiting for signal	10	3
Signal buffer limit suspend	46	3
Time suspend	47	100000
Locked device suspend	50	3
Resource number suspend	51	3
Class I/O suspend	52	3
Queue suspend	53	3

Down device suspend	54	3
I/O buffer limit suspend	55	3
Load suspend	56	2
Shared subroutine suspend	57	1
Scheduled	60	1
System Available memory suspend	61	4
Spool suspend	62	3
Extended system available memory suspend	63	4

To ensure that the information remains unchanged between the time IDINFO is called and the returned information is used, the calling program should use privileged operation (see Chapter 12).

KHAR (Character Manipulators)

The following routines are all part of the system library module KHAR. These routines allow a FORTRAN programmer to build and break apart character strings that are contained in integer variables. Note that FORTRAN does provide a character data type, which you will want to use in most cases, but does not work with EXEC calls or these subroutines.

To use the character string routines, source and destination buffers must first be set up. This is accomplished by calling the routines SETSB and SETDB, respectively. After the buffers are set up, the routines KHAR, CPUT, and ZPUT are used to manipulate characters between the source and destination buffer.

SETSB (Set Up Source Buffer)

SETSB sets up the character string source buffer and its limits.

```
CALL SETSB(bufr,chpos,bufln)
```

where:

bufr is an integer buffer where the string to be examined is placed.

chpos is the current character position. This variable is updated by the routine KHAR. It should be initialized to 1 to indicate the first character in *bufr*. The first character in buffer is assumed to be in the left half of the first word of *bufr*.

bufln is an integer defining the number of characters in *bufr*.

SETDB (Set Up Destination Buffer)

SETDB sets up the character string destination buffer.

```
CALL SETDB(dbufr,chcnt)
```

where:

dbufr is the integer destination buffer where parts of the source buffer will be placed by the routines CPUT and ZPUT.

chcnt is the character count indicating the number of characters in *dbufr*. This variable should be initialized to 0 (zero) before calling CPUT or ZPUT. CPUT and ZPUT update the character count in the variable *chcnt*.

No test is done to see if *chcnt* exceeds the dimensions of *dbufr*. *chcnt* may be directly decremented to delete characters, or set to zero to clear the buffer.

KHAR (Subroutine to Get Next Character)

KHAR gets the next character from the source buffer.

```
char = KHAR(dchar)
```

where:

char is an integer variable where the character will be returned.

dchar is the same as *char*.

Both *char* and *dchar* will be zero if there are no more characters in the source buffer. The character that is returned is placed in the left half of the word with a blank padded in the right half.

KHAR increments the variable *chpos* that contains the current character position in the source buffer.

CPUT (Put Character into Buffer)

CPUT puts the specified character in the destination buffer.

```
CALL CPUT(char)
```

where:

char is the character to be put in the destination buffer. The character should be in the left byte (FORTRAN 1H format).

CPUT increments the variable *chcnt* to indicate the number of characters in the destination buffer.

ZPUT (Store a Character String)

ZPUT stores a character string in the destination buffer.

```
CALL ZPUT(zbufr,frstc,noc)
```

where:

zbufr is the integer buffer containing the string to be stored in the destination buffer.

frstc is the position of the first character to be put in the destination buffer.

noc is the number of characters to be put in the destination buffer.

ZPUT increments the variable *chcnt* to indicate the number of characters in the destination buffer.

Character Manipulation Example

The following section of code uses most of the KHAR character routines. The code builds a string of the form "RU,RUN57,MSTOUT::RT,57 " from the strings MASOUT, ACRN, and BUFFER. Notice the use of the function KHAR as an argument to CPUT.

```
      .
      .
      .
      MPOS = 1
      MLIM = 6
      NPOS = 24
      CALL SETSB(MASOUT, MPOS, MLIM)
      CALL SETDB(BUFFER, NPOS)

C STRING NOW LOOKS LIKE "RU,RUN57"; PUT IN THE COMMA AND
C BUILD THE MASTER FILE NAME.
      CALL CPUT(1H,)
      DO 660 J = 1,6
         CALL CPUT(KHAR (KH) )
660   CONTINUE

C NOW PUT IN COLONS AND THE CARTRIDGE REF NUMBER
      CALL CPUT(1H:)
      CALL CPUT(1H:)
      MPOS = 1
      MLIM = 6
      CALL SETSB(ACRN,MPOS,MLIM)
      CALL CPUT(KHAR (KH) )
      CALL CPUT(KHAR (KH) )
      CALL CPUT(KHAR (KH) )
      CALL CPUT(KHAR (KH) )
      CALL CPUT(KHAR (KH) )
      CALL CPUT(KHAR (KH) )
      .
      .
```

8

FMP Routines

The File Management Package (FMP) is a set of routines that manage disk files for RTE-A. FMP calls from a program can open, close, position, read from and write to files, and perform a number of sophisticated file manipulation tasks.

FMP can be called from FORTRAN, Pascal, Macro, or other languages that support subroutine calls. All calling sequences use the .ENTR routine, which is described in the *RTE-A/RTE-6/VM Relocatable Libraries Reference Manual*, part number 92077-90037.

FMP is analogous to the File Manager (FMGR) on other RTE operating systems. Appendix B of this manual is a guide to converting FMGR calls to FMP calls, in order to move software from another RTE operating system to RTE-A.

General Considerations

Most FMP calls access files or file directories. Files contain programs or data; file directories identify and describe files. Refer to the Manipulating Files chapter of the *RTE-A User's Manual* for a detailed description of files, directories, and the file descriptor parameters.

The following calls allow your program to create or delete files or directories, and read or write at various locations in the files. They permit access to information in directories, including type and location information about specific files. Most programs are limited to the calls that access data in files or purge files. Some programs can use the additional higher-level calls. For FMP calls at any level, there is full security and error checking.

FMP Calling Sequence and Parameters

All parameters are required in every FMP call unless the parameter is explicitly documented to be optional. Omitting non-optional parameters causes unpredictable results. Most of the FMP routines can be called as integer functions as well as subroutines. When called as functions, they return values to program variables. When called as subroutines, the function value is returned in the A-Register. In FORTRAN, FMP routines called as integer functions must be declared as integers. The FMP routine names are shown in uppercase and lowercase letters throughout this manual to make it easier to identify their functions, but they can be specified in either case in your program.

The FMP parameters common to most calls, such as the Data Control Block (DCB), file descriptor, and error code, are described in the following sections.

Data Control Block (DCB)

A Data Control Block (DCB) is an integer array, defined by the calling program, that FMP uses to keep information about a file open to the program. A program may have several files open at once, and there must be a DCB for every open file, so the program should define several arrays to contain the DCBs. The FmpOpen routine sets up the DCB contents. Once a file is open, FMP refers to the DCB for file information. The DCB array must be defined as a minimum of 144 words in length. Its contents are maintained entirely by FMP and must not be modified by your program.

The first 16 words of the DCB contain file control information used by the FMP routines. The remaining words are used as a buffer to minimize the number of data transfers to disk. The smallest buffer permitted is one 128-word block. Larger DCB buffers must be a multiple of 128 words (128, 256, 384, and so on), up to a maximum of 127 blocks. The buffer size is independent of the file; a file created with a DCB buffer of 127 blocks can later be accessed with a DCB buffer of 128 words. The buffer only serves to reduce the number of disk accesses. File types 0 and 1 do not require buffers, so a DCB of only 16 words can be used.

File Descriptors

Files are specified by file descriptors, which can contain a file name and an optional file type extension, directory and optional subdirectory information, and a number of optional file type, size, and DS location parameters. File descriptors contain fields for all of the RTE File Manager (FMGR) namr parameters, so files from other RTE operating systems are compatible with FMP on RTE-A.

Refer to the *RTE-A User's Manual* for a full explanation of file descriptors. The following is a brief description. There are three formats for file descriptors:

1. *filnam* : *sc* : *crn* : *type* : *size* : *reclen*

2. */dir/sub/filename .file_type_extension .qual* : : : *type* : *size* : *reclen* [*user*] >*node*

3. *sub/filename .file_type_extension .qual* : : *dir* : *type* : *size* : *reclen* [*user*] >*node*

where:

dir A global directory name of up to 16 characters. The name must conform to the file name convention.

In the second format, the directory name is surrounded by slashes (/), and must appear first in the directory path. If the leading slash is omitted, the first entry is assumed to be a subdirectory.

In the third format, the directory name follows the two colons after the file name. Subdirectories may be specified in the third format. This parameter is optional when creating a file descriptor and defaults to the working directory.

sub One or more subdirectory names of up to 16 characters each. The naming rules for file names apply to subdirectories. In the second or third format, each subdirectory name is followed by a slash (/). In the second format the subdirectory entries may

follow the directory entry in the directory path; in the third format, the subdirectories, if any, make up the entire directory path. As many subdirectories as necessary may appear, with the limitation that the entire file descriptor cannot be longer than 63 characters. This parameter is optional when creating a file descriptor. The alternate directory specifiers ".", "..", and "#n" may be used in file descriptors used in the FMP routines.

filename An FMP file name of up to 16 characters. The file name must conform to the naming conventions described in the *RTE-A User's Manual.*

filnam A FMGR file name of up to six characters; used only in the first format type, for FMGR files. The naming conventions are the same as for FMP file names.

file_type_extension
(Optional) A period followed by one to four characters; it is used to describe the type of information in the file.

qual (Optional) Mask qualifier, separated from the file type extension by a period. Mask qualifiers are described in detail in the *RTE-A User's Manual.*

sc (Optional) A positive integer, a negative integer, or two ASCII characters that conform to the file name conventions. A positive integer other than zero or two ASCII characters protects the file from write attempts. A negative integer provides read and write protection. Used only in the first format, for FMGR files.

crn (Optional) A positive cartridge reference number, the negative logical unit number, or two ASCII characters that conform to the file name conventions. Used only in the first format, for FMGR files.

type (Optional) The RTE file types are as follows:

−1	Symbolic link file.
0	I/O device (non-disk file); variable length records.
1	Random access file; fixed length 128-word records.
2	Random access file; fixed length user-defined records.
3	Sequential access file; variable length records; can be ASCII or binary.
4	ASCII text file; similar to type 3 file.
5	Relocatable binary file; similar to type 3 file.
6	Memory-image program file; similar to type 3 file, but accessed like a type 1 file.
7	Absolute binary program file; similar to type 3 file.
8	and above: user-defined file types, accessed like type 3 files. Special processing based on file type must be supplied by the application program.

size (Optional) The number of blocks in the file.

reclen (Optional) For type 2 files, specifies the length of the records in the file.

user (Optional) The user account name under which the file exists; delimited by square brackets. Used only in systems with the VC+ package.

node (Optional) The DS node where the file resides; preceded by a right angle bracket (>). Used only in systems with the VC+ package.

The first format is the same as a FMGR file descriptor and is used to access files stored in the FMGR file system. The second and third formats are for files stored in an FMP system.

When creating any of these three types of file descriptors, the only parameter required is the file name. When accessing existing CI files, the correct directory/subdirectory path and file type extension must be specified. The optional parameters are used when necessary to more specifically identify a file. Leading (dir and subdir) parameters can be omitted if not required. Trailing (for example, type and size) parameters can also be omitted if not required, but placeholders must be used when parameters are defaulted between specified parameters.

Placeholders and parameter omission are shown in the following examples:

1. `/pubs/manual/devereaux.txt:::4:24[dave]>111`
2. `manual/devereaux.txt.T::pubs:4:24[dave]>111`
3. `manual/devereaux.txt:::::[dave]>111`

All three examples specify the same file. The first uses a leading directory and subdirectory parameter, but omits the mask qualifier and record length fields. The second uses a trailing directory parameter and a leading subdirectory parameter. It specifies all but the record length field. The first two are examples of the second and third file descriptor formats. The third example specifies the file name and file type extension, defaults the directory, type, and size, omits the record length, and specifies the user and DS node.

When the directory and subdirectories are defaulted, the second and third file descriptor formats are the same, because they only differ in their directory specifications.

Character Strings

The FMP calls pass file names as character strings. This eliminates the need to count characters or treat characters as integers. The character strings are stored in the FORTRAN 77 character string format, which is described in the *FORTRAN 77 Reference Manual*, part number 92836-90001.

The FMP routines are coded in FORTRAN 77, so the character strings are treated as fixed-length strings, and are padded or truncated from right to left to fit target strings. Character strings should be left-justified. Zero-length strings are not permitted, so null strings are filled with blanks. Note that nulls in a character string (integer value of zero) are not treated as blanks but are treated as non-blank ASCII characters.

Character strings are not automatically initialized to blanks but are initialized to nulls instead. Therefore, you must ensure that character strings are initialized to blanks. You can use a data statement or blank fill the buffer before the FMP call. For example, in the call "`FmpRpProgram(file,rpname,options,error)` ", blank fill the return buffer before the FMP call as follows:

```
rpname = ' '
```

Compilers (such as Pascal) or assemblers that do not use the FORTRAN 77 character string format must create a file descriptor in a format that the program can manage and that FMP can use.

File Descriptors in Pascal

Pascal supports a variable length character string format that can communicate with FMP routines when used with the Pascal FIXED_STRING compiler option. The Pascal character string format is not directly compatible with the FORTRAN 77 character string format. The Pascal PACKED ARRAY OF CHAR is not compatible with the FORTRAN 77 character string format.

The FIXED_STRING compiler option indicates that string parameters of procedures or functions declared EXTERNAL should be converted from the Pascal variable length character string format to the FORTRAN 77 character string format before being passed.

The current length of the Pascal variable length character string is used as the maximum length of the FORTRAN 77 character string that is passed to the EXTERNAL routine.

Strings that are passed from your program to FMP should have a current length that indicates to FMP the part of the string FMP wants. The current length can include trailing blanks but should not include uninitialized areas of the string.

Strings that FMP sets to an initial value and passes back to your program should have a current length large enough to hold the number of characters expected from FMP (usually a maximum of 63 characters). The length must be greater than zero; FMP truncates or blank pads as necessary. The contents of the string within the current length do not need to be initialized.

Strings that your program passes to FMP and that FMP modifies and returns should have a current length large enough to hold the number of characters expected from FMP. The strings must be blank padded from the end of the data being passed to FMP, out to the current length.

The following Pascal program uses FIXED_STRING to call FMP routines. Note that while a constant is used as the file name to the FmpOpen call, any Pascal string variable or expression with a length less than or equal to the length of the string type PATH could be used. Also, note that anywhere FMP expects a FORTRAN 77 character string parameter, a Pascal string type must be specified in the EXTERNAL declaration and the FIXED_STRING compiler option must be in the ON state.

```
PROGRAM fmpexample;

CONST
   max_file_path    = 63;
   dcb_words        = 144;
   welcome_file     = '/SYSTEM/WELCOME.CMD';

TYPE
   INT  = -32768..32767;
   PATH = STRING [max_file_path];
   DCB  = ARRAY [1..dcb_words] OF INT;

VAR
   error_number:   INT;
   error_message:  PATH;
   file_dcb:       DCB;
   terminal:       TEXT;

$FIXED_STRING ON$

PROCEDURE FmpOpen
   (VAR dcb:  DCB;
```

```
   VAR err:   INT;
       name:  PATH;
       opts:  PATH;
       bufs:  INT); EXTERNAL

PROCEDURE FmpError
   (     err:  INT;
   VAR mess:  PATH); EXTERNAL

$FIXED_STRING OFF$

BEGIN
   rewrite (terminal, '1', 'NOCCTL');

FmpOpen (file_dcb, error_number, welcome_file, 'ROS', 1);

{Check for error on open. If error occurred, make the    }
{current length long enough to hold the message, get the}
{error message from FMP, trim blank padding, and display}
{the message on the terminal.                            }

   IF error_number < 0 THEN BEGIN
      setstrlen (error_message, strmax (error_message));
      FmpError (error_number, error_message);
      error_message := strrtrim (error_message);
      writeln (terminal, welcome_file,
      '(', error_message,')');
   END
   ELSE BEGIN
      .
      .
      .
   END;
END.
```

File Descriptors in Macro

This section describes how to call the StrDsc subroutine from a Macro program to convert character string file descriptors to a format that can be processed by the program and used by FMP.

All FMP calls that take a character string require the caller to pass a file descriptor. FORTRAN 77 does this automatically, but Macro users must set up and pass their own file descriptors. Note that these FMP calls do not work when a buffer of characters is passed as a parameter when a string is expected.

The StrDsc subroutine returns a two-word descriptor that describes a character buffer of a specified length, beginning at a specified character position. The characters in the buffer are numbered from 1 to the number of characters. The resulting two-word descriptor can be passed as an input or output parameter anywhere a FORTRAN 77 character string parameter is required. The string is transferred to and from the buffer described by the two-word descriptor. StrDsc is described in the *RTE-A/RTE-6/VM Relocatable Libraries Reference Manual*, part number 92077-90037.

The following example opens a file with a known name and options string:

```
        ext FmpOpen, StrDsc
*
* Create a file descriptor for the name
*
        jsb StrDsc
        def *+4
        def nbuffer
        def =d1
        def =d19
        dst filename
*
* And the options string
*
        jsb StrDsc
        def *+4
        def obuffer
        def =d1
        def =d3
        dst options
*
* Open the file
*
        jsb FmpOpen
        def *+6
        def dcb
        def err
        def filename
        def options
        def =d1
        ...
*
* Constants and data
*
nbuffer  asc 10,WELCOME.CMD::SYSTEM
obuffer  asc 2,ROS
filename bss 2
options  bss 2
dcb      bss 144
```

Note that

```
        jsb FmpOpen
        def *+6
        def dcb
        def err
        def nbuffer ; wrong!
        def obuffer ; also wrong!
        def =d1
```

does not work with `nbuffer` and `obuffer` declared as above.

Because `filename` and `options` define string constants, the string descriptors could be defined as follows:

```
filename dec 20          ;string byte length
         dbl nbuffer     ;string byte address
options  dec 4           ;string byte length
         dbl obuffer     ;string byte address
```

The two words associated with `filename` and `options` must appear in the order shown. If string descriptors are defined in this manner, the StrDsc routine is not necessary.

Error Returns

Errors can occur on FMP calls; for example, it is an error to try to open a non-existent file. The error is returned as a negative value, either as the function return value or in an error parameter. The error value can be passed to an error processing or reporting subroutine in your program. The error return values are listed in Appendix A. The FMP routines must be declared as integer functions in FORTRAN to receive the correct error code as the function return value.

Transferring Data to and from Files

In addition to the Data Control Block, a user buffer must be defined in the calling program for transferring individual records to and from files. Records to be sent to files must be stored in the user buffer before a write call. Records read from files are returned to the user buffer. The relationship between the user buffer, the Data Control Block buffer, and a disk file is illustrated in Figure 8-1.

Each call that reads or writes a record transfers one record between the user buffer and the Data Control Block buffer. Such transfers within memory are known as logical reads or writes.

A physical read or write transfers a block of data between the disk file and the Data Control Block buffer. A physical write is performed automatically when the DCB buffer is full, when a file is closed, or when a request for a physical write is made with the FmpPost call.

On a read request, a block of data is physically read into the DCB buffer from the disk only if the entire requested record is not already in the buffer. If a needed record is not already within the DCB buffer, (see record 7 in Figure 8-1), then FMP performs physical reads or writes of blocks until the entire record has been transferred.

For type 1 file accesses, the intermediate transfer to the DCB buffer is omitted and each 128-word record is transferred directly between the user buffer and the file as shown in Figure 8-2. Such accesses are faster than transfers through the DCB buffer.

Non-disk (type 0) file reads and writes also bypass the DCB buffer. Records in type 0 files are written or read directly to or from the device identified as a type 0 file. Words, rather than records, are the units of type 0 transfers to accommodate the record lengths of various devices.

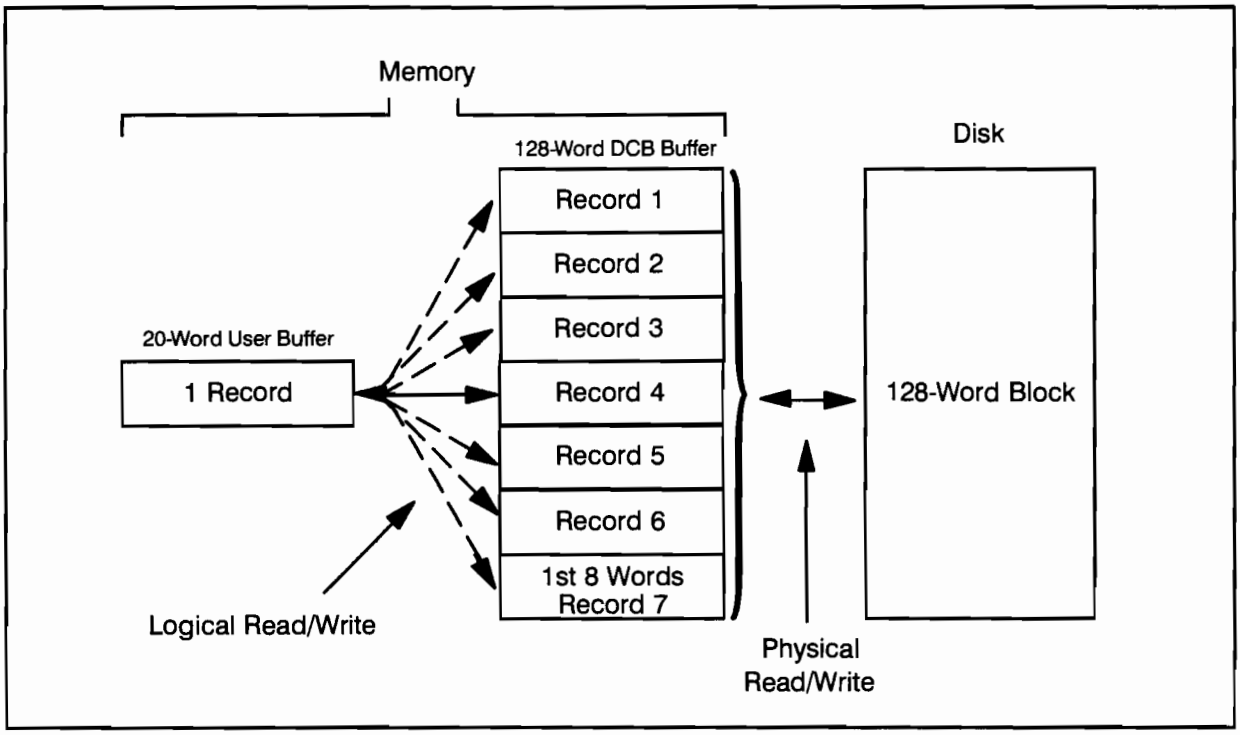


Figure 8-1. Logical Transfer Between Disk File and Buffers

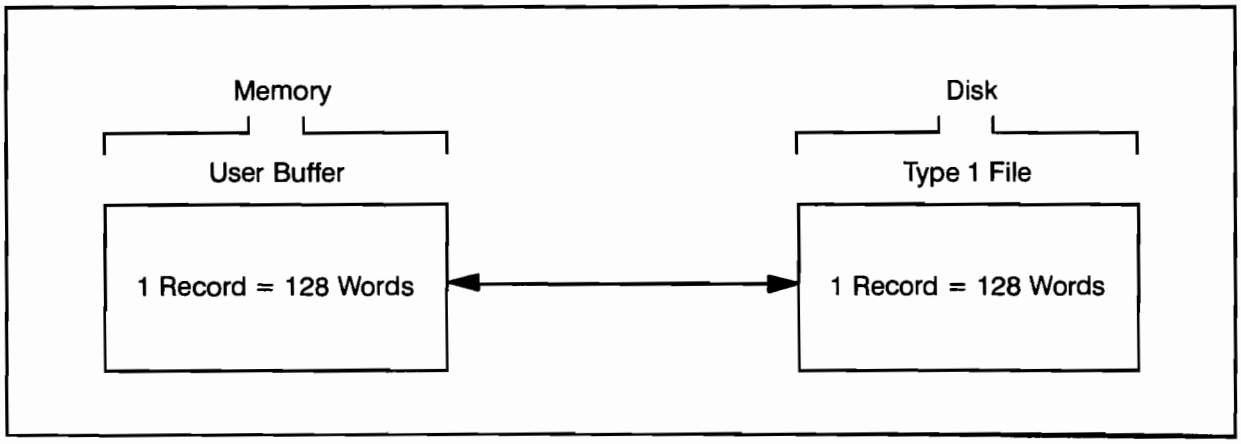


Figure 8-2. Data Transfers with Type 1 Files

Descriptions of FMP Routines

This section contains descriptions of all FMP routines; the routines are listed alphabetically. Tables 8-1 through 8-6 present functional groupings of the routines.

Table 8-1. File Manipulation FMP Routines

FMP Routine	Purpose
FmpOpen	Opens a file for access
FmpOpenScratch	Opens file on scratch directory
FmpOpenTemp	Opens a temporary file
FmpClose	Closes a file to end access
FmpRead	Reads from a file
FmpReadString	Reads a character string from a file
FmpWrite	Writes to a file
FmpPagedWrite	Writes to a file, calling FmpPaginator to break output into screen pages for terminal devices
FmpWriteString	Writes a character string to a file
FmpPosition	Returns the current file position
FmpRewind	Sets file position to the first word of the file
FmpSetPosition	Changes the file position
FmpSetWord	Changes the file position
FmpAppend	Positions a file to the EOF mark
FmpSetEof	Sets EOF mark at the current position
FmpPost	Posts data to the file
FmpTruncate	Truncates the file
FmpSetDcbInfo	Changes information in the DCB
DcbOpen	Indicates if a DCB is open
FmpMakeSLink	Create a symbolic link file.
FmpReadLink	Read the contents of a symbolic link file.

Table 8-2. Directory Access FMP Routine

FMP Routine	Purpose
FmpCreateDir	Creates a directory
FmpWorkingDir	Returns the working directory
FmpSetWorkingDir	Changes the working directory
FmpInfo	Returns the directory information for the file
FmpSetDirInfo	Changes information in a directory
FmpMount	Mounts a volume
FmpDismount	Dismounts a volume
FmpFileName	Returns the full path name of a file
FmpOpenFiles	Indicates which files in a directory are open
FmpOwner	Returns the name of the directory owner
FmpSetOwner	Changes the name of the directory owner
FmpCreateTime	Returns the time that the file was created
FmpAccessTime	Returns the time of the last access
FmpUpdateTime	Returns the time of the last update
FmpRecordCount	Returns the number of records in the file
FmpRecordLen	Returns the length of the longest record in the file
FmpProtection	Returns the access available to file or directory
FmpSetProtection	Changes the access to a file or directory
FmpEof	Returns the position of the EOF mark
FmpPostEof	Posts the EOF position and the number of records from the DCB to the directory entry
FmpSize	Returns the physical size of the file
FmpRename	Changes the file name
FmpPurge	Purges a file
FmpDcbPurge	Purges an open file
FmpUnPurge	Restores a purged file
FmpUdspInfo	Returns current UDSP information for the session
FmpUdspEntry	Returns the directory name in specified UDSP entry

Table 8-3. Masking FMP Routines

FMP Routine	Purpose
FmpInitMask	Initializes data structures for the FMP mask calls
FmpNextMask	Returns the directory entry for the next file matching
FmpMaskName	Builds a full name for a file matching the mask
FmpEndMask	Closes the files associated with a mask search
WildCardMask	Checks for wildcard characters in a mask
FattenMask	Modifies the mask
MaskOldFile	Determines if a specified file is an FMGR file
MaskMatchLevel	Returns the directory level of the last file matched
MaskDiscLu	Returns the disk LU of the last file returned by FmpNextMask
MaskOpenId	Returns the D.RTR open flag of the last file returned by FmpNextMask
MaskOwnerIds	Returns the owner and group IDs for the last file returned by FmpNextMask
MaskSecurity	Returns the security code of the last file returned by FmpNextMask
Calc_Dest_Name	Creates a destination file name from a file name, match level, and destination mask

Table 8-4. Device FMP Routines

FMP Routine	Purpose
FmpBitBucket	Determines whether type 0 file is LU 0
FmpDevice	Indicates whether a DCB is associated with a device file
FmpInteractive	Indicates whether a DCB is associated with an interactive device
FmpIoOptions	Returns the I/O options word
FmpSetIoOptions	Changes the I/O options word
FmpIoStatus	Returns the A- and B-Register values of last I/O request
FmpControl	Issues a control request to an LU
FmpLu	Returns the LU of the file or device
FmpPagedDevWrite	Performs XLUEX(2) write to interactive device, with page breaking

Table 8-5. Parsing FMP Routines

FMP Routine	Purpose
FmpBuildHierarch	Builds a file descriptor in hierarchical format from its component fields
FmpBuildName	Builds a file descriptor from its component fields
FmpBuildPath	Builds a file descriptor that includes hierarchical directory information and file masks from its component fields
FmpHierarchName	Converts a file descriptor to hierarchical format
FmpStandardName	Converts a file descriptor to the standard format
FmpLastFileName	Returns the last file name in a path
FmpParseName	Parses a file descriptor into its component fields
FmpParsePath	Parses a file descriptor that includes hierarchical directory information and file masks into its component fields
FmpShortName	Returns the shortened version of a file descriptor
FmpUniqueName	Creates and returns a unique file name

Table 8-6. Utility FMP Routines

FMP Routine	Purpose
DcbOpen	Indicates whether a DCB is open
FmpCopy	Copies a file to another file
FmpList	Lists a file to a specified LU
FmpError	Returns an error message for an FMP error code
FmpReportError	Prints an error message for an FMP error on LU 1
FmpExpandSize	Unpacks file size word to double integer
FmpPackSize	Packs double integer file size into one word
FmpCloneName	Generates program clone names
FmpRpProgram	Restores a program,
FmpRunProgram	Schedules a program
FmpRwBits	Checks a string for the letters R and W
FmpPaginator	Prompts for pagebreaks for FmpList, FmpPagedWrite, and FmpPagedDevWrite routines.

Calc_Dest_Name

Calc_Dest_Name generates a full destination file name.

```
CALL Calc_Dest_Name(sourcename, matchlevel, destmask, destname)

character*(*) sourcename, destmask, destname
integer*2 matchlevel
```

where:

sourcename	is a character string that specifies a full source file descriptor.
matchlevel	is an integer that specifies the number of the directory level in which the last file was matched as returned by MaskMatchLevel.
destmask	is a character string that specifies the destination mask.
destname	is a character string that returns the full destination file descriptor.

Calc_Dest_Name uses a file name, its *matchlevel* (returned by the MaskMatchLevel routine), and a destination mask, and generates a full destination file name. If the destination mask contains an "@" in the file name or file type extension fields, then the *sourcename* values of those fields are used. The Command Interpreter (CI) CO and MO commands use Calc_Dest_Name generated destination names.

DcbOpen

DcbOpen returns an integer value that indicates whether or not the specified DCB is open.

```
error = DcbOpen(dcb,error)

integer*2 dcb(*), error
```

where:

dcb	is an integer array containing the DCB for the file.
error	is an integer indicating the status of the DCB. If the DCB is open, *error* is set to zero. If the DCB is not open, *error* is set to a negative error code.

FattenMask

FattenMask modifies the mask parameter by adding the character "@" to the name or file type extension if it is implied by the mask.

```
CALL FattenMask(mask,how)

character*(*) mask
integer*2 how
```

where:

mask	is a character string specifying the mask to be modified.
how	is an integer specifying how the mask is to be modified. If bit 0 is set, a "D" is appended to the qualifier. If bit 1 is set and the mask is blank, "@" is not inserted in either the name or file type extension.

If the name field of mask is blank, the "@" character replaces the blank. If the name field ends with "@" and the file type extension is omitted, then a file type extension of ".@" is inserted. If the mask is a global directory in the form /global, the file type extension .DIR is appended because it is the only file type extension possible for a global directory.

The overall purpose of this call is to make implied constructs such as /DIR/ explicit, by converting them to the fuller /DIR/@.@.D described in the last paragraph.

FmpAccessTime

FmpAccessTime returns the time of the last access for the named file. The file does not have to be open, and it is not opened to read the access time.

```
error = FmpAccessTime(filedescriptor,time[,slink])

character*(*) filedescriptor
integer*2 error
integer*4 time
logical slink
```

where:

error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
filedescriptor	is a character string that specifies the name of the file.
time	is a double integer that returns the time of the last access expressed as the number of seconds since Jan 1, 1970.
slink	is an optional boolean variable that indicates whether to return the access time of a symbolic link or the file that it references. The possible values are as follows:

TRUE (negative value)
Return the access time of the symbolic link file.

FALSE (non-negative value)
Return the access time of the file referenced by the symbolic link (this is the default).

The access time is changed when a file is opened. It is not affected by calls that do not open the file, such as FmpRead or FmpClose. Access time is generally used to check activity on a file; inactive files that have outlived their usefulness, are often purged to make room for other files.

Routines are available to convert the returned time to an ASCII string. Usually, however, the returned time is compared to other times in the same format, so it may not be necessary to convert the returned time.

FmpAppend

FmpAppend positions a file of type 3 or above to the end-of-file mark to prepare for adding records to the file.

```
error = FmpAppend(dcb,error)

integer*2 dcb(*), error
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

The file must be open for write access, and must be a type 3 or above file; FmpAppend has no effect on device files, or type 1 and 2 files. FMGR files must be open for write and read access.

The effect of FmpAppend is the same as calling FmpEof and using the returned value in an FmpSetPosition call to position the file to the EOF. FmpAppend removes one step from the process.

Note that FmpEof uses the EOF position in the directory entry. Therefore, it is possible for this value to be incorrect if the program that is writing to the file is terminated before it is able to post the new EOF position with an FmpClose or FmpSetEof call.

FmpBitBucket

FmpBitBucket determines if the type 0 file associated with the specified DCB is LU 0 (the bit bucket).

```
bool = FmpBitBucket(dcb)

logical bool
integer*2 dcb(*)
```

where:

dcb is an integer array containing the DCB for the type 0 file.

bool is a flag that is set to TRUE (negative value) if the DCB is open and associated with a type 0 file, and the device is LU 0; otherwise, *bool* is set to FALSE (non-negative value).

FmpBuildHierarch

FmpBuildHierarch constructs a file descriptor in the hierarchical format.

```
error = FmpBuildHierarch(filedescriptor, dirpath, name, typex, qual, sc, type, size, rl, ds)

character*(*) filedescriptor, dirpath, name, typex, qual, ds
integer*2 sc, type, size, rl
```

where:

filedescriptor	is a 63-character string that returns the file descriptor.
dirpath	is a character string specifying the directory/subdirectory path. *dirpath* can be a maximum of 63 characters.
name	is a character string specifying the file name. *name* can be a maximum of 63 characters.
typex	is a character string specifying the file type extension. *typex* can be a maximum of 4 characters.
qual	is a character string specifying the mask qualifier. *qual* can be a maximum of 40 characters.
sc	is an integer that specifies the security code of a FMGR file.
type	is an integer that specifies the file type.
size	is an integer that specifies the size of the file in blocks.
rl	is an integer that specifies record length.
ds	is a character string that specifies the DS node name, a user name, or both. *ds* can be a maximum of 63 characters.
error	is an integer error return. The only possible error is −231 (string too long) which is returned if the string will not fit in the file descriptor. If the call was successful, *error* returns a non-negative value.

The *dirpath* parameter must conform to the following conventions:

- The global directory and each subdirectory name be followed by a slash (/).
- *dirpath* must begin with a slash except in the following cases:
  - If the file descriptor is specified relative to the working directory and one or more subdirectories are specified, *dirpath* must begin with the name of the highest-level subdirectory (for example, SUBDIR1/SUBDIR2).
  - If the file descriptor is specified relative to the working directory and no subdirectories are specified, *dirpath* must be blank.

If any of the component fields are zero or blank, the corresponding field in the *filedescriptor* parameter is left empty, with any necessary placeholders. All delimiters except those in the DS field are automatically inserted. The *ds* delimiters must be included in the *ds* parameter string. Trailing fields that are zero or blank are omitted without placeholders. There is no error detection for the component fields, so illegal parameters generate an illegal file descriptor.

FmpBuildName

FmpBuildName creates a file descriptor from its component fields. It is the inverse of FmpParseName. Its call sequence is the same as FmpParseName, but the component fields are specified, and the file descriptor is returned.

error = FmpBuildName(*filedescriptor*, *name*, *typex*, *sc*, *dir*, *type*, *size*, *rl*, *ds*)

```
character*(*) filedescriptor, name, typex, dir, ds
integer*2 sc, type, size, rl
```

where:

filedescriptor	is a 63-character string that returns the file descriptor.
name	is a character string that specifies subdirectories (if any) and the file name. *name* can be a maximum of 63 characters.
typex	is a character string that specifies the file type extension. *typex* can be a maximum of 4 characters.
sc	is an integer that specifies the security code of a FMGR file.
dir	is a character string that specifies the global directory name. *dir* can be a maximum of 16 characters.
type	is an integer that specifies the file type.
size	is an integer that specifies the size of the file in blocks.
rl	is an integer that specifies record length.
ds	is a character string that specifies the DS node name, a user name, or both. *ds* can be a maximum of 63 characters.
error	is an integer error return. The only possible error is −231 (string too long) which is returned if the string will not fit in the file descriptor.

If any of the component fields are zero or blank, the corresponding field in the *filedescriptor* parameter is left empty, with any necessary placeholders. All delimiters except those in the DS field are automatically inserted. The DS delimiters must be included in the *ds* parameter string. Trailing fields that are zero or blank are omitted without placeholders. There is no error detection for the component fields, so illegal parameters generate an illegal file descriptor.

FmpBuildName example:

Assume that *name* = SANJOSE and *dir* = CITIES.

```
error = FmpBuildName(fdesc,name,'txt',0,dir,4,24,0,'  ')
```

fdesc returns SANJOSE.TXT::CITIES:4:24.

FmpBuildPath

FmpBuildPath constructs a file descriptor from its component fields. It is similar to FmpBuildName, except that it more conveniently constructs file descriptors that contain hierarchical directory information, and it permits creation of file descriptors that contain a file mask qualifier. It is also similar to FmpBuildHierarch except that it creates file descriptors in the standard format, described in the FmpStandardName section.

error = `FmpBuildPath(`*filedescriptor*`,` *dirpath*`,` *name*`,` *typex*`,` *qual*`,` *sc*`,` *type*`,` *size*`,` *rl*`,` *ds*`)`

```
character*(*) filedescriptor, dirpath, name, typex, qual, ds
integer*2 sc, type, size, rl
```

where:

filedescriptor	is a 63-character string that returns the file descriptor.
dirpath	is a character string that specifies the directory/subdirectory path. *dirpath* can be a maximum of 63 characters.
name	is a character string that specifies the file name. *name* can be a maximum of 16 characters.
typex	is a character string that specifies the file type extension. *typex* can be a maximum of 4 characters.
qual	is a character string that specifies the mask qualifier. *qual* can be a maximum of 40 characters.
sc	is an integer that specifies the security code of an FMGR file.
type	is an integer that specifies the file type.
size	is an integer that specifies the size of the file in blocks.
rl	is an integer that specifies record length.
ds	is a character string that specifies the DS node name, a user name, or both. *ds* can be a maximum of 63 characters.
error	is an integer error return. The only possible error is −231 (string too long) which is returned if the string will not fit in the file descriptor.

The *dirpath* parameter must conform to the following conventions:

- The global directory and each subdirectory name must be followed by a slash (/).
- *dirpath* must begin with a slash, except in the following cases:
  - If the file descriptor is specified relative to the working directory and one or more subdirectories are specified, *dirpath* must begin with the name of the highest-level subdirectory, as in SUBDIR1/SUBDIR2/.
  - If the file descriptor is specified relative to the working directory and no subdirectories are specified, *dirpath* must be blank.

If any of the component fields are zero or blank, the corresponding field in the *filedescriptor* parameter is left empty, with any necessary placeholders. All delimiters except those in the *ds* and *dirpath* fields are automatically inserted. The DS and hierarchical directory path delimiters must be included in the *ds* and *dirpath* parameters. Trailing fields that are zero or blank are omitted without placeholders. There is no error detection for the specified parameters, so illegal parameters generate an illegal file descriptor.

FmpBuildPath is the inverse of FmpParsePath. It has the same calling sequence, and uses the same parameters, except that the component fields are specified and a file descriptor is built and returned.

FmpBuildPath example:

```
Path = /CITIES/CALIFORNIA/, file = @, qual = D.

CALL FmpBuildPath(fdesc,path,file,'TXT','D',0,4,24,0,' ')
```

`fdesc` returns `/CITIES/CALIFORNIA/@.TXT.D:::4:24`

FmpCloneName

FmpCloneName generates program clone names that can be used by FmpRpProgram.

```
CALL FmpCloneName(name,init)

character*(*) name
logical init
```

where:

name is a character string that specifies the program name to be cloned. The specified name is modified by the system and returned to the calling program.

init is a logical indicating whether the current call is the first call to FmpCloneName.

Before calling FmpCloneName for the first time, set the *init* parameter to TRUE (negative value). When the call is executed, FmpCloneName resets the value to FALSE (non-negative value).

The sequence of names generated by FmpCloneName is as follows (PROG is the original program name):

```
PROG, PRO.A, PRO.B, ..., PRO.Z, PROAA, PROAB, ..., PROZZ
```

FmpCloneName can be called in a loop to generate program names until a name that does not already exist on the system is found. This name then can be used in an FmpRpProgram call to RP a program.

FmpClose

FmpClose closes a file, and removes its entry from the FMP open file table.

```
error = FmpClose(dcb,error)

integer*2 dcb(*), error
```

where:

dcb	is an integer array containing the DCB for the file.
error	is an integer that returns a negative code if an error occurs or zero if no error occurs.

If the program wrote data to the file while it was open, the FmpClose call sets the time of last update to the system time when the file is closed. It also sets the backup bit in the directory. FmpClose also sets the end-of-file position in the directory to the file position at the time of the close, if the DCB specified a sequential file positioned at EOF. If FmpClose finds the DCB not open, no error will be returned and the *error* parameter will be zero.

Files should be closed after a program's access is finished, to make sure that all writes are posted to the disk, and to unlock files or devices to make them available to other programs. It is good practice to close files after access is finished, whether or not write accesses were performed.

FmpControl

FmpControl performs an I/O device control (EXEC 3) request on the LU associated with a device file DCB.

```
error = FmpControl(dcb,error,pram1,pram2,pram3,pram4)

integer*2 dcb(*), error, pram1, pram2, pram3, pram4
```

where:

dcb	is an integer array containing the DCB of a device file.
error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
pram1	is the control word (*cntwd*) of the EXEC call.
pram2 - *pram4*	are integers that can be passed as parameters to the EXEC call. The resulting EXEC call is equivalent to the following: `CALL EXEC(3,cntwd,pram2,pram3,pram4)` where *cntwd* contains the function code and the device LU associated with the DCB.

FmpCopy

FmpCopy copies one file to another.

error = FmpCopy(*name1*, *err1*, *name2*, *err2*, *buffer*, *buflen*, *options*)

```
character*(*) name1, name2, options
integer*2 buffer(*), buflen, err1, err2
```

where:

name1 is a character string that specifies the source file or logical unit.

err1 is an integer that returns errors associated with *name1*.

name2 is a character string that specifies the destination file or logical unit.

err2 is an integer that returns errors associated with *name2*.

buffer is an integer buffer that contains the source and destination DCBs and DCB buffers. *buffer* must be a minimum of 288 words in length.

buflen is an integer that specifies the length of the buffer in words. *buflen* must be set to at least 288 words.

options is a character string that specifies the data transfer mode if the source or destination is a device, as well as manipulation of the source and destination files. *options* can be set to any of the following values, either singly or in combination (such as PD):

A	ASCII
B	Binary
C	Clear backup bit on source
D	Overwrite existing file
I	Inhibit LU locking of non-interactive devices
N	Source does not have carriage control
P	Purge source after copy
Q	Quiet; do not record access time on source
S	Preserve directory information (timestamps, protection, and backup bit) of the source file
T	Truncate destination to length of valid data
U	Replace duplicate file if update time is older

FmpCopy works for all file types, including type 6 files, and type 1 or 2 files with missing extents. It uses the most efficient copy operation that works for the given files.

The calling program must specify a work buffer to contain the source and destination file DCBs and transferred records. The buffer must be at least large enough to contain two DCBs of 16 words each, plus two 128-word (one block) DCB buffers. The minimum buffer size, thus, is (2 * 16) + (2 * 128) = 288 words. The larger the buffer is, the faster the copy operation can execute. Larger buffers must be larger by 128-word increments.

When using FmpCopy to copy a type 2 file to a device or to copy from a device to a type 2 file, the work buffer must be at least large enough to contain the following:

- two DCBs of 16 words each,
- one 128-word DCB buffer for the source file, and
- one record buffer the size of the type 2 record length.

Therefore, the minimum buffer size, in words, is (2 * 16) + 128 + the record length. For optimal performance the work buffer should be made as large as possible. Type 2 files with a record length of 16384 words or greater cannot be transferred to or from devices. (File to file transfers are permitted.)

When copying from a device to another device or from a device to a type 1 file, the work buffer is divided into two DCBs of 16 words each and a record buffer. When the record length of the source device is larger than the record buffer, the records are truncated. It is the caller's responsibility to ensure that the work buffer is large enough to contain the two DCBs and a record buffer large enough to contain the maximum record length on the source device. (FmpCopy cannot determine the maximum record length on a source device and also cannot detect when a record from the source device is being truncated.)

Regardless of the size of the work buffer specified, FmpCopy truncates any records read from a source device that have a record length greater than 32512 bytes.

The A and B options are used only when the source or destination is a device. If the destination is a device or a type 3 or 4 file, and the source is a device, the default option is A. In all other cases, the default option is B.

If the destination name does not specify a file type, the source file type is used. If the source is a device and the A option is in effect, the default destination type is 3; if the B option is in effect, the default destination type is 6.

If the destination name does not specify a size, the total size of the source file (the sum of the sizes of the main and all its extents) is used. As a result, the destination file does not have any extents. If the source is a device, the default size is 24 blocks.

If the destination name does not specify a record length, the record length of the source file is used. If the source record length is greater than 128 words, the record length of the destination file is truncated to 128 words.

FmpCopy tests the break flag while copying. If it finds it set, it stops copying and reports error −235 (Break Detected). If the calling program uses the break flag, it should use the error indication to detect breaks when FmpCopy is used.

If either *err1* or *err2* contains an error code, the same error code is returned in *error*. If *error* = 0, then neither *err1* nor *err2* contains an error code.

The Q option is used when the user does not want to have the access time of the file updated. With the Q option, there is no attempt to update the access time. The Q option is useful when copying from a file residing on a write-protected disk. Normally, the file system would attempt to update the file access time when opening the file and, because the LU is write-protected, the CO command would fail.

The protection of the destination file will be that of the source file provided the source is not an LU or a FMGR file and the caller is the owner of the destination directory. Otherwise, the destination file will have the protection of the directory into which it is copied.

The S option allows you to save directory information (timestamps, protection, and backup bit) of the source file.

The T option allows you to copy a file that has wasted space into a new file as a perfect fit. The end-of-file directory information of the source file is used to determine how many blocks of valid data to copy to the destination file. This option has no effect on type 1, 2, and 6 files and FMGR files.

The U option allows you to overwrite the destination file only if the destination file's update time is older than that of the source. Because FMGR files do not have update times, they are considered the oldest.

FmpCreateDir

FmpCreateDir creates a directory.

```
error = FmpCreateDir(name,lu)

character*(*) name
integer*2 lu
```

where:

name	is a character string specifying the name of the directory to be created.
lu	is an integer specifying the disk LU on which to create the directory.

A global directory is specified by a name beginning with "::" or "/", as in ::USERS or /USERS. A subdirectory is specified with its parent directory, separated by "::", as in SUBDIR::USERS or /DIR/SUBDIR. The parent directory must already exist.

The calling program can specify a size (::DIRNAME::12), to a maximum of 64 blocks. The default size is the number of blocks per track on the disk LU.

Subdirectories are placed on the same LU as their parent directory. Global directories are placed on the specified LU. If LU 0 is specified, the global directory is created on the same LU as the working directory, if any, or on the lowest numbered disk LU on which directories can be created.

The default protection for a global directory is RW/R/R. The default protection for a subdirectory is the protection of the directory in which it is created.

FmpCreateTime

FmpCreateTime returns the time of creation for the named file. The file is not opened in the process.

```
error = FmpCreateTime(filedescriptor,time[,slink])

character*(*) filedescriptor
integer*4 time
logical slink
```

where:

filedescriptor	is a character string that specifies the name of the file.

time is a double integer that returns the time that the file was created, expressed in seconds since January 1, 1970.

slink is an optional boolean variable that indicates whether to return the create time of a symbolic link or the file that it references. The possible values are as follows:

TRUE (negative value)
Return the create time of the symbolic link file.

FALSE (non-negative value)
Return the create time of the file referenced by the symbolic link (default).

The create time is set when the file is created, and is never changed afterwards, except by the FmpSetDirInfo routine.

Routines are available to convert the returned time to an ASCII string. Usually, however, the returned time is compared to other times in the same format, so the calling program may not have to convert the format.

FmpDcbPurge

FmpDcbPurge closes and purges the open file associated with the given DCB.

```
error = FmpDcbPurge(dcb)

integer*2 error, dcb(*)
```

where:

error is an integer that returns a negative code if an error occurs.

dcb is an integer array containing the open DCB for the file.

FmpDcbPurge performs the combined functions of FmpClose and FmpPurge. This routine is useful where it is important that there be no time lag between the time the file is closed and the time it is purged. This routine prevents re-opening or moving a file after it is closed but before it is purged.

FmpDevice

FmpDevice indicates whether the specified DCB is associated with a device file.

```
flag = FmpDevice(dcb)

logical flag
integer*2 dcb(*)
```

where:

flag is a boolean set to TRUE (negative value) if the specified DCB is associated with a device file. *flag* is set to FALSE (non-negative value) if the DCB is associated with a disk file or is not open.

dcb is an integer array containing the DCB for the file.

FmpDismount

FmpDismount dismounts a disk volume.

```
error = FmpDismount(lu)

integer*2 error, lu
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

lu is an integer that specifies the LU of the disk volume.

Global and subdirectories on the specified LU are made unavailable, and the disk is removed from the cartridge list.

If there are any open files, RP'd programs, working directories, or directories contained in a UDSP on the volume, D.RTR reports an error identifying the first such conflict that it finds.

FmpEndMask

FmpEndMask closes the files associated with a mask search.

```
CALL FmpEndMask(dirdcb)

integer*2 dirdcb(*)
```

where:

dirdcb is an integer array initialized by FmpInitMask.

FmpEndMask should always be called after a masked search terminates. If it is not called, directories may be left open to your program after the search ends.

FmpEof

FmpEof returns the current word position of the end-of-file mark for the specified file.

error = FmpEof(*filedescriptor*, *eofpos*[, *slink*])

```
integer*2 error
character*(*) filedescriptor
integer*4 eofpos
logical slink
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string specifying the name of the file.

eofpos is an integer that returns the word position of the last word in the main file area, or of the highest numbered extent, if any, plus 1.

slink is an optional boolean variable that indicates whether to return the word position of the EOF of a symbolic link or the file that it references. The possible values are as follows:

TRUE (negative value)
Return the word position of the EOF of the symbolic link file.

FALSE (non-negative value)
Return the word position of the EOF of the file referenced by the symbolic link (this is the default).

The first word in the file is word 0, so if *eofpos* = 0 for a file of type 3 or above, the file is empty. For type 1 or 2 files, *eofpos* is the word position of the last word in the main file area, or of the highest numbered extent, if any, plus 1.

If the file is currently open, the returned value may not be accurate because the program that has it open may have added to the file without updating the EOF position in the directory entry. The EOF position in the directory entry is set by an FmpClose or FmpSetEof call.

FmpError

FmpError returns a string that describes the error identified by the *error* parameter. FmpError should be used to report errors to ensure consistent error reporting.

```
CALL FmpError(error,message)

character*(*) message
integer*2 error
```

where:

error is an integer that specifies the error code.

message is a character string variable that returns an error message (for example, "NO SUCH FILE" or "CANNOT PURGE FILE").

The list of possible messages is given in Appendix A. The maximum error description length is 40 characters. If there is not a defined error message for the error identified by the error parameter, a generic error message in the form "FMP error -*xxx*" is issued by the system.

The system program D.ERR generates the text of FMP error messages. If an FMP error occurs and the system cannot find D.ERR, the following message is generated:

```
(warning -250) FMP error xxx
```

The error code −250 indicates that D.ERR was not available and *xxx* is the FMP error that occurred.

FmpError should be used by programs that need more flexible error processing than is provided by FmpReportError.

FmpExpandSize

FmpExpandSize unpacks the size word into a double integer value that specified the number of blocks in the file.

```
blocks = FmpExpandSize(size)

integer*2 size
integer*4 blocks
```

where:

blocks is a double integer indicating the number of blocks in the file.

size is an integer indicating the size of the file, in one word.

If *size* > 0, then the number is not changed. If *size* < 0, it is multiplied by −128.

For FMGR files, the packed size must be divided by 2 if it is positive, before the call to FmpExpandSize. If the *size* parameter of a FMGR file is negative, it works just as an FMP file size.

FmpFileName

FmpFileName returns the full file descriptor of the file associated with the specified DCB.

```
error = FmpFileName(dcb,error,filedescriptor)

integer*2 dcb(*), error
character*(*) filedescriptor
```

where:

dcb	is an integer array containing the DCB for the specified file.
error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
filedescriptor	is a character string that returns the name of the file associated with the specified DCB. The file descriptor includes the full directory path, and file type, size, and (for type 2 files) record length, returned in decimal ASCII. The size is the total size of the file, including extents. For remote files, the file descriptor includes the user name and remote node name.

The normal string assignment rules apply to the returned string, although FmpFileName never returns a file descriptor longer than 63 characters. The file descriptor is truncated to fit in 63 characters, even if it causes an incorrect name to be returned by truncating part of the file name or the directory name.

FmpFileName can be used to return the file descriptor of an open file for use in other calls that need a file descriptor, or for use in error reporting routines. The DCB must be open when the call is made.

FmpHierarchName

FmpHierachName converts a file descriptor to the hierarchical format, in which leading (/DIR/FILE) directory notation, rather than trailing (FILE::DIR), is always used.

```
error = FmpHierarchName(filedescriptor)

character*(*) filedescriptor
```

where:

filedescriptor	is a character string containing the file descriptor to be converted.
error	is an integer error return. The only possible error is −231 (string too long) which is returned if the string will not fit in the file descriptor. If the call was successful, *error* returns a non-negative value.

Hierarchical names are much easier to use in programs that manipulate hierarchical directory structures. They cannot be used for FMGR files, however, so programs that must process FMGR files should call FmpStandardName to convert names to the FMGR-compatible standard format before passing the file descriptor to routines such as FmpOpen.

FmpInfo

FmpInfo returns a copy of the directory entry for the file specified by the DCB. It allows the calling program to get all of the information in the directory with minimum delay. This call should not be used unless absolutely necessary because it is likely to be affected by future changes to the directory structure.

```
error = FmpInfo(dcb,error,info,flag)

integer*2 dcb(*), error, info(32), flag
```

where:

dcb	is an integer array containing the DCB for the file.
error	is an integer that returns a negative code if an error occurs or non-negative code if no error occurs.
info	is a 32-word integer array into which the directory information is returned. For FMGR, only the first 16 words are used; the last 16 words are zeros.
flag	is an integer flag that returns the file system type ~~indicating the file system required~~; 0 for FMGR files and one (1) for FMP files.

FmpInitMask

FmpInitMask initializes the buffers, pointers, and control constructs used by FmpNextMask to select file names according to a file mask.

```
error = FmpInitMask(dirdcb,error,mask,diropenname,dcblen,[msc])

integer*2 dirdcb(*), error, dcblen, msc
character*(*) mask, diropennam
```

where:

dirdcb	is a control array of at least 372 words to be used only with FmpNextMask. A value of *dirdcb* longer than 372 words, up to 8308 words, may be provided to improve masking performance.
error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
mask	is a character string that specifies a set of files. The mask format is: *dirpath/name.typex.qual:sc:dir:type:size:rl*
diropenname	is the returned character string directory path.
dcblen	is the length of *dirdcb* in words.
msc	is the system security code. If specified, the routine MaskSecurity and FmpMaskName will return the security codes for FMGR files even if the security code was not specified in the original mask.

The *dirdcb* and *diropenname* parameters must not be altered between the FmpInitMask call and the FmpNextMask calls that follow.

The program example at the end of this chapter shows how FmpInitMask, FmpNextMask, FmpMaskName, FmpLastFileName, and FmpEndMask are related and work together.

The fields in the mask qualifier of particular interest to FmpInitMask are *dir*, *dirpath*, and *qual*. Using the *dir* and *dirpath* information, the appropriate directory is opened in preparation for checking entries. If the search qualifier (*qual*) is included, its state is recorded to allow FmpNextMask to perform the search in the correct order. For a complete description of the mask qualifier, see the *RTE-A User's Manual*, part number 92077-90002.

FmpInteractive

FmpInteractive returns a boolean value that reports whether or not the specified DCB is associated with an interactive device.

```
flag = FmpInteractive(dcb)

logical flag
integer*2 dcb(*)
```

where:

flag is a boolean variable that is set to TRUE (negative value) if the specified DCB is associated with an interactive device. *flag* is set to FALSE (non-negative value) if the specified DCB is not associated with an interactive device.

dcb is an integer array containing the DCB for the file.

FmpIoOptions

FmpIoOptions returns the 16-bit I/O option word for the specified DCB.

```
error = FmpIoOptions(dcb,error,options)

integer*2 dcb(*), error, options
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

options is an integer that returns the 16-bit I/O option word.

The upper ten bits of the option word correspond to the upper ten bits of *cntwd* used in EXEC calls. The returned option word is described in the Standard I/O chapter of this manual.

The value returned is undefined if the DCB does not represent a device file.

FmpIoStatus

FmpIoStatus returns the values in the A- and B-Registers after the last I/O request.

```
CALL FmpIoStatus(areg,breg)

integer*2 areg, breg
```

where:

areg is a one-word integer containing the value of the A-Register.

breg is a one-word integer containing the value of the B-Register.

Because it does not specify a DCB, FmpIoStatus returns the values of the A- and B-Registers saved after the last FmpRead or FmpWrite I/O request. The status information in the registers is guaranteed to be accurate only if FmpIoStatus is called immediately after the I/O operation that posted status in the registers.

The value returned is the status and transmission log of a successful request, or a two-word error return for an unsuccessful request. Unsuccessful requests are identified by an error code = −17.

FmpLastFileName

FmpLastFileName extracts the file name from the passed file descriptor.

```
CALL FmpLastFileName(filedescriptor,lastname)

character*(*) filedescriptor, lastname
```

where:

filedescriptor is a character string that specifies the complete file descriptor.

lastname is the file name portion of *filedescriptor*. The file name is identified as the characters between the slash after the directory path (if any) and the first period or colon.

For example, "`FmpLastFileName('SUB/FILE.TXT:::3',last)` " returns "`FILE`".

FmpList

FmpList lists a file to the specified LU.

error = FmpList(*filedescriptor*, *lu*, *option*, *rec1*, *rec2*)

```
character*(*) filedescriptor, option
integer*4 rec1, rec2
integer*2 error, lu
```

where:

error — is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor — is a character string that specifies the name of the file.

lu — is an integer that specifies the output LU.

option — is a character string that selects the output format and options. The values are as follows:

A ASCII output
B Binary output displayed as octal
C File has FORTRAN carriage control characters in column 1
M Count lines longer than 80 characters as multiple lines for page breaking
Q Quiet; do not record access time of file
T Truncate trailing blanks on lines

File types 0, 3, and 4 default to A; all other file types default to B.

rec1 — is a double integer that specifies the first record to be listed.

rec2 — is a double integer that specifies the last record to be listed.

If both *rec1* and *rec2* are set to 0, the entire file is listed.

By default, the listing to an interactive device pauses after printing one page of output. For sessions that have $LINES defined in their Environment Variable Block (EVB), the number of lines per page will be the value of $LINES minus 2. When $LINES is not defined in the EVB, the number of lines per page will be 22.

When FmpList pauses, it prompts you for one of five legal responses. The responses may be preceded by a number from 1 to 32767 called *n*:

a or q	Abort the listing
<space>	List another page
<cr>	List the remainder of the file without pausing
+	List one more line or skip *n* lines and list 1 more
p	Set page size to *n* and list another page
z	Suspend calling program (restart with the system GO command)

For additional information, refer to the FmpPaginator routine in this manual.

If the LU is not interactive, the listing does not pause.

FmpList is limited by buffer constraints to lines up to 256 bytes long. See FmpListX for longer lines.

The Q option is used when the user does not want to have the access time of the file updated. With this option, there is no attempt to update the access time. The Q option is useful when listing a file residing on a write-protected directory. Normally, the file system attempts to update the file access time and because the directory is write protected, the LI command will fail.

The Q option may be combined with either the A or B option; for example: option = `'BQ'`.

FmpListX

FmpListX, the extended version of FmpList, lists a file to the specified file or LU. This version allows the caller to provide a buffer so that lines longer than 256 bytes can be listed. This also allows the listing to be sent to a file, not just an LU.

error = `FmpListX(`*sourcefile*`,` *destfile*`,` *options*`,` *startrec*`,` *endrec*`,` *buffer*`,` *maxlength*`)`

```
character*(*)  sourcefile, destfile, options
integer*2      buffer(*), maxlength, error
integer*4      startrec, endrec
```

where:

sourcefile is the name of the file to be listed.

destfile is the name of the destination listing file.

options is the character string that selects the output format and options. The values are as follows:

A	ASCII output
B	Binary output displayed as octal
C	File has FORTRAN carriage control characters in column 1
M	Count lines longer than 80 characters as multiple line for page breaking
Q	Quiet; do not record access time of file
T	Truncate trailing blanks on line

File types 0, 3, and 4 default to A; all other file types default to B.

startrec is the first record number to be listed.

endrec is the last record number to be listed.

buffer is the buffer for transporting records between *sourcefile* and *destfile*.

maxlength is the maximum number of bytes that may be contained in *buffer*.

If both *startrec* and *endrec* are set to 0, the entire file is listed.

By default, the listing to an interactive device pauses after printing one page of output. For sessions that have $LINES defined in their Environment Variable Block (EVB), the number of lines per page will be the value of $LINES minus 2. When $LINES is not defined in the EVB, the number of lines per page will be 22.

When FmpListX pauses, it prompts you for one of five legal responses. The responses may be preceded by a number from 1 to 32767 called *n*:

a or q	Abort the listing
<space>	List another page
<cr>	List the remainder of the file without pausing
+	List one more line or skip *n* lines and list 1 more
p	Set page size to *n* and list another page
z	Suspend calling program (restart with the system GO command)

For additional information, refer to the FmpPaginator routine in this manual.

If the LU is not interactive, the listing does not pause.

The Q option is used when the user does not want to have the access time of the file updated. With the Q option, there is no attempt to update the access time. The Q option is useful when listing a file residing on a write-protected directory. Normally, the file system would attempt to update the file access time and, because the directory is write-protected, the LI command would fail.

FmpLu

FmpLu returns the LU of the file or device associated with the specified DCB.

```
lu = FmpLu(dcb)

integer*2 dcb(*), lu
```

where:

dcb is an integer array containing the DCB for the file.

lu is an integer indicating the LU number of the file or device associated with the specified DCB.

If the DCB is associated with a type zero file, the value returned in the *lu* parameter is the number of the device LU. If the DCB is associated with a disk file, the value returned is the LU of the disk on which the file resides. If the specified DCB is not open, a −11 (DCB not open error) error is returned.

FmpMakeSLink

FmpMakeSLink creates a symbolic link file with the name specified in *symlink*. The contents of the link is the path specified in *fdesc*. The file named in *symlink* must not already exist.

```
error = FmpMakeSLink(dcb,error,fdesc,symlink)

integer*2 dcb(*), error
character*(*) fdesc, symlink
```

where:

dcb	is a 16-word integer array to contain the DCB of the symbolic link being created.
error	is an integer that returns a negative code if an error occurs.
fdesc	is a character string that contains the path name to be used as the contents of the symbolic link being created.
symlink	is a character string that contains the name of the symbolic link file to be created.

Symbolic links may point to either FMP files, FMP directories, or device LU numbers. When creating a link to a file or directory, the *fdesc* parameter must be in hierarchical format. The symbolic link file being created cannot be a FMGR file.

FmpMaskName

FmpMaskName builds a full file descriptor from the *entry* and *curpath* parameters returned by a call to FmpNextMask.

```
CALL FmpMaskName(dirdcb,newname,entry,curpath)

character*(*) newname, curpath
integer*2 dirdcb(*), entry(32)
```

where:

dirdcb	is a control array, initialized by FmpInitMask.
newname	is a character string that returns the file descriptor.
curpath	is a character string directory path returned by FmpNextMask.
entry	is a 32-word directory entry returned by FmpNextMask.

The file descriptor returned to *newname* includes all of the fields specified by *entry* (name, file type extension, full directory specification, type, size and record length). Null fields are omitted in the file descriptor.

The names generated by FmpMaskName often exceed the 63-character file system limit because the names include the type, size, and at least four colons.

FmpMount

FmpMount mounts a disk volume.

```
error = FmpMount(lu,flag,blks)

integer*2 lu, flag, blks
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

lu is an integer that specifies the system LU of the disk.

flag is an integer that determines whether to initialize the disk before mounting it. The values of *flag* are:

0 Do not initialize before mounting.
1 Initialize if the disk does not have a valid directory.
2 Initialize disk before mounting.

blks is an integer that specifies the number of blocks to leave free at the beginning of the volume. These blocks are never allocated to files or directories; they are used to contain bootable programs such as BOOTEX or an offline utility.

When a volume is mounted, the disk becomes available to the system, global directories can be made available, and the disk space can be used by its owner. An entry is made in the cartridge list to let the system remount the volume automatically after a system shutdown.

It is an error to mount a disk that is already mounted, or to try to mount a non-disk LU.

FmpNextMask

FmpNextMask returns the directory entry for the next file in the directory.

```
more = FmpNextMask(dirdcb, error, curpath, entry)

logical more
integer*2 dirdcb(*), error, entry(32)
character*(*) curpath
```

where:

more	is a boolean variable that indicates whether the search can continue. It is set TRUE (negative value) if there is another entry to be searched, whether or not an error occurred. If it is TRUE and an error has occurred, the current entry is not valid. It is set FALSE (non-negative value) if an error occurred that prevents successful continuation of the current search process.
dirdcb	is a control array, initialized by FmpInitMask.
error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
curpath	is the returned character string directory path.
entry	is a 32-word array that returns the directory entry for each file found.

For recoverable errors, the calling program can determine the response, and terminate or continue the search.

When the search is complete, error returns a 0 and *more* is FALSE.

As the search changes directories, *curpath* is updated to reflect the new path. *curpath* can be used by the calling program when the desired file is found. Errors reported by FmpNextMask are associated with *curpath*; they report errors in accessing the directory in *curpath*.

FmpNextMask tests the program's break flag (IFBRK) and if set, it returns error −235 (Break Detected). Thus, if your program also calls IFBRK, the break flag may have been cleared by FmpNextMask.

FmpEndMask should be called after a mask search terminates. If FmpEndMask is not called, directories may be left open to your program after the search ends.

FmpOpen

FmpOpen opens the named file with the specified options. Files must be opened before any operation that accesses their contents can be performed. Once opened, a file can be accessed until it is closed by FmpClose. When a file is opened, it is positioned to the first word in the file, at record number 1. FmpOpen cannot open FMGR type 0 files.

type = `FmpOpen(`*dcb*`,` *error*`,` *filedescriptor*`,` *options*`,` *buffers*`)`

```
integer*2 dcb(*), error, buffers
character*(*) filedescriptor, options
```

where:

type is a non-negative integer that returns the type of the opened file. If an error occurs, *type* returns a negative error code. Note that when symbolic link files are opened with the L option, *type* returns the value 32767.

dcb is an integer array to contain the DCB for the file. The array must be at least 16 words long to contain file control information. For access to type 0 or 1 files, this minimum size is all that is required. For access to other type files, at least one DCB buffer of 128 words should also be allocated in *dcb*.

error is an integer that returns a negative code if an error occurs or returns the type of open file if the call is successful.

filedescriptor is a character string that specifies the name of the file or the LU number of a device. The device in this case is referred to as a type 0 file even though no real file exists on disk.

options is a character string that selects options for opening the file. The options are selected by the letters in the following list:

Access Mode:

R	Open for reading
W	Open for writing

File Existence:

C	Create a new file
O	Open an existing file

Miscellaneous:

D	File descriptor specifies a directory
E	Force LU locking of interactive devices
F	Force type to 1 for nonbuffered access
I	Inhibit LU locking of non-interactive devices
L	Open symbolic links
N	File does not contain carriage control
Q	Open file quickly, do not record access time
S	Open a shared file
T	File is temporary
U	Open in update mode
X	Access extents in type 1 or 2 file
n	Use UDSP #*n* when searching for the file (*n* = 0,...,8)

The options can be specified in any order, and in uppercase or lowercase characters. Any combination of options is legal, but the options should be grouped by type for readability.

buffers is an integer between 1 and 127 that specifies the size of the DCB buffer, expressed as the number of 128-word buffers in the user array *dcb*, in addition to the 16-word file control information area. The larger the DCB buffer, the faster sequential file accesses can execute. The user array *dcb* must contain at least as many 128-word buffers as the parameter buffers indicates, or the file system may overwrite your program. The entire DCB buffer is used unless it is larger than the size of the accessed file or extent. Type 0 and 1 files (including files forced to type 1) do not use the DCB buffers, so the DCB need only have room for 16 words of file control information.

If the file being opened is on a FMGR cartridge, the file descriptor must be in the file::dir format. Also, a file being created on a FMGR cartridge is always opened exclusively.

FmpOpen updates the time of last access, unless the Q option is selected. FmpOpen sets the time of creation and time of last update for files that it creates.

The DCB specified in the call is closed before it is used for the file to be opened, even if it had last been used for the same file. Re-opening a file (to change the access options, for example) momentarily closes the file.

If the file descriptor specifies an LU number or a symbolic link to an LU number, FmpOpen assigns a DCB to the specified device. The device is referred to as a type 0 file, even though no real file exists on disk.

If the device is opened exclusively, the LU is locked unless the device is interactive. FmpOpen sets flags and option bits in the DCB according to the device type (that is, terminals are opened for read and write access, but line printers are open for write access only). The I/O options can be changed with the FmpSetIoOptions routine. An example of FmpOpen is as follows:

```
type = FmpOpen(dcb,error,d'DATABASE.DB','rwso',8)
```

This call opens the existing file `DATABASE.DB` for shared read and write access, with a DCB buffer 1024 words (8 * 128) in length. The file must exist, because the create option is not selected. Your programs must coordinate shared write access.

Some examples of option combinations are:

To open an existing file for shared read access, specify `'ROS'`.

To create a new file for exclusive write access, specify `'WC'`. The O option can be specified at the same time as the C option for output files to create a new file if the specified file does not exist, or to overwrite an existing file. As a result, the C option should be used only for output files, not for sequential read files, because it can overwrite the file when it opens it. Note that because creating a file implies write access to the file, the W option always must be specified with the C option.

To create a temporary write/read scratch file, specify `'WRCT'`.

The calling program must have access privileges to all files that it tries to open. An error is generated if a program tries to access a file in a way that is not specified by the open request options, such as writing to a file that is opened only for reading. Changing the protection for a file after it is open to one or more programs has no effect on their access to the file.

C Option

The C option creates a file. The W option also must be specified because creating a file implies write access. If you do not specify the W option, error −203 (Did not ask to write) is returned.

FmpOpen can be used to create any type of file. The *filedescriptor* parameter must specify the file name, type, directory, and all other file information. To create a file of type 2, with 200 blocks of records that are 10 words in length, the following file descriptor is used:

```
FILE.DAT::DIRECTORY:2:200:10
```

FmpBuildName or FmpBuildPath can be called to create a file descriptor from a file name and integer file information.

Note If the O and C options are specified and the file already exists, all of the information in the file descriptor after the directory is ignored, the existing file is opened and, for a variable length record file, the EOF mark is placed at the beginning of the file to make the file empty. The type of the existing file is unchanged; it is returned as a function value.

If only the file name and directory are specified, the file system will default to type 3. The default size for FMGR files is 24 blocks. For hierarchical files, the default size is either 24 blocks or 32 blocks depending on the size of the disk volume where the file is being created. For disk volumes greater than 256 Megabytes, the default size is 32 blocks.

Files larger than 32767 (16383 blocks) sectors are created by specifying the size as a negative number of 128-block "chunks". A file of 128000 blocks is specified with a size of −1000. Positive numbers larger than 32767 are meaningless, but do not cause an error.

If a size of −1 is specified when creating a FMGR file, the rest of the space on the FMGR cartridge is used, up to a maximum of 16383 blocks.

D Option

The D option allows the *filedescriptor* parameter to specify a directory rather than a file. It is used by programs that scan directories. Directories are usually read as type 2 files with 32-word records. Directories cannot be opened for write access.

E Option

The E option is used only for device files associated with interactive devices. When specified on exclusive opens, the LU of the interactive device will be locked.

F Option

The F option forces a file to type 1 for nonbuffered access, which ignores record marks. This option does not change the file type or extents of the file. The *type* parameter of FmpOpen returns the correct file type regardless of whether the F option is specified for the file.

Type 1 access is faster because a block of data is transferred directly from the disk to the user buffer (*ibuf*); the DCB buffer is bypassed. The calling program is responsible for calculating record length and accessing entire records.

An error occurs if you specify the F option for a device file.

I Option

The I option inhibits LU locking of non-interactive devices when opened exclusively.

L Option

The L option only applies when opening a symbolic link file. If the L option is specified, the symbolic link file itself is opened. Note that to change the contents of a symbolic link, the symbolic link file should be purged and recreated with the FmpMakeSlink call.

N Option

The N option is used only for device files associated with line printers. If FmpWrite or FmpWriteString are used with the N option specified, the first byte in the record is NOT used for carriage control and will be printed. Without the N option, the first character is assumed to be a carriage control character and it will not be printed.

Q Option

The Q option opens a file quickly, without recording the access time. This is useful when a file is opened repeatedly, which makes the access time unimportant. It is also used when the system time is not set.

S Option

The S option opens a file for shared access. By default, files are opened exclusively; no other program can access the file as long as it is opened exclusively to another program.

If a file is opened for reading only, it should be opened for shared access to allow other programs to read from the file at the same time.

No program can exclusively open a file that is already open for shared access.

T Option

The T option creates temporary files. These files are flagged as temporary files in the directory and should be purged by the calling program when no longer needed.

FMP automatically purges temporary files if a calling program creates and opens exclusively a temporary file, and terminates without closing the temporary file. The temporary file is purged the next time FMP scans its internal file table; for example, FMP scans its internal file table when a program accesses a file for the first time.

Temporary files that are closed by FmpClose are not automatically purged. You can make a temporary file permanent by opening the file without specifying the T option.

You can use the temporary flag to clean up after a system failure by using the masking T option with the PU command (PU @.@.T).

The T option is ignored for FMGR files.

U Option

The U option reads the block containing the record to be updated into the DCB before the record is modified. This prevents existing records in the block from being destroyed.

Update mode is automatically in effect when a type 2 file is opened for write access. The U option must be specified in all other circumstances; for example, modifying a record in the middle of a sequential file.

Update mode is not related to the time of last update found in other FMP routines.

X Option

All file types can be extended to allocate additional disk space when the file becomes full. The X option is not required for sequential files, because they are automatically extended, but it is necessary for random access (type 1, 2, or 6) files, so that they can be extended when the last record of the existing file is filled. Some programs cannot automatically access extents for type 1 and 2 files; the X option allows them to access the extents. Type 6 files are program files, so they should not be extended.

n Option

The number *n* specifies the number of the User-Definable Directory Search Path (UDSP) to be used in searching for the file. *n* can be set to a value from zero to 8, inclusive.

The *n* option is ignored if directory information is included in the file descriptor; FmpOpen searches only the directory specified in the file descriptor.

If the file descriptor does not include directory information, FmpOpen searches each directory in the specified UDSP until the file is found. If the file is not found, a −6 (No such file) error is returned.

If the UDSP specified with the *n* option does not exist, a −247 (UDSP not defined) error is returned.

Refer to the PATH command in the *RTE-A User's Manual*, part number 92077-90002, for more information on UDSPs.

FmpOpenFiles

FmpOpenFiles finds open files in a directory.

```
error = FmpOpenFiles(dcb,error,loc,flag)

integer*2 dcb(*), error, loc, flag
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

loc is an integer that returns the directory position of an open file. The calling program initializes it to zero to indicate that this is the first call. Each time this routine is called, the location and flag value for one file are returned in the *loc* and *flag* parameters.

flag is an integer that returns the ID segment number of the program that opened the file (in bits 0-7) and the exclusive open bit (in bit 15).

The location is returned as a record number in a type 2 file (the directory). *loc* = 1 is the first 32-word entry in the file, the directory header. *flag* contains the ID segment number of the program that opened the file in bits 0-7, and the exclusive open bit in bit 15.

Locations are returned in ascending order. Only one flag is returned per file, so there is no way to tell how many programs are sharing an open file. When all of the open files in the directory have been reported, *loc* is returned as −1.

FmpOpenScratch

FmpOpenScratch is an interface to the FmpOpen routine. FmpOpenScratch standardizes the search path used in the creation of scratch files.

```
type = FmpOpenScratch(dcb,error,filedescriptor,options,buffers,nameused)

integer*2 dcb(*), error, buffers
character*(*) filedescriptor, options, nameused
```

where:

type is a non-negative integer that returns the type of the opened file. If an error occurs, *type* returns a negative error code.

dcb is an integer array to contain the DCB for the file. The array must be at least 16 words long to contain file control information. For access to type 0 or 1 files, this minimum size is all that is required. For access to other type files, at least one DCB buffer of 128 words should also be allocated in *dcb*.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string specifying the name of the file.

options is a character string that selects options for opening the file. Options are the same as the options for FmpOpen with the addition of the following option:

Z Use file name as prefix for FmpUniqueName

The options can be specified in any order, and in uppercase or lowercase characters. Any combination of options is legal, but the options should be grouped by type for readability.

buffers is an integer between 1 and 127 that specifies the size of the DCB buffer, expressed as the number of 128-word buffers in the user array *dcb*, in addition to the 16-word file control information area. The larger the DCB buffer, the faster sequential file accesses can execute. The user array *dcb* must contain at least as many 128-word buffers as the parameter buffers indicates, or the file system may overwrite your program. The entire DCB buffer is used unless it is larger than the size of the accessed file or extent. Type 0 and 1 files (including files forced to type 1) do not use the DCB buffers, so the DCB need only have room for 16 words of file control information.

nameused is a character string that returns the complete file descriptor of the scratch file that was opened. The returned file descriptor includes the full directory path, file type, and file size. Record length in decimal ASCII is also returned for type 2 files. The file size is the total size of the file, including extents. For remote files, the file descriptor includes the user name and remote node number.

If a directory is specified in the *filedescriptor* parameter, then FmpOpenScratch calls FmpOpen using that directory. If no directory is given, FmpOpenScratch calls FmpOpen one or more times using the standard sequence to find a scratch directory. FmpOpenScratch:

1. tries the directory /SCRATCH first. If an error occurs (such as 'no such directory'), then it
2. tries FMGR cartridge specified by entry point $SCRN. This entry point contains a FMGR disk LU defined at bootup to be used as a scratch cartridge. The BOOTEX command, SC, sets the value of $SCRN. If any error occurs (such as 'cartridge full'), then it
3. tries the default directory (' '). FmpOpen then uses either the calling programs working directory or, if there is no working directory, the first available FMGR cartridge.

With the exception of the Z option and the *nameused* parameter, the parameters for FmpOpenScratch are identical to FmpOpen parameters.

The Z option causes the routine to take the file name from the file descriptor given, and use it as a prefix to generate a unique name using the FmpUniqueName routine (refer to the description of this routine documented later in this chapter). For example, if the file descriptor is `'TEST::4:5'`, with the Z option in the options parameter, FmpOpenScratch calls FmpUniqueName with the name "TEST" as the prefix. The unique name that results is used in the FmpOpen call.

FmpOpenScratch calls FmpFileName which builds the actual file descriptor. The file descriptor is returned in the *nameused* parameter. (For details refer to the description of FmpFileName.) Note that FmpOpenScratch uses this parameter to build the file descriptor that it uses in the FmpOpen call; therefore, the size of the variable passed should equal the size of the maximum file descriptor allowed (63 characters).

All parameters except *nameused* are passed by the FmpOpenScratch routine to FmpOpen. The FmpOpen routine returns any values directly to the routine calling FmpOpenScratch. The value of the FmpOpenScratch function is either the file type (if no error occurs), or the error (as returned by FmpOpen). This calling sequence is identical to the FmpOpen calling sequence. Therefore, you should be able to use this routine as a direct replacement for the FmpOpen call in situations where the scratch directory is used.

FmpOpenTemp

FmpOpenTemp interfaces with the FmpOpen routine to open or create a temporary file.

type = FmpOpenTemp(*dcb*, *error*, *name*, *options*, *buffers*)

```
integer*2 type, dcb(*), error, buffers
character*(*) options
```

where:

type is a non-negative integer that returns the type of the opened file. If an error occurs, *type* returns a negative error code.

dcb is an integer array to contain the DCB for the file (see the *dcb* description under the FmpOpen call).

error is an integer that returns a negative code if an error occurs.

name is a character string specifying characters to be included in the file name. The file name is generated by taking this string adding a string of 4 digits made up of the system CPU number and the ID segment number of the program; this number will be unique for each program. The name is constructed based on where the file exists or is to be created, whether it is a FMGR cartridge or a CI volume, as follows:

FMGR the digits appear first, followed by the first two characters of the specified name string.

CI the name string appears first, followed by the string of digits.

The result is a temporary file name on a FMGR cartridge (files whose names start with a leading digit are treated as temporary files), or a temporary file on a CI volume (files created with the T option are treated as temporary). The 4 digits in the file name are unique for the program. If the program is going to create more than one file, the *name* strings specified must be carefully chosen so as to make the files unique.

options is a character string that selects options for opening the file. Options are the same as the options for FmpOpen except that the T option is automatically added if not specified.

buffers is an integer between 1 and 127 that specifies the size of the DCB buffer (see the description under the FmpOpen call).

If a directory is specified along with the file name string, that directory is used for the file. If no directory is specified, a directory or cartridge is chosen as follows:

1. If a scratch cartridge is defined for the system (specified by the FMGR VL or BOOTEX SS command), that cartridge is used.
2. Otherwise, if /SCRATCH exists, that directory is used.
3. Otherwise, if a working directory is defined, that is used.
4. Otherwise, the first available FMGR cartridge with sufficient space is used.

The file name is constructed based on whether the location selected is a FMGR cartridge (1 or 4) or a CI volume (2 or 3).

If the file is created with this call, it is considered temporary, that is, if the program fails to close the file or aborts without closing the file, the file will be purged at a later time. A temporary FMGR file is purged when the file system finds the file while looking through the cartridge directory for some other purpose; a temporary CI file is purged during the periodic consistency check done against CI open flags.

FmpOwner

FmpOwner returns the name of the owner of the specified directory.

```
error = FmpOwner(dir,owner)

character*(*) dir, owner
```

where:

dir is a character string that specifies the name of the directory or the number of the CI volume.

owner is a character string that returns the logon name of the user who owns this directory or volume.

FmpPackSize

FmpPackSize packs the double integer file size into a single word.

```
size = FmpPackSize(doublesize)

integer*2 size
integer*4 doublesize
```

where:

size is an integer that returns the file size in one word.

doublesize is a double integer specifying the file size.

If *doublesize* is less than 16384, there is no change. If *doublesize* is greater than 16383, it is rounded up to the nearest multiple of 128 and divided by 128, and the sign is changed. No overflow check is made. Refer to the FmpExpandSize routine for a description of special considerations for FMGR size parameters.

Because of overflow problems and rounding errors,

```
size = FmpPackSize(FmpExpandSize(size))
```

is an identity for all values of *size*, but

```
doublesize = FmpExpandSize(FmpPackSize(doublesize))
```

is not always an identity.

FmpPagedDevWrite

FmpPagedDevWrite performs an XLUEX(2) write to a device with page breaking for interactive devices. See the FmpPaginator description for more information on page breaking.

```
status = FmpPagedDevWrite(cntwd, buffer, length, pageinfo)

integer*2 status, cntwd(2), buffer(*), length, pageinfo(0:4)
```

where:

cntwd is a two-word XLUEX control word describing the LU (0..255) to be written to.

buffer is an integer array containing the data to be transferred.

length is an integer holding the positive number of words or the negative number of bytes to be transferred from the buffer.

pageinfo is a five-word array holding paging information for FmpPaginator (see the discussion of that routine for more information).

status returns zero (0) if ready for another line, or one (1) if you want to abort the listing.

FmpPagedWrite

FmpPagedWrite writes data to a file of any type if it is opened for write access. FmpPagedWrite is similar to FmpWrite (described in a subsequent section), but it calls FmpPaginator to break the output into screen pages for terminal devices. See the description of FmpPaginator for more information on page breaking.

status = FmpPagedWrite(*dcb*, *error*, *buffer*, *length*, *pageinfo*)

integer*2 *status*, *dcb*(*), *error*, *buffer*(*), *length*, *pageinfo*(0:4)

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

buffer is the name of a word-aligned buffer that contains the data to be transferred.

length is the number of bytes to write; it is interpreted as an unsigned one-word integer from 0 to 65534. For values larger than 32767, set *length* to the desired number of bytes minus 65534.

pageinfo is a five-word array that holds paging information for FmpPaginator (see that routine for details). If the first word is zero, the default values are filled in for each word on the first call.

status is an integer that returns one of the following:

zero (0)	if ready for another line to be sent
one (1)	if you want to abort the listing
negative	FMP error code

FmpPaginator

FmpPaginator prompts for page breaks for the FmpList, FmpPagedWrite, and FmpPagedDevWrite routines. FmpPaginator does not transfer data to be listed; it simply prompts and interprets the response. FmpPaginator assumes that the LU parameter describes a terminal device and is called before a line of text is about to be sent to that device.

status = FmpPaginator(*lu*,*pageinfo*)

integer*2 *status*, *lu*, *pageinfo*(0:4)

where:

lu is an integer containing the LU number (0..255) to prompt to.

pageinfo is a five-word array that holds the following information:

word	usage
0	page size in lines or zero if default values are desired
1	lines to print before page break, or −1 if no paging is desired
2	address of prompt buffer
3	length of prompt buffer in bytes
4	flags: **bit** **meaning (if set)**
	15 current page is not to be printed
	0 use unbuffered I/O

If the first word (page size) is zero, the default values are filled in for each word on the first call:

word	default value
0	$LINES minus 2 if $LINES is defined in the EVB, or 22 if $LINES is not defined in the EVB.
1	same value as word 0
2	address of string "More..."
3	7 characters (length of above string)
4	zero (no special flags set)

status returns one of the following:

0	if it is okay to list the next line
1	if you want to abort the listing
2	if you want to continue the listing but skip this line

FmpPaginator checks word 1 of *pageinfo* to see if paging is enabled. If so, that line count is decremented. If the line count is greater than zero, FmpPaginator returns zero (it is okay to list another line). When the line count reaches zero, the prompt pointed to by words 2 and 3 of *pageinfo* is displayed and your response is read. The responses may be preceded by a number from 1 to 32767, called *n*, in these valid response descriptions:

character	action
<space>	list another page, or another *n* lines, if given
<return>	list the rest of the text without paging
A or Q	abort listing (return 1)
+	list one more line or skip *n* lines and list 1 more
P	set page size to *n* and list another page
Z	suspend calling program (restart with the system GO command)

FmpParseName

FmpParseName parses the specified file descriptor into its component fields. It is similar to FmpParsePath.

```
CALL FmpParseName(filedescriptor,name,typex,sc,dir,type,size,rl,ds)

character*(*) filedescriptor, name, typex, dir, ds
integer*2 sc, type, size, rl
```

where:

filedescriptor	is a 63-character string that specifies the file descriptor to be parsed.
name	is a character string that returns the subdirectories (if any) and the file name. *name* can be up to 63 characters in length.
typex	is a character string that returns the file type extension. *typex* can be up to 4 characters in length.
sc	is an integer that returns the security code.
dir	is a character string that returns the global directory name. *dir* can be up to 16 characters in length.
type	is an integer that returns the FMP file type.
size	is an integer that returns the file size in blocks.
rl	is an integer that returns the record length.
ds	is a character string that returns the DS node name, user account name, or both. *ds* can be up to 63 characters in length. Refer to the DS File Access section of the *RTE-A User's Manual*, part number 92077-90002, for a description of the DS node name and user account name.

FmpParseName should be used to upgrade programs designed to manipulate FMGR files to RTE-A, or in new programs when the hierarchical and file masking features of FmpParsePath are not required. The differences between FmpParseName and FmpParsePath are described in the FmpParsePath section of this chapter.

FmpParseName converts the character string input fields of the *filedescriptor* parameter into integers when necessary, as for the type and size fields. When characters appear in numeric fields, they are returned as packed ASCII. For example, if the security code in the *filedescriptor* parameter is "DH", the returned *sc* parameter is 17480. Character fields are returned just as they appear in *filedescriptor*. Numeric fields omitted in the *filedescriptor* parameter are returned as zeroes; omitted character fields are returned as blanks. No error checking is made on *filedescriptor* or the returned parameters.

For example, assume that `fdesc = SANJOSE.TXT::CITIES:4:24`.

```
CALL FmpParseName(fdesc,file,ext,sc,dir,type,size,reclen,ds)

file = SANJOSE, ext = TXT, sc = 0, dir = CITIES, type = 4,
size = 24, reclen = 0, and ds = blank.
```

FmpParseName is not designed to parse file descriptors with hierarchical directory paths (that is the function of FmpParsePath), but it can parse them, with the following limitations.

When a leading directory and subdirectories are specified, the directory name is returned to *dir*, and the rest of the directory path and file name is returned in the name parameter. For example:

If `fdesc = /CITIES/CALIFORNIA/SANJOSE.TXT:::4:24`

```
CALL FmpParseName(fdesc,name,ext,sc,dir,type,size,reclen,ds)
```

`name = CALIFORNIA/SANJOSE, ext = TXT, sc = 0, dir = CITIES, type = 4, size = 24, reclen = 0, and ds = " "`

FmpParsePath

FmpParsePath parses the specified file descriptor into its component fields. It is similar to FmpParseName, except that it parses hierarchical directory paths in a way that is more convenient for you to use programmatically, and parses file descriptors that contain a mask qualifier field.

`CALL FmpParsePath(`*filedescriptor*`,` *dirpath*`,` *name*`,` *typex*`,` *qual*`,` *sc*`,` *type*`,` *size*`,` *rl*`,` *ds*`)`

```
character*(*) filedescriptor, dirpath, name, typex, qual, ds
integer*2 sc, type, size, rl
```

where:

filedescriptor	is a 63-character string that specifies the file descriptor to be parsed.
dirpath	is a character string that returns the hierarchical directory path. *dirpath* can be a maximum of 63 characters.
name	is a character string that returns the file name. *name* can be a maximum of 16 characters. *name* does not return any part of the hierarchical directory information.
typex	is a character string that returns the file type extension. *typex* can be a maximum of 4 characters.
qual	is a character string mask qualifier. *qual* can be a maximum of 40 characters.
sc	is an integer that returns the security code.
type	is an integer that returns the FMP file type.
size	is an integer that returns file size in blocks.
rl	is an integer that returns the record length.
ds	is a character string that returns the DS node name, user account name, or both. DS can be a maximum of 63 characters. Refer to the DS File Access section of the *RTE-A User's Manual*, part number 92077-90002, for a description of the DS node name and user account name.

FmpParsePath should be used when writing new programs that will use the hierarchical file system features, and must be used if file masking is required. Refer to the *RTE-A User's Manual* and to the FMP mask routines described in this chapter for more information about file masking.

The hierarchical directory path (returned in dirpath) is defined as everything that appears to the left of the first character of the file name. All of the directory information in the *filedescriptor* parameter is combined and returned in dirpath. If *filedescriptor* uses the trailing directory notation, as in FILE::GLB, FmpParsePath converts *filedescriptor* to the leading (hierarchical) notation, as in /GLB/FILE, and returns the directory path in dirpath.

Qual permits FmpParsePath to correctly parse file descriptors that contain masks. Mask qualifiers are described in the *RTE-A User's Manual*.

FmpParsePath differs from FmpParseName in two main ways:

- FmpParsePath parses file descriptors with file masks as well as regular file names, and includes the *qual* parameter to return the mask qualifier field.
- FmpParsePath parses hierarchical directory path information in a more convenient way for you to use programmatically. All of the directory information in the *filedescriptor* parameter is returned in *dirpath*, never in the *name* parameter as with FmpParseName.

The following examples illustrate these differences:

Input	**FmpParsePath Output**			**FmpParseName Output**		
filedescriptor	***dirpath***	***name***	***typex***	***dir***	***name***	***typex***
/GLB/SUB/FILE.FTN	/GLB/SUB/	FILE	FTN	GLB	SUB/FILE	FTN
SUB/FILE.FTN::GLB	/GLB/SUB/	FILE	FTN	GLB	SUB/FILE	FTN
/GLB/SUB.DIR	/GLB/	SUB	DIR	GLB	SUB	DIR
/GLB.DIR	/	GLB	DIR	GLB	blank	blank
/GLB/	/GLB	blank	blank	GLB	blank	blank
::GLB	/GLB/	blank	blank	GLB	blank	blank
S1/S2/FILE.REL	S1/S2/	FILE	REL	blank	S1/S2/FILE	REL
FILE.REL	blank	FILE	REL	blank	FILE	REL

The following is an example of how FmpParsePath parses a full file descriptor:

```
Filedesc  =  CALIFORNIA/SANJOSE.TXT.T:23:CITIES:2:24:32[PLANNER]>SYS3

CALL FmpParsePath(filedesc,path,name,extn,qual,sc,type,size,rl,ds)

Path = /CITIES/CALIFORNIA/
name = SANJOSE
extn = TXT
qual = T
sc = 23
type = 2
size = 24
rl = 32
ds = [PLANNER]>SYS3.
```

FmpPosition

FmpPosition returns the current record number and reports the internal file position in a format that can be used later by FmpSetPosition.

```
error = FmpPosition(dcb,error,record,position)

integer*2 dcb(*), error
integer*4 record, position
```

where:

record	is a double integer that returns the current record number.
dcb	is an integer array containing the DCB for the file.
error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
position	is a double integer that returns the current internal file position.

Refer to the FmpSetPosition section of this chapter for a description of how the current record and internal file position are used to change the file position.

Each record in a file is numbered. The first is number one, and the others are numbered consecutively. As the file is read or as information is written to it, the current position is incremented. It is also changed by the FmpSetPosition and FmpRewind routines.

For fixed record length files, the function ((record number −1) * (record size)) indicates the internal file position. The current record position does not identify an exact byte location in variable record length files.

The internal file position specifies the current word offset from the first word of the file. The first word of a file is position zero. The internal position does not depend on actual disk location of the file, so positions can be used even after a file is moved or copied. This value is meaningless for device files.

FmpPosition along with FmpSetPosition can be used to manipulate or to move around in a file in a manner other than sequentially.

FmpPost

FmpPost posts the data in the DCB buffer into the disk file if the data has been changed. Other programs can then access the information by reading the disk file. FmpPost is also used to back up the DCB buffer into the disk file in case the program is aborted. When the DCB buffer is posted, the data in the buffer is invalidated, so the next read call reads the disk file, not the DCB buffer.

```
error = FmpPost(dcb,error)

integer*2 dcb(*), error
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

FmpPost is used to coordinate shared write access to a file. Resource numbers are often used with FmpPost to coordinate the sharing of write access. Refer to the RNRQ section of the Resource Management chapter for more information about resource numbering. Each of a group of cooperating programs that accesses the shared file should perform the following sequence:

1. Lock the file's resource number
2. Access the file
3. Call FmpPost to post the data in the disk file
4. Unlock the resource number

FmpPostEof

FmpPostEof posts the EOF position of a file and the number of records from the DCB to the directory entry for that file.

```
error = FmpPostEof(dcb,error)

integer*2 dcb(*), error
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs.

The file must be a variable length file that is currently being written to and positioned at EOF. This routine does not post the data buffer in the DCB to the disk file. The FmpPost routine should be used for that purpose.

FmpProtection

FmpProtection returns the access rights of the owner and others to the specified file or directory.

error = FmpProtection(*filedescriptor*, *owneraccess*, *othersaccess*[, *groupaccess*])

character*(*) *filedescriptor*, *owneraccess*, *othersaccess*, *groupaccess*

where:

error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
filedescriptor	is a character string specifying the name of the file or the number of the CI volume.
owneraccess	is a character string that returns the access rights of the owner of the file/directory/volume.
othersaccess	is a character string that returns the access rights of all other users of the file/directory/volume.
groupaccess	is a character string that returns the access rights of members of the owner's group to the file/directory/volume.

The access rights are returned as ASCII "R" for read access, "W" for write access, or "RW" for both. "N" is returned when read and write access is denied.

The owner of a directory or of a volume is the user who creates it or is assigned ownership via the FmpSetOwner routine. The owner of a directory owns all of the files within it.

FmpPurge

FmpPurge purges the file specified by the file descriptor, marking the directory entry as purged, to free disk space allocated to the file. The file must exist, must not be open, and must not be an RP'd program. The calling program must have write access to the directory, but not necessarily write access to the file.

error = FmpPurge(*filedescriptor*)

character*(*) *filedescriptor*

where:

error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
filedescriptor	is a character string specifying the name of the file.

The file descriptor can specify a directory by specifying it as ::NAME or SUB.DIR (note the .DIR file type extension). If the directory contains anything other than purged files, it cannot be purged. Purged files can be unpurged with the FmpUnPurge routine, unless their disk space or directory is overwritten.

FmpRawMove

FmpRawMove reads or writes data to a disk file starting at a specified internal file position.

length = `FmpRawMove(`*dcb*`,`*error*`,`*position*`,`*buffer*`,`*maxlength*`,`*how*`)`

where:

length is an integer that returns the number of words successfully transferred to or from the disk file.

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

position is a double integer specifying the desired internal file position.

buffer is a word-aligned integer buffer that either contains the data to be transferred (*how* = 2) or returns the data being transferred (*how* = 1).

maxlength is an integer that contains the number of words to be transferred.

how is an integer that specifies the direction of the transfer.

1 Read data from the file into *buffer*.
2 Write data from *buffer* into the file.

The internal file position after the call is undefined. It is the caller's responsibility to reset the internal file position after the call.

FmpRead

FmpRead reads data from a file of any type. FmpRead reads the record at the current file position. The file positioning routines described in this chapter explain how to change the current file position. The file must be opened for read access before FmpRead is called.

length = `FmpRead(`*dcb*`,`*error*`,`*buffer*`,`*maxlength*`)`

`integer*2` *dcb*`(*),` *error*, *buffer*`(*),` *maxlength*

where:

length is an integer that returns the number of bytes actually read, or a negative error code. If the call reads more than 32767 bytes, the return length may be negative even though no error occurs; in such cases *error* should be compared to the length return. If they match, an error has probably occurred.

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

buffer is an integer array that returns the data being transferred. The buffer is word-aligned.

maxlength is a one-word integer that contains the maximum number of bytes to transfer. The *maxlength* parameter is treated as an unsigned single integer from 0 to 65534. Values larger than 32767 are expressed as negative numbers equal to the number of bytes to be transferred minus 65536; for example, 40000 bytes is expressed as −25536 (40000 − 65536 = −25536).

If an odd number of bytes are transferred, the lower byte of the word containing the last byte is undefined. The requested transfer length can be longer or shorter than the actual length of the record, but the number of bytes read never exceeds *maxlength*.

The file position is set to the beginning of the next record even if some of the data that was read does not fit into the user buffer.

For sequential files (type 3 and above), one variable-length record is transferred from the current file position. The DCB buffer is used during the transfer. The record length is maintained with the record; if for some reason the record length information is invalid, error −5 is returned. When end-of-file is reached, the returned length is −1; an error is not returned. If your program attempts to read past the end-of-file, error −12 is returned (the returned length is −12).

For type 2 files, one fixed-length record is transferred, using the file record length, which is always an even number of bytes. The DCB buffer is used during the transfer. There is no end-of-file mark; if a program tries to read past the end-of-file, the actual length of the record is returned, and no error is indicated, but subsequent reads will report an error.

For type 1 files (or files forced to type 1), multiple records may be read, depending on *maxlength*. The data is read directly into the user buffer, without using the DCB buffer. Type 1 files are always positioned at a block boundary, so they behave like files with 128-word records. Type 1 files behave like type 2 files when the end-of-file mark is encountered.

For type 0 (device) files, one record is read. The data is read directly into the user buffer, without using the DCB buffer. End-of-file is set if the end-of-file or end-of-medium bits are set in the returned status following the read. The returned length is −1. The control-D character is the end-of-file mark for reads from a terminal; zero-length reads are not treated as the end-of-file. No more than 32767 bytes can be read from type 0 (device) files.

FmpReadLink

FmpReadLink opens a symbolic link file, returns the contents of the file, and closes the file.

```
error = FmpReadLink(dcb,error,symlink,fdesc)

integer*2 dcb(*), error
character*(*) symlink, fdesc
logical error
```

where:

error	is an integer that returns a negative code if an error occurs.
dcb	is a 16-word integer array containing the DCB of the symbolic link file being accessed.
symlink	is a character string containing the name of the symbolic link file.
fdesc	is a character string that returns the contents of the symbolic link file named in *symlink*.

FmpReadString

FmpReadString is an integer function that allows reading character from a file.

```
length = FmpReadString(dcb,error,string)

integer*2 length, dcb(*), error
character*(*) string
```

where:

length	is an integer that returns the positive number of bytes transferred, or a negative error code. *length* cannot be more than 256 because the data must pass through an internal buffer that is 256 bytes.
dcb	is an integer array containing the DCB for the file.
error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
string	is a character string of up to 256 bytes into which data is transferred. The string cannot be more than 256 bytes because the data passes through an internal buffer that is 256 bytes. If *string* is longer than 256 bytes, an error code is returned in the *error* parameter.

FmpReadString is similar to FmpRead, except the data is returned in the *string* parameter. The returned length is the length of the record read; it may be less than the actual length of the *string* parameter, but never more. The string is filled with blanks if the record is shorter than the string.

FmpRecordCount

FmpRecordCount returns the number of records in the specified file.

error = `FmpRecordCount` (*filedescriptor*, *nrecords* [, *slink*])

```
character*(*) filedescriptor
integer*4 nrecords
logical slink
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string specifying the name of the file.

nrecords is a double integer that returns the number of records in the file.

slink is an optional boolean variable that indicates whether to return the number of records in a symbolic link file or the file that it references. The possible values are as follows:

TRUE (negative value)
Return the number of records in the symbolic link file.

FALSE (non-negative value)
Return the number of records in the file referenced by the symbolic link (this is the default).

For type 1 and 2 files, FmpRecordCount returns the maximum number of records that can fit in the file, not the actual number of records currently in the file. For type 3 files and above, *nrecords* is the number of records before the end-of-file; however, if the file is currently open for writing, *nrecords* may not reflect the actual record count because write requests that have not been posted may not be present in the file.

FmpRecordLen

FmpRecordLen returns the length of the longest record in a file.

```
error = FmpRecordLen(filedescriptor, len [, slink])

character*(*) filedescriptor
integer*2 len
logical slink
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string specifying the name of the file.

len is an integer that returns the length of the longest record in the file.

slink is an optional boolean variable that indicates whether to return the length of the longest record in a symbolic link file or the file that it references. The possible values are as follows:

TRUE (negative value)
Return the length of the longest record in the symbolic link file.

FALSE (non-negative value)
Return the length of the longest record in the file referenced by the symbolic link (this is the default).

For a type 1 or 2 file, FmpRecordLen returns the fixed record length in words, that was defined when the file was created. For type 3 files and above, it returns the length, in words, of the longest variable-length records in the file.

Note The length returned for type 3 or above files is actually the length of the longest record ever written to the file, even if that longest record has been overwritten.

FmpRename

FmpRename changes the name of the specified file.

```
error = FmpRename(name1,err1,name2,err2)

integer*2 err1, err2
character*(*) name1, name2
```

where:

error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
name1	is a character string specifying the name of the existing file. The file must be closed.
err1	is an integer that returns any error associated with *name1*.
name2	is a character string specifying the new name for the file.
err2	is an integer that returns any error associated with *name2*.

The file specified by *name1* must exist, and must not be open. It may, however, be an active program. *name2* must not already exist in the directory.

The calling program must have write access to the directory containing the file to be renamed, and to the directory that will contain the file after the rename, if it is not the same as the original directory.

FmpRename can change any combination of the file name, its file type extension, or directory. The security code, type, size, and record length cannot be changed. If they are specified in *name2*, they are ignored. The new file name (*name2*) must specify the desired security code and directory; they cannot be defaulted to match the security and directory of *name1*.

If the directory name is changed, the file directory entry is moved to the new directory, but the actual file data is not moved. The new directory must be on the same LU as the original. *name1* and *name2* can specify directories as either ::NAME or /NAME.DIR (note the .DIR file type extension). It is possible to convert subdirectories into global directories, or vice versa. If the working directory is renamed, it remains the working directory, but under the new name. *err1* returns errors associated with *name1* and *err2* returns errors associated with *name2*. If either *err1* or *err2* contains an error code, the same error code is returned in *error*. If *error* = 0, then neither *err1* nor *err2* contains an error code.

FmpReportError

FmpReportError prints an error message at your terminal (LU 1).

```
CALL FmpReportError(error,filedescriptor)

character*(*) filedescriptor
integer*2 error
```

where:

error is an integer that specifies the error code whose message is to be written to your terminal.

filedescriptor is a character string that specifies the name of the file.

The printed message consists of the message returned by FmpError, followed by the passed file name; for example:

```
No such file FILE.EXT::USER
```

If it is necessary to print the message somewhere other than on LU 1, you should use FmpError to retrieve the error text and write the message to the desired file or device.

Note FmpReportError uses an EXEC call with the no-suspend bit cleared; therefore, FmpReportError suspends your program if your terminal is down or has an LU lock on it. If you do not want your program suspended, use FmpError and perform your own I/O error processing.

FmpRewind

FmpRewind positions the file specified by the DCB to the first word in the file. For disk files this is equivalent to an FmpSetPosition call with position set to zero. For device files, a rewind control call is issued.

```
error = FmpRewind(dcb,error)
integer*2 dcb(*), error
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

FmpRpProgram

FmpRpProgram restores a program from a type 6 file, creating a program or prototype ID segment for the program in the operating system.

error = FmpRpProgram(*filedescriptor*, *rpname*, *options*, *error*)

```
character*(*) filedescriptor, rpname, options
integer*2 error
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string that specifies the name of the type 6 file.

rpname is a character string that either specifies the program name or returns it: if *rpname* is specified, the specified name is used; if *rpname* is blank, the name assigned by the system is returned. The returned name is the first five characters of *filedescriptor* (minus the directory path and file type extension). Note that the string must be initialized to blanks if a program name is not specified. Refer to the Character Strings section of this chapter for details.

options A character string that contains "C", "D", "P", or both "C" and "P" to select either of the following options:

C (clone) Create a clone name if the specified or assigned name already is assigned to an RP'd program. The program is not cloned if:

- *rpname* is not blank and there is already a program with the specified name RP'd in your session or in the system session. Error −239 is returned in this case.
- there is already a "system utility" program with the assigned or specified name RP'd in any session, including the system session. A system utility is a program loaded using the LINK SU command to inhibit cloning. Either error −239 or −251 is returned in this case.
- *filedescriptor* does not include a directory path and there is already a program with the assigned name permanently RP'd and dormant in your session or the system session. Error −239 is returned in this case.

D (duplicate) Create a prototype ID segment, not a program ID segment, for the program file.

P (permanent) Do not release the ID segment when the RP'd program completes.

If FmpRpProgram needs to clone, it will replace the fourth and fifth characters of the program name with ".A". If that name is also taken, it will use ".B", and so forth.

If the RPL checksum of the type 6 file does not match the system, the checksum of the file is changed, and the program is RP'd, but FmpRpProgram returns error −240 (RPL checksum changed). This error is a warning. The program performing the FmpRpProgram call may choose to issue a warning if this error is returned. This is considered good programming practice because the error may indicate that the type 6 file was linked for another system with an incompatible set of microcode RPLs. In such a situation, the RP'd program will likely incur a UI (Unimplemented Instruction) violation when it executes a microcode instruction not present on the current host. In any case, the program issuing the FmpRpProgram call may safely ignore the −240 error.

A program may use FmpRpProgram to create a temporary ID segment for a type 6 file. In this case, the RP'd program should be scheduled with an EXEC call, not with an FmpRunProgram call nor by an operator entering an RU command from the console. Both FmpRunProgram and the RU command use FmpRpProgram to create an ID segment. FmpRpProgram will not use the temporary ID segment created by the previous FmpRpProgram call. Instead, FmpRpProgram will create a new ID segment, and the original 'temporary' ID segment will not be purged when the program completes. If, on the other hand, an EXEC call is used, the temporary ID segment will be used. When the program completes, the temporary bit in the ID segment is checked and the ID segment will be purged.

The process by which FmpRpProgram determines the program to restore is the same as that used by the CI RP command. See the *RTE-A User's Manual*, part number 92077-90002, for a description of the RP command.

FmpRunProgram

FmpRunProgram executes a program.

```
error = FmpRunProgram(string,prams,runname[,alterstring])

character*(*) string, runname
integer*2 error, prams(5)
logical alterstring
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

string is a character string that specifies a runstring. If the string does not begin with RU or XQ, FmpRunProgram inserts RU so the program can correctly parse the runstring. If XQ is specified, the program is executed without wait.

prams is an integer array that returns the RMPAR parameters from the program when it completes. If *string* specifies XQ, *prams* is meaningless.

runname is a character string that returns the true name used to schedule the program.

alterstring is an optional boolean variable indicating how FmpRunProgram is to handle the *string* parameter. The possible values are as follows:

TRUE (negative value)
The string is converted to uppercase and each group of one or more consecutive blanks is converted to a comma (this is the default).

FALSE (non-negative value)
The string is not altered.

If a program with the same name and session ID already exists then an attempt is made to create a clone name by replacing the last two characters with ".A". If that fails, ".B" is tried and so on. If ":IH" follows the program name (for example, RU,PROG:IH), cloning is inhibited.

The process by which FmpRunProgram determines the program to schedule is the same as that used by the CI RU command. See the *RTE-A User's Manual*, part number 92077-90002, for a description of the RU command.

FmpRwBits

FmpRwBits is an integer function that determines whether the returned string of the FmpProtection routine indicates read or write access availability, and whether an options list for FmpOpen contains read or write access requests.

```
rwbits = FmpRwBits(string)

character*(*) string
```

where:

rwbits is an integer that indicates read or write access availability for the string returned by FmpProtection, and read or write access requests for the options list of FmpOpen. FmpRwBits returns one of four values, depending upon whether or not the *string* parameter contains the uppercase letters R or W. The values for *rwbits* are as follows:

0	Neither W nor R present
1	W but not R present
2	R but not W present
3	R and W present

string is a character string. *string* can be a maximum of 256 bytes.

In the *string* parameter, the R and W can be in any order and other characters can be present.

FmpSetDcbInfo

FmpSetDcbInfo changes information in the DCB.

```
error = FmpSetDcbInfo(dcb, error, records, eofpos, reclen)

integer*2 dcb(*), error, reclen
integer*4 records, eofpos
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

records is a double integer that specifies the number of records in the file plus 1.

eofpos is a double integer that specifies the current internal file position.

reclen is an integer that specifies the length, in words, of the longest record.

FmpSetDcbInfo should be called only when a file of type 3 or above that has been forced to type 1 in the FmpOpen call is copied. The DCB for the copied file contains information for a type 1, rather than a type 3 file. FmpSetDcbInfo can be used to change the DCB information to reflect the fact that the file is really of type 3 or above. The call should be used with care, and only by users with a detailed knowledge of DCB information.

The *records* and *eofpos* parameters correspond to the current record and internal file position parameters of the FmpSetPosition routine.

Do not read or write any more data from the DCB after using this routine; call FmpClose to close the DCB, then FmpOpen to re-open it for further access.

FmpSetDirInfo

FmpSetDirInfo changes file directory information.

error = FmpSetDirInfo(*dcb*, *error*, *ctime*, *atime*, *utime*, *bbit*, *prot*[, *option*])

```
integer*2 dcb(*), err, bbit, prot
integer*4 ctime, atime, utime
character*(*) option
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

ctime is a double integer specifying the create time.

atime is a double integer specifying the access time.

utime is a double integer specifying the update time.

bbit is an integer specifying the backup bit.

prot is an integer specifying the new protection for the file, where:

Bit 0 = 1 general user may write
Bit 1 = 1 general user may read
Bit 2 = 1 owner may write
Bit 3 = 1 owner may read
Bit 6 = group may write only if G specified in option string
Bit 7 = group may read only if G specified in option string

Any bit set to zero denies the permission associated with that bit.

option is an optional string that determines the interpretation of the *prot* parameter.

G = *prot* contains valid group bits. If G is not specified, or if the parameter is not present, the group bits (bits 6 and 7) of the *prot* parameter are ignored. In this case, the general user bits (bits 0 and 1) are used for group bits.

The calling program can change the create, access, and update time stamps, set or reset the backup bit, and change the file protection.

If a supplied parameter is negative, the corresponding value in the directory entry is not changed.

If the calling program owns the file, it also can set the file protection to the lower 4 bits of *prot*. *prot* is ignored if the calling program is not the owner.

Do not read or write any more data from the DCB after using this routine.

FmpSetDirInfo should be called after FmpSetDcbInfo if both are to be called.

FmpSetEof

FmpSetEof sets the end-of-file to the current position in a sequential file, or issues an end-of-file control request for a device file. It has no effect on type 1 and 2 files.

```
error = FmpSetEof(dcb,error)

integer*2 dcb(*), error
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

FmpSetEof is not required in normal operation because the end-of-file is set automatically following writes to sequential files that are not opened in the update mode. It should be used only to reset the end-of-file mark in files opened in the update mode, and for writing to device files that require an explicit end-of-file control request, such as magnetic tapes. It does not remove any other EOF marks in the file, so it cannot be used to expand a file; it can be used only to make the file smaller.

FmpSetIoOptions

FmpSetIoOptions changes the I/O option word for the specified DCB.

```
error = FmpSetIoOptions(dcb,error,options)

integer*2 dcb(*), error, options
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

options is an integer that returns the 16-bit I/O options word.

Once changed, the new options remain in effect until another FmpSetIoOptions call (or an FmpOpen call). The options word is described in the Standard I/O chapter of this manual. All of the options except the Z-bit can be set, because the FmpSetIoOptions call does not permit a Z buffer to be sent.

The call is ignored if the DCB is not open to a device file. FmpSetIoOptions should not be called under normal operation; in most cases, you should allow the file system to set the I/O option word.

FmpSetOwner

FmpSetOwner changes the owner of a directory or CI volume to the specified user. You must be the current owner or a superuser.

```
error = FmpSetOwner(dir,err1,owner,err2)

character*(*) dir, owner
integer*2 err1, err2
```

where:

dir is a character string that specifies the name of the directory or the number of the CI volume whose owner is being changed.

err1 is an integer that returns errors associated with *dir*.

owner is a character string that specifies the name of new owner of the directory.

err2 is an integer that returns errors associated with *owner*.

If either *err1* or *err2* contains an error code, the same code is returned in *error*. If *error* = 0, then neither *err1* nor *err2* contains an error code.

FmpSetPosition

FmpSetPosition sets or changes the current file position. The position can be set either to a record number or to an internal file position.

```
error = FmpSetPosition(dcb,error,record,position)

integer*2 dcb(*), error
integer*4 record, position
```

where:

error is an integer that returns a negative code if an error occurs or non-negative if no error occurs.

dcb is an integer array containing the DCB for the file.

record is a double integer that specifies the desired record number.

position is a double integer that specifies the desired internal file position.

All files can be positioned to a particular record number. All disk files can be positioned to an internal file position as returned by FmpPosition. For fixed record length files, the record number and internal file positions are related by the function ((record_number−1) * record_size). For sequential files there is no such correlation because the records are variable in length.

Positioning sequential and device files by record number is very slow because it requires starting at the first record and stepping through to the desired record. Positioning by internal position is

much faster for sequential files, but the position must be at the start of a record because read and write calls depend upon being at the beginning of a record. FmpPosition can be called to return the position of the start of a record to pass it to FmpSetPosition.

If the *position* parameter is positive, FmpSetPosition interprets it as the desired internal file position. The passed record number is saved as the current record number for later use, with the exception of type 1 and 2 files where the record number is always forced to represent the position according to the function mentioned above. Be aware that if the record number is not accurate to the true position, then upon closing the file, the directory entry will contain the same inaccuracy.

If the *position* parameter is negative, positioning occurs by record. Device files are always positioned by record number only, regardless of the internal position value. Double integer variables should be used for the record number and internal position for device files because they are often large numbers.

Although FmpSetPosition is usually called to position a file to a location already in the file, it can be used to create extents in a file opened for writing. Positioning a type 1 or 2 file can create an extent, but it can create a sparse file, which has missing extents between the file and a full extent. If a read request tries to access a record in one of the missing extents, an error occurs. Positioning a file of type 3 or above creates an extent without skipping extents, even if the file is forced to type 1 by the F option in the FmpOpen call.

FmpSetProtection

FmpSetProtection allows the owner of a file, directory, or CI volume to change the access rights to the file or directory.

error = `FmpSetProtection(`*filedescriptor*`,` *owneraccess*`,` *othersaccess*`[,` *groupaccess*`])`

`character*(*)` *filedescriptor*`,` *owneraccess*`,` *othersaccess*`,` *groupaccess*

where:

filedescriptor	specifies the name of the file or the CI volume number.
owneraccess	specifies the access rights of the owner of the file/directory/volume.
othersaccess	specifies the access rights of other users of the file/directory/volume.
groupaccess	is an optional character string specifying the access rights of members of the owner's group to the file/directory/volume.

The access rights are specified as ASCII "R" for read access, "W" for write access, or "RW" for both. The suggested setting is "RW" for owner, "R" for others.

When the access rights to a directory are changed, the access rights to files or subdirectories already in it are not changed, but new files or subdirectories created in it receive the new access rights. If the *groupaccess* parameter is not specified, the group access rights will not be changed.

The owner of a directory is the user who creates it or is assigned ownership via the FmpSetOwner routine. The owner of a directory owns all the files in it.

To prevent owners from being locked out of their own directories, owners do not need write access to a directory to change its protection. A superuser can change protection on any file or directory. A file's protection status can be changed while it is open, because protection status is only checked when the file is opened. Files that already have the file open are not affected by the protection change.

FmpSetWord

FmpSetWord positions a disk file to a specified internal position in the file.

```
error = FmpSetWord(dcb,error,position,how)

integer*2 dcb(*), error, how
integer*4 position
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

position is a double integer specifying the desired internal file position.

how is an integer that specifies whether the file system should create an extent to contain the new position if it is outside the existing file area. *how* can be set to the following values:

1 Extent creation is not permitted; the usual setting for read operations that must only access existing file areas.

2 Extent creation is permitted.

FmpSetWord is a special case of the FmpSetPosition routine, and should be used only to minimize code size. FmpSetPosition is the general purpose positioning routine, and uses more code space.

FmpSetWord works exactly as FmpSetPosition does when it is called to position a file by internal file position, rather than by record. FmpSetWord does not update the record number in the DCB, so once it has been called, positioning by records must not be attempted. It also does not record the end-of-file position when a position beyond the existing end-of-file is selected without extent creation enabled, nor does it reset the end-of-file condition if a position before the end-of-file is selected. Its only advantage is that it does not add to the code size of the calling program, because it is used by FmpRead and FmpWrite, so it is already part of the code.

FmpSetWorkingDir

FmpSetWorkingDir changes or sets the working directory for you. The working directory can be a global directory or a subdirectory. Setting the working directory changes the working directory for all programs in the current session. It should be used with caution.

```
error = FmpSetWorkingDir(directory)

character*(*) directory
integer*2 error
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

directory is a character string that specifies the working directory.

If the directory is specified as the character string `'0'` (zero), then you have no working directory until another call is made to establish one. This is useful in changing the search behavior for files when no directory is specified. If there is no working directory, the FMP calls can search FMGR disks for a specified file.

If the directory name is longer than 63 characters, error −15 is returned.

FmpShortName

FmpShortName returns the file descriptor for the file associated with the specified DCB.

```
error = FmpShortName(dcb,error,filedescriptor)

character*(*) filedescriptor
integer*2 dcb(*), error
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string that returns the name of the file.

The returned file descriptor is not a full file descriptor; it does not include the file type, size, or record length. FmpShortName is similar to FmpFileName, described in this chapter, except that it returns a truncated file descriptor.

FmpSize

FmpSize returns the physical size of the file in blocks.

error = FmpSize(*filedescriptor*, *size*[, *slink*])

```
character*(*) filedescriptor
integer*4 size
logical slink
```

where:

filedescriptor is a character string specifying the name of the file.

size is a double integer that returns the physical size of the file in blocks.

slink is an optional boolean variable that indicates whether to return the physical size of the symbolic link file or the file that it references. The possible values are as follows:

TRUE (negative value)
Return the physical size of the symbolic link file.

FALSE (non-negative value)
Return the physical size of the file referenced by the symbolic link (this is the default).

The physical size of a file is the number of blocks of disk space it occupies, including extents.

FmpStandardName

FmpStandardName converts a file descriptor to the standard format.

error = FmpStandardName(*filedescriptor*)

```
character*(*) filedescriptor
```

where:

filedescriptor is a character string that specifies the name of the file.

error is an integer error return. The only possible error is −231 (string too long) which is returned if the string will not fit in the file descriptor.

The standard format uses the trailing directory notation, as in FILE.FTN::DIR. If the specified file descriptor includes subdirectories, it uses the hierarchical format, with a leading directory path, as in /DIR/SUB/FILE.FTN. If the file descriptor refers to a global directory, it also uses the hierarchical format, as in /GLB.DIR.

The standard is convenient for users familiar with FMGR files because the "::" notation is used whenever the file descriptor does not include a hierarchical directory structure.

FmpTruncate

FmpTruncate releases some of the disk space allocated to a file. The file must be opened for writing.

```
error = FmpTruncate(dcb,error,blocks)

integer*2 dcb(*), error
integer*4 blocks
```

where:

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

blocks is a double integer specifying the minimum number of blocks to which the file is to be truncated.

The file specified by DCB is truncated to no less than the specified double integer number of blocks. More blocks than this may actually remain, depending on internal considerations. Files are never truncated to less than one block. It is the responsibility of the calling program to make sure that valid data is not truncated. The EOF mark should be in the area that remains after truncation. You should close the file after it is truncated.

For example, if, after performing sequential writes to a variable-length record file (type 3 and above), you want to truncate the space beyond the current EOF mark, you can use the following (assuming the file is positioned at EOF mark):

```
Call  FmpPosition(dcb,error,record,position)
if (error.lt.0) ...
blocks = (position + 128)/128
Call  FmpTruncate(dcb,error,blocks)
if (error.lt.0) ...
Call  FmpClose(dcb,error)
```

The calculation "position + 128" includes one word for the EOF mark, and rounds up the position so that all words in the current block are included. Dividing by 128 converts the number of words to number of blocks.

FmpUdspEntry

FmpUdspEntry returns the directory name for the specified entry and User-Definable Directory Search Path (UDSP).

error = FmpUdspEntry(*udspnum*, *entnum*, *dirname*, *error*)

```
integer*2 udspnum, entnum, error
character*(*) dirname
```

where:

udspnum is an integer that specifies the UDSP number.

entnum is an integer that specifies the entry for the UDSP number.

dirname is a character string that returns the directory name for the specified entry in the specified UDSP.

error is an integer that returns one of the following values:

0	No error occurred
−1	Not under session control
−2	UDSP tables not set up correctly
−247	If the entry is undefined, or if *udspnum* and *entnum* are out of bounds with the definition for the session.

FmpUdspInfo

FmpUdspInfo returns the current User-Definable Directory Search Path (UDSP) information for your session.

error = FmpUdspInfo(*udsps*, *depth*, *next*, *error*)

```
integer*2 error, udsps, depth, next
```

where:

udsps is an integer that returns the number of UDSPs defined for the current session.

depth is an integer that returns the UDSP depth defined for the current session.

next is an integer that returns the next available UDSP. *next* is set to zero if all UDSPs are defined.

error An integer that returns one of the following values:

0	No error occurred
−1	Not under session control
−2	UDSP tables not set up correctly

FmpUniqueName

FmpUniqueName creates a 16-character file name that should be unique within a system that does not contain files from another system.

```
CALL FmpUniqueName(prefix,uniquename)

character*(*) prefix, uniquename
```

where:

prefix is a character string specifying a prefix for the file name.

uniquename is a character string that returns the generated file name.

The name is created by appending a reading of eleven characters from the system clock to a user-supplied prefix. The clock reading is expressed as a string of hex digits. A typical *uniquename* is "TEMP7C43E20FF21". If the user-supplied prefix is less than five characters, the file name is padded with blanks on the right. If the prefix is greater than five characters, the file name is truncated on the right.

If the file may be transferred to a FMGR directory, the prefix should be chosen to minimize the chance of a duplicate file name when *uniquename* is truncated to six characters.

FmpUnPurge

FmpUnPurge restores a purged file. The file must have existed and been purged, and its disk space must not have been allocated to another file.

```
error = FmpUnPurge(filedescriptor)

character*(*) filedescriptor
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string specifying the name of the file to be unpurged.

FmpUnPurge verifies the directory entry for the file and any extents, and ensures that none of its disk space has been allocated to another file. If it passes both tests, FmpUnPurge reallocates all of its space and converts its directory entries back to the normal status. The file's protection, time stamps, and other attributes are restored exactly as they were at the time that the file was purged.

Directories cannot be unpurged.

If several purged files have the same name, it is difficult to determine which is to be unpurged. The result of an FmpUnPurge call is not defined.

Files cannot be unpurged if a file already exists with the same name; the existing file must be renamed first.

FmpUpdateTime

FmpUpdateTime returns the time of the last update for the named file. The file is not opened in the process.

error = FmpUpdateTime(*filedescriptor*, *time* [, *slink*])

```
character*(*) filedescriptor
integer*4 time
logical slink
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

filedescriptor is a character string specifying the name of the file.

time is a double integer that returns the time of the last update expressed in seconds since January 1, 1970.

slink is an optional boolean variable that indicates whether to return the time of the last update of the symbolic link file or the file that it references. The possible values are as follows:

TRUE (negative value)
Return the time of the last update of the symbolic link file.

FALSE (non-negative value)
Return the time of the last update of the file referenced by the symbolic link (this is the default).

The update time is set when a file is closed, but only if the file was changed while it was open.

Routines are available to convert the time value to an ASCII string. Usually, however, the returned time is compared to times in the same format, so the calling program may not have to convert the format.

FmpWorkingDir

FmpWorkingDir returns the name of your current working directory. The current working directory can be either a global directory or a subdirectory.

```
error = FmpWorkingDir(directory [,format])

character*(*) directory
integer*2 error, format
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

directory is a character string that returns the name of the current working directory.

format is an optional integer parameter that defines the format of the directory string being returned. Possible values for *format* and their definitions are:

- 0 (default) if a working directory is a global directory, it is returned in the trailing directory format (::dir); otherwise, the working directory is returned in hierarchical format with no trailing slash.
- 1 the working directory is returned in hierarchical format with no trailing slash.
- 2 the working directory is returned in hierarchical format with a trailing slash.

The returned name is in a format suitable for passing to other routines, such as FmpSetWorkingDir.

If the name contains more than 63 characters, the name is truncated to 63 characters and an error is returned.

If there is no working directory, then an error is returned and the name is undefined.

FmpWrite

FmpWrite writes data to a file of any type. The file must be opened for write access.

length = FmpWrite(*dcb*, *error*, *buffer*, *maxlength*)

integer*2 *length*, *dcb*(*), *error*, *buffer*(*), *maxlength*

where:

length is an integer that returns the number of bytes actually transferred, or a negative error code. If more than 32767 bytes are transferred, the returned length is a negative number. If this negative number is equal to the value of the *error* parameter, an error has probably occurred.

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

buffer is the name of a word-aligned buffer that contains the data to be transferred.

maxlength is the maximum number of bytes to write; it is interpreted as an unsigned one-word integer from 0 to 65534. For values larger than 32767, set *maxlength* to the desired maximum number of bytes minus 65536; for example, 40000 bytes is expressed as −25536 (40000 − 65536 = −25536).

FmpWrite writes data at the current position of the file. The file position can be set by other FMP routines, such as FmpSetPosition and FmpAppend.

For sequential (type 3 or above) files, one record is written. The DCB buffer is used during the transfer. If the file is not opened in update mode, the entire record is transferred and an end-of-file mark is written after it. If the file is opened in update mode, then the length transferred will be the shorter of the existing and supplied record lengths. No end-of-file mark is written.

For type 2 files, one record is written, using the shorter of the defined and supplied record lengths. The DCB buffer is used for the transfer.

For type 1 files (and files forced to type 1), multiple records may be written, depending on the supplied record length. The data is transferred directly from the user buffer to the disk. The returned length is rounded up to an even number if necessary.

For type zero (device) files, one record is transferred. The data is written directly from the user buffer to the device. No more than 32767 bytes can be transferred with one call.

FmpWriteString

FmpWriteString is similar to FmpWrite, except that the data to be transferred is supplied in the *string* parameter.

```
length = FmpWriteString(dcb,error,string)

integer*2 length, dcb(*), error
character*(*) string
```

where:

length is an integer that returns the length of the record written to the file, or a negative error code. It may be less than the actual string length, but never longer.

dcb is an integer array containing the DCB for the file.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

string is a character string of up to 256 bytes from which data is transferred. The *string* parameter cannot be greater than 256 bytes because the data must pass through an internal buffer of 256 bytes. If *string* is longer than this limit, an error is returned.

MaskDiscLu

MaskDiscLu returns the disk LU of the last file returned by FmpNextMask. It can also be used to obtain the DS connection number.

```
disklu = MaskDiscLu(dirdcb)

integer*2 disklu,dirdcb(*)
```

where:

dirdcb is a control array, initialized by FmpInitMask

The following declarations can be used to get the DS connection number:

```
integer*4 MaskDiscLu, RTNVAL
integer*2 dirdcv(*), diskLu, DSnum
integer*2 Irtnval(2)
equivalence (IRTNVAL, RTNVAL, diskLu),
            (IRTNVAL(2),DSnum)
   .
   .
   .
RTNVAL = MaskDiscLu(dirdcb)
```

MaskIsDS

MaskIsDS is a logical function that determines if masking is searching a remote file system.

```
bool = MaskIsDS(dirdcb[,dsinfo])

logical bool
integer*2 dirdcb(*)
character*(*) dsinfo
```

where:

bool is a boolean variable that returns TRUE (negative value) if masking is searching a remote file system; otherwise, *bool* returns FALSE (non-negative value).

dirdcb is a control array, initialized by FmpInitMask.

dsinfo is an optional character string that returns the DS information of the mask. For example, the remote user account name, node name, or both, along with the required delimiters are returned, as in ">27", ">SYS3", "[USER]", and ">SYS3[USER/PASSWORD]".

MaskMatchLevel

MaskMatchLevel is an integer function that returns the number of the directory level in which the last file was matched.

```
matchlevel = MaskMatchLevel(dirdcb)

integer*2 matchlevel, dirdcb(*)
```

where:

matchlevel is an integer set to the number of the directory level containing the last file that was matched.

dirdcb is an integer array initialized by FmpInitMask.

For example, if the search mask is /GLOBAL.DIR.D and the matched file is /GLOBAL/SUBDIR/FILE, then *matchlevel* returns 2, to indicate that the file is nested two levels below the global directory. This value can help in creating new names for copy or rename operations, although Calc_Dest_Name is more commonly used for that function.

MaskOldFile

MaskOldFile is a boolean function that checks if the last file returned by FmpNextMask is a FMGR file.

```
bool = MaskOldFile(dirdcb)

integer*2 dirdcb(*)
logical bool
```

where:

bool is a boolean variable that is set to TRUE (negative value) if the last file returned by FmpNextMask is a FMGR file; otherwise, *bool* is set to FALSE (non-negative).

dirdcb is an integer array initialized by FmpInitMask.

MaskOpenId

MaskOpenId is an integer function that returns the D.RTR open flag of the last file returned by FmpNextMask.

```
openid = MaskOpenId(dirdcb)

integer*2 openid, dirdcb(*)
```

where:

openid is an integer that returns the ID number of the program that has the file open. If the file is not open, *openid* is set to zero. If the file is open, the ID number of a program that has the file open is returned in bits 0 through 7, and the value of the exclusive bit is returned in bit 15.

dirdcb is an integer array initialized by FmpInitMask.

The returned program may not be the only program that has the file open. Refer to the FmpOpenFiles routine description for more information on the format of the open flag.

MaskOwnerIds

MaskOwnerIds returns the owner and group IDs for the last file returned by FmpNextMask.

```
CALL MaskOwnerIds(dirdcb,ownerid,groupid)

integer*2 dirdcb(*), ownerid, groupid
```

where:

dirdcb is a control array, initialized by FmpInitMask.

ownerid is the integer ID number of the file's owner.

groupid is the integer ID of the file owner's group.

The *ownerid* and *groupid* parameters along with the DS Connection number can be used with DsIdToOwner and DsIdToGroup to obtain the ASCII owner and group names. The DS connection can be obtained from MaskDiscLu.

MaskSecurity

MaskSecurity is an integer function that returns the security code of the last file returned by FmpNextMask, if the file is a FMGR file. For FMP files, it returns zero.

```
seccode = MaskSecurity(dirdcb)

integer*2 seccode, dirdcb(*)
```

where:

seccode is an integer that returns the security code of the last file returned by FmpNextMask, if the file is a FMGR file. For FMP files, *seccode* is set to zero.

dirdcb is an integer array initialized by FmpInitMask.

WildCardMask

WildCardMask checks the mask for wildcard characters.

```
wild = WildCardMask(mask)

logical wild
character*(*) mask
```

where:

mask is a character string that contains the mask to be checked.

wild is a boolean indicating the presence of a wildcard character. The *wild* parameter returns one of the following values:

TRUE (a negative value)
: The mask contains a wildcard character ("@" or "−"), or the mask qualifier contains any of the search directives ("d", "e", or "s"), or the specified mask can refer to more than one file for another reason.

FALSE (non-negative value)
: The mask cannot refer to more than one file.

If WildCardMask returns FALSE, there is no need to use the mask search routines to find a specific file; it is faster to use the specified mask to open and access the file directly.

Using the FMP Routines with DS

All of the FMP calls that use a *filedescriptor* parameter can access files over DS, except FmpRunProgram, FmpSetWorkingDir, and FmpSetOwner because they perform system functions that should not be performed from a remote system.

The file descriptor must contain 63 or fewer characters, including the remote user account name and node specifications. As a result, there may be some files that cannot be accessed over DS because they have a long file name or directory path that cannot fit with the DS information into the 63-character file descriptor.

The name building and parsing routines return the DS field as their last parameter. The returned DS field contains the DS delimiters. If a file is located in a remote system, the name returned by FmpFileName includes the node name.

Some of the FMP routines do not perform exactly the same over DS as they do on a single system. The limitations are as follows:

- FmpOpen does not use a DCB buffer larger than 8 blocks (1024 words), even if a larger buffer is specified.
- FmpOpen cannot open an LU at a remote system. It returns an error if such an attempt is made.
- FmpOpenFiles can only identify the program that has a file open if the program and the file are on the same system. If a file is open via DS, FmpOpenFiles reports that it is open, but cannot report the name of the program that has it open, because all files opened via DS are opened by the TRFAS program.
- Files opened exclusively via DS are honored, except for FMGR files.

Special Purpose DS Communication Routines

The following calls permit your programs to perform special functions, all with DS transparency. They allow you to establish connections to accounts at remote systems.

Note The following routines are internal FMP routines, so they should be used with caution. For example, it is possible to inadvertently close the wrong file by passing an incorrect connection number.

All of the variables used by the special purpose routines are single integers, except as noted.

DsCloseCon

DsCloseCon closes a connection opened by DsOpenCon.

```
error = DsCloseCon(conn)

integer*2 error, conn
```

where:

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

conn is an integer that specifies the connection number.

It is important to close connections when the DS operations are completed, because only 64 connections are available, and they are not automatically released when the calling program terminates or when the DS operations complete.

DsDcbWord

DsDcbWord returns the first word of the DCB as it would appear if the file associated with it was not opened through DS.

```
error = DsDcbWord(conn,word)

integer*2 conn, word
```

where:

conn is an integer that specifies the connection number.

word is an integer that returns the first word of the DCB.

DS transparency is implemented by replacing the first word of the DCB with the negative connection number. A DCB associated with a file over DS is detected by examining bit 6 of the first word of the DCB, but that practice is not recommended.

DsDiscInfo

DsDiscInfo returns the number of tracks and blocks per track of the specified disk volume on the system associated with the connection number.

```
error = DsDiscInfo(conn,lu,ntracks,bpert)

integer*2 error, conn, lu, ntracks, bpert
```

where:

error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
conn	is an integer that specifies the connection number.
lu	is an integer that specifies the LU of the disk volume about which the track and blocks per track information is wanted.
ntracks	is an integer that returns the number of tracks for the specified disk volume.
bpert	is an integer that returns the number of blocks per track of the specified disk volume.

DsDiscRead

DsDiscRead reads the disk on the system specified by the connection number.

```
error = DsDiscRead(conn,buf,len,track,sector)

integer*2 buf(*), error, conn, len, track, sector
```

where:

error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
conn	is an integer that specifies the connection number.
buf	is an integer array that returns data from the disk.
len	is an integer that specifies the number of words ~~amount~~ of data to be read. A maximum of 1024 words ~~4096 characters~~ can be read.
track	is an integer that specifies the track from which to read.
sector	is an integer that specifies the sector from which to read (64 words per sector).

The first word of the DCB that contains *conn* must first be set by DsSetDcbWord.

This routine should be used only by users with a detailed knowledge of DCBs and their contents.

DsFstat

DsFstat performs an FSTAT call for the system associated with the specified connection number.

```
error = DsFstat(conn,buffer,len[,iform[,iop]])

integer*2 buffer(256), error, len, iform, iop
```

where:

error	is an integer that returns a negative code if an error occurs or zero if no error occurs.
conn	is an integer that specifies the connection number.
buffer	is an integer array that returns the status of the cartridges.
len	is an integer that specifies the length of the buffer in words.

The *iform* and *iop* parameters are optional parameters that are used only when the remote node is an RTE-6/VM system. These parameters are identical to the *iform* and *iop* parameters in the FSTAT call for RTE-6/VM (see the *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005, for a description).

DsNodeNumber

DsNodeNumber returns the node number associated with the specified file.

```
node = DsNodeNumber(filedescriptor)

character*(*) filedescriptor
integer*2 node
```

where:

node	is an integer that returns the number of the node associated with the specified file. A zero is returned if the file is not remote.
filedescriptor	is a 63-character string that specifies the name of a file.

DsOpenCon

DsOpenCon opens a connection to the remote user account/node specified.

```
error = DsOpenCon(string,conn)

integer*2 error, conn
character*(*) string
```

where:

string is a character string that specifies the remote user account name, node name, or both, along with the required delimiters, as in ">27", ">SYS3", "[USER]", and ">SYS3[USER/PASSWORD]". *string* must not contain a file name, only DS information.

conn is an integer that returns the connection number.

error is an integer that returns a negative code if an error occurs or zero if no error occurs.

The connection number returned by DsOpenCon is used in the other DS communication routines to identify the connection.

DsSetDcbWord

DsSetDcbWord changes the first word of the DCB to make the DsDiscRead routine work.

```
error = DsSetDcbWord(conn,word)

integer*2 error, conn, word
```

where:

error is an integer that returns a negative code if an error occurs or a zero if no error occurs.

conn is an integer that specifies the connection number.

word is an integer that specifies the word to be changed.

This routine should be used only by users with a detailed knowledge of DCBs and their contents.

Example Programs for FMP Routines

Three sample programs follow. The first program simply demonstrates the use of the simplest (open, close, read, write) FMP routines. The second shows how file masking, a somewhat more advanced FMP function, is used. The third combines many of the FMP routines in an advanced application.

Read/Write Example

The following program copies one file into another, one record at a time. It illustrates the use of FmpOpen, FmpRead, FmpWrite, and FmpClose, as well as FmpReportError.

```
ftn7x,s
      program copy
      implicit integer(a-z)

c Program to copy a file to another file.

      integer dcb1(528), dcb2(528), buffer(128)
      character file1*30,file2*30

c Open the source and destination files;
c use large DCBs to go fast.

      call fparm(file1,file2)
      type1 = FmpOpen(dcb1,err,file1,'ros',4)
      if (err .lt. 0) goto 10

      type2 = FmpOpen(dcb2,err,file2,'woc',4)
      if (err .lt. 0) goto 20

c copy the data

      do while (.true.)
            len = FmpRead(dcb1,err,buffer,256)

c look for errors and end-of-file

         if (err .lt. 0) goto 10
         if (len .eq. -1) goto 30

c none of those, so write the record.

         call FmpWrite(dcb2,err,buffer,len)
         if (err .lt. 0) goto 20
      enddo

c come here to report errors

 10   call FmpReportError(err,file1)
      goto 30
 20   call FmpReportError(err,file2)

c come here to close files and quit

 30   call FmpClose(dcb1,err)
      call FmpClose(dcb2,err)
      stop
      end
```

Mask Example

The following program shows how FmpInitMask, FmpNextMask, and FmpMaskName can be used to generate a list of files that match a mask.

```
ftn7x,l,s
      program files
      implicit integer (a-z)

c files lists the names of files that match the mask

      integer dirdcb(372), entry(32)
      character curpath*(63), newname*(63), mask*(63)
      logical FmpNextMask

c get the mask

      call fparm(mask)

c initialize the directory dcb, report errors

      if (FmpInitMask(dirdcb,err,mask,curpath,372).lt. 0) then
            call FmpReportError(err,mask)
            stop
      endif

c while errors are nonfatal, print name of file

      do while (FmpNextMask(dirdcb,err,curpath,entry))
         if (err .lt. 0) then
            call FmpReportError(err,curpath)
         else
            call FmpMaskName(dirdcb,newname,entry,curpath)
            write(1,*) newname
         endif
      enddo

c if search ended with error, print error

      if (err .lt. 0) then
         call FmpReportError(err,curpath)
      endif
c
c end mask search
c
      call FmpEndMask(dirdcb)
      stop
      end
```

Advanced FMP Example

The following is a much larger program that builds a data base and writes records to it.

In the example, FmpUniqueName is called to create a unique file name for the data base in the directory "CRDB" with a file type extension of "DAT". The program illustrates name building, file positioning, and many other less-frequently used FMP routines. The database built here is simply a type 2 file, it should not be confused with an Image data base.

```
ftn7x,s
      program crdb
      implicit integer(a-z)

c Program to create a database in a type 2 file

      parameter (recordlen=30)
      parameter (recordbytes=2*recordlen)
      parameter (filesize=24)

      integer dcb(144), buffer(recordlen)
      character name*63, asciitime*28,charbuffer*(recordbytes)
      character tempname*16

c Note use of double integers for times, record numbers

      integer*4 time, currec

c Allow "charbuffer" as the string version of "buffer"

      equivalence (buffer,charbuffer)

c Make up the name

      call FmpUniqueName('D',tempname)
      call FmpBuildName(name,tempname,'DAT',0,'CRDB',2,
     *                  filesize, recordlen,' ')
      namelen = trimlen(name)

c Open the database for read, write; create it; update is implicit.

      call FmpOpen(dcb,err,name,'RWC',1)
      if (err .lt. 0) goto 20

c Print the file name, and when it was created

      err = FmpCreateTime(name,time)
      if (err .lt. 0) goto 20
      call daytime(time,asciitime)
      write(1,*) 'File ',name(1:namelen),' created ',asciitime

c Loop on adding records

      do while (.true.)

c See which record number to change
```

```
 5  write(1,*) 'Record to add? _'
    read(1,*,end=10,err=10) currec

c Position to this record (allow FMP to trap bad record number)

    call FmpSetPosition(dcb,err,currec,-1J)
    if (err .eq. -12) then
      write(1,*) 'That record does not exist'
      goto 5
    endif
    if (err .lt. 0) goto 20

c Get a value for the record

    write(1,*) 'Enter record contents: _'
    read(1,'(a)') charbuffer

c Put it in the file

    call FmpSetPosition(dcb,err,currec,-1J)
    if (err .lt. 0) goto 20
    call FmpWrite(dcb,err,buffer,recordbytes)
    if (err .lt. 0) goto 20

c Post the file to show what to do if this is shared access

    call FmpPost(dcb,err)
    if (err .lt. 0) goto 20
  enddo

c Come here when the last record is entered

 10 write(1,*) 'All done'
    goto 30

c Come here to report errors

 20 call FmpReportError(err,name)

c Come here to close file, purge it, and quit

 30 call FmpClose(dcb,err)
    err = FmpPurge(name)
    if (err .lt. 0) then
      call FmpReportError(err,name)
    endif
    stop
    end
```

9

VMA and EMA Programming

Both the Virtual Memory Area (VMA) and Extended Memory Area (EMA) features of RTE-A allow applications to manage large data arrays. Both features permit the amount of data used by a program to exceed the 32-page maximum size of logical memory.

VMA implements a demand-paged virtual memory subsystem that allows your application programs to access data areas as large as 65,536 pages. The program data resides on disk, and the operating system swaps pages of data into memory as they are needed.

EMA implements a subset of VMA where data resides entirely in physical memory, that is, the data is not swapped between disk and memory. Use of EMA rather than VMA may result in a great increase in program speed by avoiding disk I/O.

EMA data can be shared by other programs; VMA data, on the other hand, is not shareable. A VMA or EMA variable cannot be used as a parameter in an EXEC or FMP call. However, there are VMA/EMA subroutines that provide these EXEC and FMP functions with VMA/EMA variables. See the section on General Purpose VMA/EMA subroutines.

The large arrays are managed via the VMA/EMA software and firmware. VMA, EMA, and shareable EMA are all declared in the same manner in the source program; the distinction is made when linking the program.

Table 9-1. 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 in physical memory.
Physical Memory	The actual semiconductor memory installed in the computer.
Virtual Memory Area (VMA)	An area on disk that can be used to extend main memory. This disk memory can contain very large data arrays accessible to your programs. Data in disk memory (virtual data) can be accessed via a simple program statement (I=J (5000), for example). The operating system makes disk memory look like logical memory to the program.
Extended Memory Area (EMA)	A subset of virtual memory. This is an area of physical memory that extends beyond the logical address space of the program, and can be used for large data arrays.
Virtual Memory Mapping Segment (VSEG)	The last two pages of the user logical address space. The VMA/EMA firmware uses these two pages to map in VMA or EMA data accessed by your program.
Mapping Segment Size (MSEG)	The guaranteed maximum VMA or EMA size (in pages) that can be present in the user program logical address space. The maximum size pertains regardless of page boundaries. The maximum number of EMA pages that can be mapped into the user logical address space is the MSEG size plus one. The MSEG area is the last (MSEG size + 1) logical pages of the user map. The extra page is known as the spillover page. Some routines, such as MMAP and .ESEG, start mapping at the start of the MSEG area, rather than only in the VSEG area. They are software routines that make special micro-code calls to do the mapping.
Page Table (PTE)	A data structure that indicates to the firmware which pages of the VMA data are currently in physical memory, and where they are located.
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.
Backing Store File	The file on disk used for storage of the virtual data.

Virtual Memory Area (VMA)

VMA handles large data arrays by storing the data on disk, maintaining a portion of the data in physical memory as the data is used by the program. The portion of VMA data that resides in physical memory is known as the "working set". The portion that is stored on disk resides in an FMP file called the "backing store". Up to 65,536 pages of data can be accessed this way.

The location of the working set in physical memory is within the program's partition, immediately after the program's in-memory image. For CDS programs, the working set is within the data partition, just after the pages used for the data segment. The working set contains both VMA data pages and pages devoted to the VMA page table (PTE). The PTE is a data structure that indicates which VMA data pages are currently in physical memory, and in which physical pages. Figure 9-1 shows the memory placement of the working set.

When a program is first dispatched, none of the VMA data resides in the working set. For the program to access a VMA data element, the data must reside in physical memory. The VMA/EMA software and firmware routines transfer data between the backing store and the working set and map data residing in the working set into the program's logical address space.

A virtual memory array or variable is declared in the program as an EMA array or variable. When the program is linked, use the LINK VM command to use the VMA subsystem, rather than EMA. The size of the working set and of the virtual memory area on disk are set by using LINK commands. The default sizes of the working set and the virtual memory are 32 and 8192 pages, respectively. These values can also be modified using the WS and VS RTE 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, disk space for the backing store file is allocated only as it is needed.

An alternate mode of VMA operation can be used for programs that create a large amount of data that will be used later, or programs using 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 alternate 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 disk, and the requested page of data is swapped into that page.

In high level languages, after your program indicates which arrays are to be treated as VMA or EMA, the compiler automatically emits calls to the VMA/EMA software and firmware routines. For a Macro/1000 program, explicit calls to the VMA/EMA subroutines must be made in order to map in data.

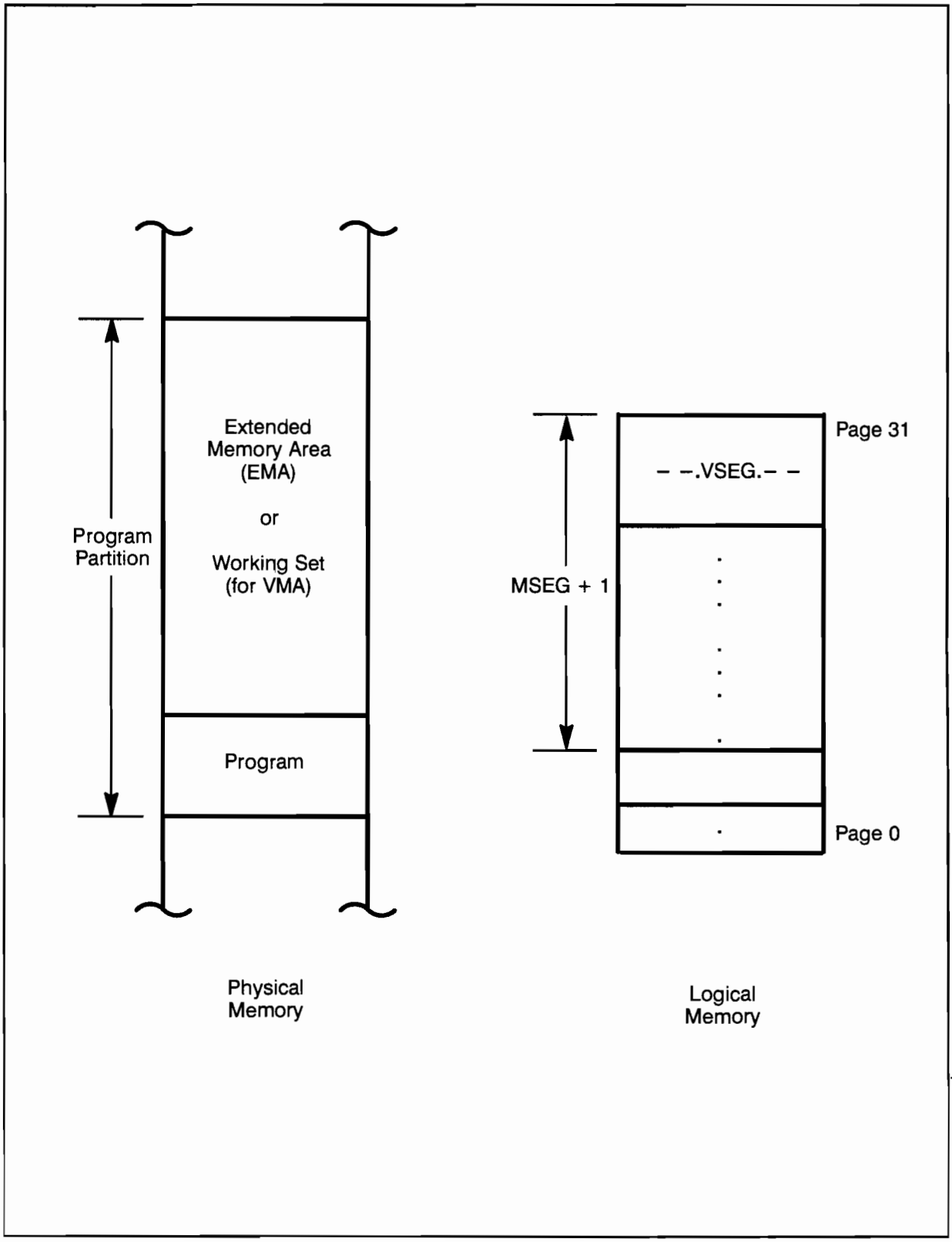


Figure 9-1. VMA Memory Structure

Extended Memory Area (EMA)

EMA is a subset of VMA. With EMA, however, all the program data resides in physical memory. You can think of EMA as a special case of VMA where the working set is large enough to accomodate all data pages, and therefore the backing store file is not necessary. EMA data is mapped into logical memory via the EMA software and firmware routines.

The Extended Memory Area resides within the program's partition, just after the program itself. For CDS programs, the EMA area is in the data partition. Figure 9-1 shows the structure of physical and logical memory. EMA makes use of a page table (PTE), just as VMA does. For an EMA area, the PTE indicates to the firmware that each data page resides in memory.

Using Shareable EMA

EMA data may be either local, meaning it is accessible only to the program that declared it, or shared between multiple programs. Shareable EMA allows large amounts of data to be shared easily among programs. Programs that use shareable EMA can be written the same as programs using local EMA data. At LINK time, a program's EMA data can be declared as shareable by using the LINK SH command. Note that VMA data cannot be shared, and that programs compiled by FTN4X or ADA cannot use shareable EMA.

A shareable EMA program requires an additional partition available at execution time for the EMA data. The shared EMA data is placed in a separate EMA partition in the dynamic memory area or in a reserved partition, as specified at link time. The label associated with the area is specified at link time. EMA data residing in this partition can be shared by up to 255 different programs at once. This is done by declaring the same shareable EMA label at link time for each program. Note that each of the programs accessing the shareable EMA partition must declare the EMA partition exactly the same. For example, if an EMA partition contains six arrays, then all programs accessing the partition must contain identical declarations for each of the six arrays.

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 count of the number of programs using each shareable EMA partition. As more programs are scheduled that use the data area, the count increases. As these programs terminate, the count decreases. When the count reaches 0, the partition is released to the free memory pool, unless the partition is locked, as with a LKEMA subroutine call.

Note that if a program terminates saving resources, the shareable EMA partition (a resource) is not released, and the active count does not decrease.

In some cases, you may wish to leave the partition as a shareable EMA partition, even when all programs accessing the area are dormant. In this case, use the subroutine LKEMA to lock the shareable EMA partition as a data area. The partition will then remain a reserved data area until the system routine ULEMA is called to unlock the shareable EMA partition or the operator enters the UL system command to forcibly unlock the partition, and the active user count is zero or drops to zero.

The section on Programming with VMA and EMA describes the LKEMA and ULEMA library routines.

Resource numbers, described in Chapter 2, can be used to synchronize the use of shareable EMA data. Coordination of multiple programs updating the SHEMA must be performed by your programs.

Shareable EMA Program Considerations

At link time, a program's EMA data can be declared as shareable data by using the LINK SH command:

SH, *label* [, *<reserved partition number>*]

The specified label may be any sixteen ASCII characters starting with a non-numeric character. Another program for which you specify the same label will use the same shareable EMA partition when it executes. If a reserved partition number is not specified, the shareable EMA is allocated in dynamic memory. At most, 255 programs can use the same shareable EMA area at one time.

For each shareable EMA area, there is an entry in a shareable EMA table that resides in Extended System Available Memory (XSAM). The maximum number of entries in this table is dependent on the amount of free XSAM available. When an ID segment or prototype ID segment is created for a shareable EMA program, the operating system also creates an entry in the shareable EMA table for the appropriate label, provided there is not already an entry for that label in the table.

If there is already an entry for the shareable EMA label, then the "in-system" count in that entry is incremented by one. The maximum number of RP'ed programs using the same shareable EMA area is 255. If there is not already an entry for the label and there is space in XSAM for a new entry, then an entry is made, and the count for that label entry is set to one. If there is no room in XSAM and there is not yet an entry for the label, then an error is issued.

When an ID segment for a shareable EMA program is deallocated from the system (an OF,prog,ID command will do this), the "in-system" count in the shareable EMA table entry is decremented. If the count reaches zero, then, provided the shareable EMA area is not locked, the table entry is deallocated and the shareable EMA area is no longer known to the system.

Partition Considerations

A VMA or EMA program can run in the dynamic memory area or can be assigned to a reserved partition. A shareable EMA program can also run in the dynamic memory area or in a reserved partition. If an EMA program uses shareable EMA, the shareable EMA can be specified in the LINK SH command to be allocated in a reserved partition. If no reserved partition number is specified in the LINK SH command, the shareable EMA area will be allocated in the dynamic memory area.

If two programs declare the same shareable EMA label, and only one specifies that the shareable EMA area should go into some reserved partition, then the shareable EMA area may or may not be set up in a reserved partition. The system allocates the shareable EMA table entry when it sets up the ID segment for the program requesting shareable EMA. This entry holds information about where the shareable EMA area is to be allocated. Whether or not the EMA area is in a reserved partition depends upon which program is set up first. The same holds true for two programs declaring the same shareable EMA label but specifying different reserved partitions for the shareable EMA area.

Note that a program cannot run in the same reserved partition as that allocated to its shareable EMA. An attempt to do so will result in an EM91 abort when the program is run.

The shareable EMA area is allocated when the first program using it is scheduled and dispatched. The size of the area allocated is dependent upon the EMA size declared by this program. A program scheduled later that declares a larger EMA size will abort with an EM 90 error; therefore, it is best that all programs using a common shareable EMA have the same EMA size (use the LINK EM command).

Shareable EMA Partitions

Shareable EMA labels are defined at link time and the associated partitions are created when the program is run, not at generation or reconfiguration time. Once a shareable EMA area is set up, whether it is in the dynamic area or in a reserved partition, it is not swappable, and it is not deallocated until the number of active (non-dormant) programs using it reaches zero (unless the area is locked).

It is highly recommended that all programs using the same shareable EMA area declare the same EMA size and location (reserved partition number or dynamic memory) when the programs are linked. This precaution will prevent any EM90 or EM91 errors at dispatch time.

System Common and SHEMA Examples

This section shows program examples, in both FTN7X and Pascal, that communicate using shared EMA. The following two FORTRAN programs illustrate a simple exchange of data using SHEMA. The shared variables must be declared in a labeled common block and the common block must be declared in the $EMA statement.

Program 1

```
FTN7X,L
$EMA/BIG/
       PROGRAM SHAR1
       COMMON /BIG/IARRAY(100)
       DIMENSION INAM(3)
       DATA INAM/6HSHAR2 /
C
C    INITIALIZE THE ARRAY
C
       DO 10 I=1,100
          IARRAY(I)=1
 10    CONTINUE
C
C    LOCK THE SHAREABLE EMA PARTITION
C
       CALL LKEMA

C
C    SCHEDULE 'SHAR2' WITHOUT WAIT
C
       ICODE=9
       CALL EXEC(ICODE,INAM)
       END
```

Program 2

```
FTN7X,L
$EMA/LARG/
       PROGRAM SHAR2
       COMMON /LARG/IARRAY(100)
C
C    PRINT OUT THE ARRAY
C
       WRITE(1,*)'THE ARRAY IS : '
       WRITE(1,*)(IARRAY(I),I=1,100)
C
C    UNLOCK THE PARTITION
C
       CALL ULEMA
C
       END
```

The following two Pascal programs demonstrate how the pointer to a shared data item located in system common passes information between two programs with access to the same SHEMA partition area. The programmer must declare the SHEMA partition area with the LINK SH command and use the routines Pas.A1SharedSize and Pas.A1SetShared located in the Pascal library to access the SHEMA partition.

Program 3

```
$HEAP 2$
PROGRAM prog1( INPUT, OUTPUT);

TYPE    int = -32768..32767;
        big = array [1..100] of int;
        bigptr = ^big;
        com = record
                biggy :bigptr;
              end;
        comptr = ^com;

VAR     i, sizeblank : int;
        sizeshared, start, heap_stack : INTEGER;
        biggyptr : bigptr;
        commy : comptr;

FUNCTION sharedsize $ALIAS 'Pas.A1SharedSize'$ : INTEGER; EXTERNAL;
{ Returns the size of the SHEMA Partition that the program has access to }

FUNCTION setshared $ALIAS 'Pas.A1SetShared'$
        (start, heap_stack : INTEGER) : BOOLEAN; EXTERNAL;
{ Sets heap/stack area to begin at the specified address   }
{ ('start'), and to extend for specified number of words   }
{ ('heap_stack').  A true value returned indicates that    }
{ pointer has been set to the specified area.              }

FUNCTION common_blank $ALIAS 'Pas.BlankCom2'$ : comptr; EXTERNAL;
{ Returns a pointer to the system common area }

FUNCTION blank_size $ALIAS 'Pas.BlankSize'$ : int; EXTERNAL;
{ Returns the size of blank common area }

BEGIN
  sizeblank := blank_size;              { Get size of blank               }
  IF sizeblank <> 0 THEN BEGIN          { common area If not             }
    sizeshared := sharedsize;           { equal to zero, then            }
    IF sizeshared <> 0 THEN BEGIN       { Get size of shareable EMA      }
       {}                               { If SHEMA not equal to zero, then }

       commy := common_blank;           { Get pointer to sys common area }
       IF (setshared(0,150)) THEN BEGIN { Set up heap stack area         }
          NEW (biggyptr);
          WITH commy^ do
           biggy := biggyptr;
          FOR i := 1 TO 100 DO
             biggyptr^[i] := 1;         { Initialize the array           }
      END ELSE
      WRITELN('heap stack setup failure');
    END ELSE
    WRITELN('no access to shema');
  END ELSE
   WRITELN('no common available');
END.
```

Program 4

```
$HEAP 2$
PROGRAM prog2( INPUT, OUTPUT);

TYPE    int = -32768..32767;
        big = array [1..100] of int;
        bigptr = ^big;
        com = RECORD
                biggy : bigptr;
              END;
        comptr = ^com;

VAR     i, sizeblank : int;
        sizeshared, start, heap_stack : INTEGER;
        commy : comptr;

FUNCTION sharedsize $ALIAS 'Pas.A1SharedSize'$ : INTEGER; EXTERNAL;
{ Returns the size of the SHEMA Partition that the program has access to}

FUNCTION setshared $ALIAS 'Pas.A1SetShared'$
        (start, heap_stack : INTEGER) : BOOLEAN; EXTERNAL;
{ Sets heap/stack area to begin at the specified address ('start'), and }
{ to extend for specified number of words ('heap_stack').  A true value }
{ turned indicates rethat pointer has been set to the specified area.   }


FUNCTION common_blank $ALIAS 'Pas.BlankCom2'$ : comptr;
        EXTERNAL;
{ Returns a pointer to the system common area }
FUNCTION blank_size $ALIAS 'Pas.BlankSize'$ : int; EXTERNAL;
{ Returns the size of blank system common }

BEGIN
  sizeblank := blank_size;          { Get size of blank common area        }
  IF sizeblank <> 0 THEN BEGIN      { If not equal to  zero, then          }
    sizeshared := sharedsize;       { Get size of shareable EMA            }
    IF sizeshared <> 0 THEN BEGIN { If SHEMA not equal to zero, then       }
     {}                             {                                      }
      commy := common_blank;        { Gets pointer to sys common area      }
      IF (setshared(151,200)) THEN BEGIN { Set up heap/stack area          }
         FOR i := 1 TO 100 DO
         WRITELN(commy^.biggy^[i]);      { Write out the array             }
      END ELSE
      WRITELN('heap stack setup failure');
    END ELSE
    WRITELN('no access to shema');
  END ELSE
  WRITELN('no common available');
END.
```

Class I/O calls, program to program communication, also pass information between programs. The first Pascal program in this next program pair passes data to a buffer in SAM with a class number that it shares with the second program. The second program now has access to the buffer and awaits action by the first. Note: Pascal programs have EXEC calls declared as externals; FORTRAN programs do not.

Program 5

```
$HEAP 2$
Program IOSH1 (INPUT,OUTPUT);

TYPE
        int = -32768..32767;
        ntype = packed array [1..6] of char;
        string = packed array [1..12] of char;
        btype = array [1..10] of char;
        bufptr = ^btype;
VAR
        ibufr  : bufptr;
        runstr : string;
        name   : ntype;
        class, icode, buflen : int;
        i, start, heap_stack, shemasize  : integer;

FUNCTION sharedsize $ALIAS 'Pas.A1SharedSize'$ : INTEGER ; EXTERNAL;
{ Returns size of the SHEMA Partition that the program  has access to.  }

FUNCTION setshared $ALIAS 'Pas.A1SetShared'$
        (start, heap_stack : INTEGER) : BOOLEAN; EXTERNAL;
{ Sets heap/stack area to begin at the specified address ('start'), and }
{ to extend for specified number of words ('heap_stack').  A true value }
{ returned indicates that pointer has been set to the specified area.   }

PROCEDURE exec_9 $ALIAS 'EXEC'$
        (icode : int; name : ntype; class,dum1,dum2,dum3,dum4 : int;
        runst : string; buflen : int ); EXTERNAL;
{ Exec 9 call - immediate schedule a program without wait  }

PROCEDURE exec_20 $ALIAS 'EXEC'$
       (icode, icnwd : int; ibufr : bufptr; buflen,iop1,iop2,class : int);
        EXTERNAL;
{ Exec 20 call - class write/read }

PROCEDURE lock $ALIAS 'LKEMA'$; EXTERNAL;
{ Lock the SHEMA partition, so only the scheduled program  }
{ can access it                                            }

BEGIN
        runstr := 'ru,iosh2,1,1';
        name := 'IOSH2';
        {}
        shemasize := sharedsize;            { Get the size of SHEMA       }
        IF shemasize <> 0 THEN BEGIN        { If SHEMA not equal to       }
        {}                                  { zero, then                  }
          IF (setshared(0,25)) THEN BEGIN   { Set up the heap/stack area  }
            new(ibufr);
            FOR I := 1 TO 10 DO
```

```
                ibufr^[i] := '1';
                {}
                class := 0          { Set class to zero, so the system     }
                {}                  { can allocate a unique class number   }
                buflen := 10;
                icode := 20;

                { PLACE THE DATA OF 'IBUFR' INTO A BUFFER IN SAM, ALONG  }
                {  WITH ITS CLASS NUMBER                                 }

                exec_20(icode,0,ibufr,buflen,0,0,class);

                { LOCK THE SHEMA PARTITION }

                lock;

                { SCHEDULE 'IOSH2' AND SEND IT THE CLASS NUMBER }

                icode := 9;
                exec_9(icode,name,class,0,0,0,0, runstr,-12);
                {}
            END ELSE
            WRITELN('heap stack setup failure');
        END ELSE
        WRITELN('no access to shema');
        {}
END.
```

Program 6

```
$HEAP 2$
PROGRAM IOSH2 (INPUT,OUTPUT);

TYPE
        int = -32768..32767;
        btype = array [1..10] of char;
        ptype = array [1..5] of int;
        bufptr = ^btype;

VAR
        ibufr  : bufptr;
        class, icode, buflen : int;
        i, start, heap_stack, shemasize : integer;
        pram : ptype;

FUNCTION sharedsize $ALIAS 'Pas.A1SharedSize'$ : INTEGER ; EXTERNAL;
{ Returns the size of the SHEMA Partition that the program has access to }

FUNCTION setshared $ALIAS 'Pas.A1SetShared'$
        (start, heap_stack : INTEGER) : BOOLEAN; EXTERNAL;
{ Sets heap/stack area to begin at the specified address ('start'), and }
{ to extend for specified number of words ('heap_stack').  A true value }
{ returned indicates that pointer has been set to the specified area.   }


PROCEDURE exec_21 $ALIAS 'EXEC'$
        (icode, class : int; ibufr : bufptr; buflen :int);  EXTERNAL;
{ Exec 21 call - class get }
```

```
PROCEDURE unlock $ALIAS 'ULEMA'$; EXTERNAL;
{ Unlock the SHEMA partition }

PROCEDURE params $ALIAS 'RMPAR'$
       (pram : ptype); EXTERNAL;
{ Pick up the parameters sent fron the 'father' program }

BEGIN
       params(pram);              { pram[1] contains the class number    }
       {}
       shemasize := sharedsize;           { Get the size of SHEMA       }
       IF shemasize <> 0 THEN BEGIN       {  If SHEMA not               }
       {}                                 { equal to zero, then         }
          IF (setshared(26,50)) THEN BEGIN { Set up the heap/stack area }
             {}

             buflen := 10;
             icode := 21;
             class := pram[1];

             { GET THE DATA FROM SAM AND PUT IT IN 'IBUFR'

             exec_21(icode,class,ibufr,buflen);

             { PRINT OUT THE BUFFER AND CHECK IT FOR CORRECTNESS}

             write('The buffer read-in is ');
             FOR I := 1 TO 10 DO
               write(ibufr^[i]);

             { UNLOCK THE SHEMA PARTITION }

             unlock;
          {}
          END ELSE
          WRITELN('heap stack setup failure');
       END ELSE
       WRITELN('no access to shema');
{}
END.
```

Programming with VMA and EMA

Programming with VMA and EMA is available in FORTRAN, Pascal/1000, Macro/1000, and C/1000. In every VMA or EMA program, an extended memory area must be declared. This area can be subdivided using multiple labeled common blocks and array declarations.

In FORTRAN, Pascal/1000, and C/1000, calls to the VMA/EMA mapping subroutines are made without explicit user action (they are automatically emitted by the compiler). The VMA/EMA mapping subroutines described later in this chapter must be called from Macro/1000 programs in order to map in data. Additional VMA/EMA subroutines are summarized and described in this section. VMA and EMA errors that can be reported by the VMA/EMA subroutines are listed in Appendix A.

EMA programs compiled or assembled on RTE operating systems other than RTE-A or RTE-6/VM are not supported by the RTE-A Operating System. These EMA programs must be re-compiled or re-assembled before loading and executing so that calls to the RTE-A VMA/EMA mapping subroutines will be generated.

Programs that create default backing store files must have the scratch cartridge designated or the working directory assigned.

Programs that rely only on the external VMA/EMA features of RTE-6/VM will run in an identical manner on RTE-A.

All VMA/EMA subroutine call descriptions use the FORTRAN subroutine call format. If desired, the description of general formats included in the *Relocatable Libraries Reference Manual*, part number 92077-90037, can be consulted to convert the calls to FORTRAN functions, Pascal/1000, or Macro/1000 formats. In FORTRAN, routines called as functions should be declared as single integers unless otherwise stated. To convert the calls to the C/1000 calling sequence, refer to the *C/1000 Reference Manual*, part number 92571-90001.

The Three Models of EMA/VMA

The discussion of EMA and VMA up to this point has concentrated on what is known as the "Normal model" of EMA/VMA. There are actually three models of EMA/VMA: Normal, Large, and Extended.

The Normal EMA/VMA model describes the model to which most EMA/VMA programs conform. This model is used by programs that do not need access to multiple shared EMAs, or to both a local and a shared EMA, and do not require an EMA or working set size greater than 1022 pages. This model is also backward-compatible with the RTE-6/VM EMA/VMA scheme. The Normal model is available on all A-Series processors.

The Large EMA/VMA model is used by programs that must access multiple shared EMAs, or both a local EMA/VMA and a shared EMA. The maximum size of a Large model EMA or working set remains at 1022 pages. The Large model is available on all A-Series processors.

The Extended EMA/VMA model is available only on the A990 CPU with the "Extended VMA" firmware (A990 firmware revision 10 or later). You may also have A990 revision 9 firmware and use the DOWNLOAD utility to download the Extended VMA firmware with RTE-A revision 6000 or later. Refer to the *RTE-A Generation and Installation Manual*, part number 92077-90034, for information on the DOWNLOAD utility. The Extended EMA/VMA model offers the functionality of the Large model, but it increases the maximum size of an EMA or working set to 32,733 pages.

The appropriate EMA/VMA model is chosen at link time. The EM and VM LINK commands accept an option which selects the Large or Extended EMA/VMA model, rather than the default of Normal EMA/VMA. The features of the three models are summarized in Table 9-2. The "Number of pages of PTE overhead per program" feature refers to PTE pages devoted to the program, as opposed to PTE pages devoted to an EMA/WS area used by the program, which is given by the last line. For example, a Normal model SHEMA program contains no PTE pages in the program partition, but the SHEMA partition contains one PTE page. A Large model VMA program contains 2 PTE pages in the program partition: one for the program, and one for the working set.

Table 9-2. Features of the Three EMA/VMA Models

Feature	EMA/VMA Model		
	Normal	Large	Extended
Maximum size of any one EMA in pages	1022	1022	32733
Maximum size of a VMA working set in pages	1022	1022	32733
Maximum number of SHEMAs a program may access	1	64	64
Number of SHEMAs accessible when local EMA/VMA used	0	63	63
Number of pages of PTE overhead per program	0	1	2
Number of pages of PTE overhead per EMA or WS *	1	1	1/1024

* "1/1024" indicates that there is one PTE page for every 1024 data pages.

Declaring Extended Memory Area (EMA)

The first step in programming with VMA and EMA is to declare an extended memory area. Note that the VMA programs are identical to EMA programs until link time.

The FORTRAN $EMA directive or EMA statement may be used to specify that data is to reside in VMA or EMA. See the *FORTRAN 77 Reference Manual*, part number 92836-90001, for more information.

An EMA can be declared in Pascal by using the HEAP and, optionally, the EMA compiler options. Refer to the *Pascal/1000 Reference Manual*, part number 92833-90005.

An EMA can be declared in a Macro/1000 program using the EMA and ALLOC pseudo instructions. Refer to the *Macro/1000 Reference Manual*, part number 92059-90001.

An EMA can be declard in a C/1000 program using the "ema" pragma. Refer to the *C/1000 Reference Manual*, part number 92571-90001.

Allocating Secondary SHEMA Areas

Large and Extended model programs may access more than one shared EMA (SHEMA), and may access SHEMAs when a local EMA/VMA is declared as well. These programs declare EMA, either local or shared, or VMA just as Normal model programs do. But a Large or Extended model program may call library routine RteAllocShema, discussed in a later section, to programmatically attach SHEMAs to itself. The "local" area, which may be a SHEMA, is called the "primary" area, and the programmatically-attached SHEMAs are known as "secondary" SHEMAs.

At the time a Normal, Large, or Extended EMA program is initially scheduled, RTE-A sets up the named SHEMA area for those programs loaded with the LINK "SH" command. Large and Extended SHEMA programs (which are often called "SHEMA-only" programs) may also allocate extra SHEMAs via the RteAllocShema routine mentioned above.

When a secondary SHEMA is allocated to a program, some portion of the program's EMA/VMA address space is used to refer to that SHEMA. That is, some range of EMA/VMA addresses will access data in the SHEMA. Due to the format of the underlying page tables, SHEMAs are attached at 1024-page boundaries in a program's EMA/VMA address space. A program's primary EMA/VMA area always starts at EMA/VMA page 0. Assuming that the local area is less than 1024 pages in size, the first EMA/VMA page at which a secondary SHEMA may be attached is page 1024, the next is page 2048, and so on.

Because EMA/VMA addresses can reference up to 65,536 (64K) pages, the maximum number of SHEMAs that may be attached to a program is 64. To attach all 64, the program must be a SHEMA-only program, because all available 1024-page ranges are used for SHEMA. Similarly, a program that declares a local EMA/VMA may attach up to 63 SHEMAs. If the size of any SHEMA or the local EMA/VMA area exceeds 1024 pages then that area occupies more than one 1024-page range in the program's address space, and reduces the number of SHEMAs that may be attached to the program. These 1024-page ranges are sometimes referred to as "EMA segments" and each has a number from 1 to 64.

For example, consider a program with an EMA area of 32 pages that attaches a secondary SHEMA of 5 pages at page 1024 in its EMA address space. References to EMA pages 0 through 31 access the local EMA area, pages 32 through 1023 are illegal, and pages 1024 through 1028 access the SHEMA. References to the remaining pages, 1029 through 65535, are also illegal.

EMA/VMA Subroutines

The EMA/VMA subroutines described in the following sections provide size information about EMA and VMA, manage I/O transfers, and lock or unlock a shareable EMA partition. These subroutines can be divided into functional groups as follows:

Information Subroutines:

EMAST	Returns general information about VMA and EMA.
VMAST	Returns size of VMA and EMA.
RteExtendedEV	Checks if the CPU has extended EMA/VMA capability.

I/O Management Subroutines:

VMAIO	Performs VMA and EMA I/O data transfers to or from an LU.
EIOSZ	Determines the maximum guaranteed length of data transfer using VMAIO.
LOCKVMA	Locks from 1 to 64 VMA pages.
LOCKVMABUF	Locks a single VMA buffer.
LOCKVMA2BUF	Locks two VMA buffers.

Shareable EMA Subroutines:

LKEMA	Locks a shareable EMA partition.
ULEMA	Unlocks a shareable EMA partition.
RteAllocShema	Attaches a shared EMA to the program.
RteReturnShema	Detaches a shared EMA from the program.
RteRenameShema	Renames a shared EMA label.
RtePrimeShInfo	Returns information about the primary SHEMA of a SHEMA-only program.

VMA File Subroutines:

VMAOPEN	Creates/opens backing store file.
VMAPURGE	Purges the backing store file.
VMAPOST	Posts the working set to the backing store file.
VMACLOSE	Posts the working set and closes the backing store file.
VMAREAD	Reads data from the data file into VMA and EMA.
VMAWRITE	Writes data from the VMA or EMA to a data file.

FMGR VMA Subroutines:

CREVM	Creates backing store file.
OPNVM	Opens the backing store file.
PURVM	Purges the backing store file.
PSTVM	Posts the working set to the backing store file.
CLSVM	Posts the working set and closes the backing store file.
VREAD	Reads data from a data file into the VMA and EMA.
VWRIT	Writes data from the VMA or EMA to a data file.

VMA/EMA Mapping Management Subroutines:

.IMAP	Resolves address of array element and maps into logical memory.
.IRES	Resolves address of array element (does not map).
.JMAP	Resolves address of array element and maps; double integer.
.JRES	Resolves address of array element (does not map); double integer.
MMAP	Maps consecutive pages of EMA/VMA into logical memory.
.ESEG	Maps several pages of EMA/VMA into logical memory.

.LBP, .LBPR	Converts a virtual address to a logical address.
.LPX, .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.

Information Subroutines

EMAST (Return Information on VMA and EMA)

The EMAST subroutine returns information about the VMA or EMA of the calling program.

```
CALL EMAST(nema,nmseg,imseg[,iws])
```

where:

nema	is total page size of VMA or EMA (not including page tables).
nmseg	is total page size of mapping segment (MSEG), excluding the spillover page.
imseg	is starting logical page of MSEG.
iws	is working set page size (optional parameter). For an EMA program, this value is the same as *nema*.

An error is returned if a VMA or EMA is not defined in the calling program.

Upon return:

A-Register =	0	if normal return
	−1	if error occurred
B-Register =	0	if a Normal model program
	1	if a Large model program
	2	if an Extended model program

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 (Return Size of VMA and EMA)

The VMAST subroutine determines if the calling program uses VMA or EMA and returns the size of VMA or EMA.

```
CALL VMAST(ivma,isize)
```

where:

ivma indicates whether the calling program is VMA or EMA:

−2 = Not a VMA or EMA program
0 = EMA program
1 = VMA program

isize is the VMA or EMA size in pages. If the program is not a VMA or EMA program, a zero is returned.

Upon return:

A-Register = 0 if it is a Normal model program
1 if it is a Large model program
2 if it is an Extended model program

Example: Use the VMAST routine to determine if the program is a VMA or EMA program and its VMA or EMA size.

```
$EMA (BIG,0)
     PROGRAM VMST
     COMMON/BIG/IARRAY (12500)

     CALL VMAST (IVMA,ISIZE)
     IF (IVMA) 200,50,100
     .
     .
50 WRITE (1,'("EMA PROGRAM")')
     .
     .
100 WRITE (1,'("VMA PROGRAM")')
     .
     .
200 WRITE (1, '("NOT VMA/EMA PROGRAM")')
```

RteExtendedEV (Check EMA/VMA Capability)

The RteExtendedEV routine checks if the CPU has Extended EMA/VMA capability.

extended_ucode = `RteExtendedEV()`

`logical*2 RteExtendedEV,`*extended_ucode*

where:

extended_ucode is .TRUE. if the CPU has Extended EMA/VMA microcode.

An Extended-model application may call this routine prior to EMA/VMA access in order to determine whether or not Extended EMA/VMA is available. If not, the application can then issue an error and gracefully decline to continue execution. An attempt to access Extended EMA/VMA on a CPU that does not have the appropriate firmware receives an EM80 or VM80 violation.

Extended EMA/VMA capability is determined by the following:

1. The CPU must be an A990.
2. The firmware revision ID (product #0) must be 9 or greater.
3. If the firmware is revision 9, then microcode of revision 10 or greater must be downloaded into control store. The DOWNLOAD program must have been run to download the file REV10UPGRADE.MIC after the system was booted. The DOWNLOAD program sets RTE entry point $A990_CSID to the revision of microcode downloaded into control store. RteExtendedEV checks that entry point for a nonzero value.

I/O Management Subroutines

VMAIO (Perform Large VMA or EMA Data Transfers)

The VMAIO subroutine allows your program to perform large (up to 32 pages) I/O transfers to/from the VMA or EMA and any I/O device.

For a non-class I/O operation (*ecode* equals 1 or 2), the format is:

```
CALL VMAIO(ecode,cntrl,ibuff,ilen[,pram3[,pram4]])
```

For a class I/O operation (*ecode* equals 17, 18 or 20), the format is:

```
CALL VMAIO(ecode,cntrl,ibuff,ilen,pram3,pram4,class[,uv[,keynum]])
```

Finally, for a class Get operation (*ecode* equals 21), the format is:

```
CALL VMAIO(ecode,class,ibuff,ilen[,rtn1[,rtn2[,rtn3[,uv]]]])
```

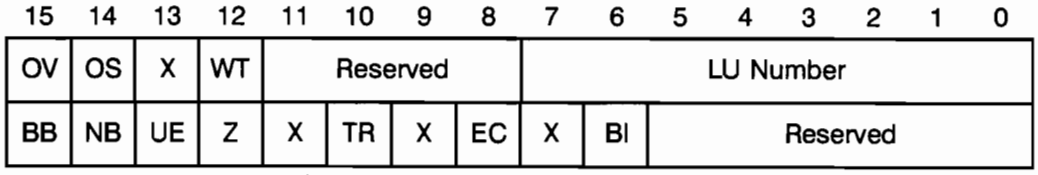
where:

ecode is any of the following standard EXEC I/O request codes:

1 Read
2 Write
17 Class Read
18 Class Write
20 Class Write/Read
21 Class Get

No other values should be used for *ecode* with VMAIO.

cntrl is a two-word quantity with the following format:



LU Number (Bits 0-7)
is the logical unit number of the device that data is to be transferred to or from.

Bits 6-13 and 15 of word 2 of CNTRL are identical to the corresponding bits of the EXEC 1 or 2 *cntwd*.

NB forces nonbuffered operation. For *ecode* equal to 1 or 2, NB should be set to 0 because all requests are unbuffered. For *ecode* equal to 17, 18, 20, or 21, NB is set as follows:

0 Request is buffered. If the value of ILEN is greater than the amount of SAM in the system, an IO04 error occurs.

1 Request is nonbuffered.

ibuff is a two-word quantity in double integer format (most significant word first, least significant word last). This represents the VMA/EMA word offset to the start of the buffer to be transferred.

Note This offset is automatically set up by the FORTRAN compiler when the call to VMAIO is made and the buffer is in the VMA/EMA area. Refer to the *FORTRAN Reference Manual*, part number 92836-90001.

---

ilen is 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 32767 characters. Use a positive number of words should this situation arise. Note that the length of *ibuff* cannot be greater than the EMA or Working Set size.

Note The maximum transfer length is 32 pages, or 100000B words. A value of 100000B in this parameter is treated as a positive word count rather than a negative character count. Remember that 32 pages can only be transferred when the transfer starts at the beginning of a physical page. It is, however, always possible to transfer up to 31 pages.

---

pram3 is an optional parameter or optional buffer, as in the EXEC 1 or 2 call.

pram4 is an optional parameter or optional buffer length.

class is the class number. This parameter is the same as the *class* parameter for a Class I/O EXEC call.

uv is the user-defined variable returned from a previous read or write.

keynum is the key number of the locked LU. The key number is returned in the *keynum* parameter of the LURQ call. See Chapter 2 for more details.

rtn1 corresponds to *pram3* for a read or write call.

rtn2 corresponds to *pram4* for a read or write call.

rtn3 is the request code passed to the driver on an initial read or write as follows:

1 Call was 17 or 20 (Read or Write/Read)
2 Call was 18 (Write)

Because a VMA or EMA variable cannot be used as a parameter in an EXEC call, VMAIO is available to handle I/O for a VMA or EMA buffer. The VMA or EMA I/O buffer to be transferred need not be mapped into the user logical address space prior to the request. The buffer is automatically mapped. The only requirement is that the number of pages required to map the buffer must be less than 32.

Two restrictions apply if a nonbuffered class VMAIO request is made.

1. If the program uses EMA, then EMA accesses may be made during the I/O, but the actual EMA buffer area within which I/O is proceeding must not be modified until the I/O is complete or unpredictable results will occur.

2. If the program uses VMA, then either no VMA accesses may be made during the I/O, or the pages that hold the buffer must be locked into memory prior to the VMA accesses. If you perform VMA accesses without locking the buffer space, unpredictable I/O results will occur.

Error messages that occur due to execution of the VMAIO subroutine are contained in Appendix A.

If the calling program is not a VMA or EMA program, the program will abort with an EM81 error unless the noabort bit is set.

Example: Read from the terminal into a VMA array.

```
$ALIAS VMAIO,NOABORT
$EMA (BIG,0)
      PROGRAM VMIO
      COMMON/BIG/IARRAY (50000)
      DIMENSION ICNWD (2)
C
C  READ FROM THE TERMINAL
C
      ICODE=1+100000B
      ICNWD(1)=1
      ICNWD(2)=0
      ILEN=250
      CALL VMAIO (ICODE,ICNWD,IARRAY,ILEN,*100)
C
C  RETURN HERE IF NO ERROR
      :
 100  ERROR PROCESSING ROUTINE
```

EIOSZ (Determine Maximum Length of Transfer)

The maximum guaranteed buffer length possible for a large VMA or EMA I/O transfer can be obtained by calling the subroutine EIOSZ. In RTE-A, the size returned is always 100000B words. This routine is provided for RTE-6/VM compatibility.

```
CALL EIOSZ(isize)
```

or

```
isize = EIOSZ()
```

where:

isize is set to the maximum length available in words. This value is always 100000B. The maximum length is also returned in the A-Register if the call is successful. This routine need never be called. It is provided for RTE-6/VM compatibility. On return, the A-Register = −1 if an error occurs (that is, not VMA or EMA program). If the call is successful, the A-Register will indicate the maximum length available for transfer in words (100000B).

Example: Determine the maximum buffer length available to transfer the VMA array using the VMAIO subroutine.

```
$EMA (BIG,0)
      PROGRAM ESZ
      COMMON/BIG/IARRAY (500000)

      CALL EIOSZ(ISIZE)

C  USE VMAIO TO TRANSFER DATA WITH
C  BUFFER LENGTH <= ISIZE
```

LOCKVMA, LOCKVMABUF, LOCKVMA2BUF (Lock VMA Pages/Buffers)

These routines lock and unlock pages and buffers in virtual memory.

```
error = LOCKVMA(ipages,inumpg)

error = LOCKVMABUF(ibuff,ilen)

error = LOCKVMA2BUF(ibuf1,ilen1,ibuf2,ilen2)
```

where:

ipages is the name of an array containing the page numbers of the virtual pages to be locked. IPAGES can contain a maximum of 64 page numbers. Each page number in the array is unsigned and one word in length and in the range 1 through 64.

inumpg is the number of pages specified in IPAGES.

ibuff, *ibuf1*, and *ibuf2*
are two-word quantities in double integer format (most significant word first, least significant word last) that represent the VMA offset to the start of the buffer to be transferred. Note that this offset is set up automatically by the FORTRAN compiler when the VMA lock call is made and the buffer is in the VMA area. (Refer to the FORTRAN Reference Manual.) The value must be in the defined range of virtual memory unless the associated buffer length is set to zero.

ilen, *ilen1*, and *ilen2*
: are one-word buffer lengths. A positive value indicates the number of words in the buffer and a negative value indicates the number of bytes in the buffer. (100000B is equal to 32768 words.) A buffer cannot be larger than 32 pages.

error
: indicates whether or not the call was successful. The possible values for *error* are as follows:

0	The pages/buffers were locked successfully
−82	The specified page numbers or buffer is out of bounds
−89	*ipages* is not in the range 0 to 64, inclusive, or the specified buffer is too large.

All pages can be unlocked by calling LOCKVMA with *ipages* equal to zero, LOCKVMABUF with *ilen* equal to zero, or LOCKVMA2BUF with *ilen1* and *ilen2* equal to zero.

A call to any of these three routines unlocks all previously locked pages.

Any error return unlocks any previously locked pages.

If virtual memory will be accessed while I/O is in progress, virtual pages must be locked before no wait VMAIO (using VMA) is performed.

When using LOCKVMA, the first *inumpg* page numbers in *ipages* must be in defined virtual memory. Only these pages are checked for validity.

For double buffering using VMAIO and VMA, both buffers must be locked so that pages from one buffer are not paged out when I/O is started on the second buffer.

A program that uses a VMA lock routine and terminates serial reusable must unlock all pages either immediately after beginning execution and before accessing VMA, or before terminating serial reusable. If this is not done and the program is re-executed, VMA pages may be left locked from a previous VMA lock call.

A program should not lock more than the number of pages in its working set minus one. When a page fault occurs, the page that caused the fault and the next page are brought into memory. If the program has locked exactly the number of pages in its working set, there may not be space for both pages (a VM83 error occurs).

Shareable EMA Subroutines

LKEMA/ULEMA (Lock/Unlock a Shareable EMA Partition)

The subroutines LKEMA and ULEMA allow a primary shareable EMA partition (that was specified at link time) to be locked and unlocked by the calling program. Locking a shareable EMA partition ensures that the partition is 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 as a primary area.

```
CALL LKEMA
  or
CALL ULEMA
```

These subroutine calls are ignored if the program 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 not locked, the call is also ignored.

Normally, a shareable EMA partition is released once the number of programs actively using it drops to 0. 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.

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,0)
      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,0)
      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
```

RteAllocShema (Attach a Secondary SHEMA)

The RteAllocShema routine is called by a Large or Extended model program to attach a secondary shared EMA (SHEMA) to itself. The calling sequence is:

error = RteAllocShema(*shemalabel*,*pagesize*,*startaddr*,*flags*)

```
character   shemalabel*16
integer*2   pagesize
integer*4   startaddr, flags
integer     error
```

where:

shemalabel is a character string that specifies the label of the SHEMA. Up to the first 16 characters of the label are significant.

pagesize is a single word integer specifying the number of EMA data pages desired. For Large model programs, this value cannot be greater than 1023; for Extended model programs, it cannot be greater than 32735.

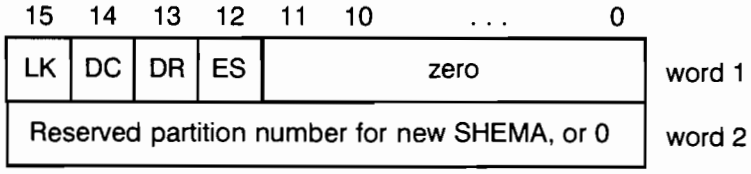
If the SHEMA partition has not yet been allocated, it will be created large enough to accommodate this many data pages (plus overhead for page tables). Normally, this value must be less than or equal to the number of EMA data pages accomodated by the existing SHEMA partition, if it has already been allocated. However, if the ES bit is set in the *flags* word and the partition has already been allocated, then this parameter returns the existing size of the SHEMA.

startaddr is the double-word starting EMA address at which the SHEMA should be attached in your program's EMA address space. This value may name any address in the 1024 pages of the first EMA segment to which the SHEMA should be attached.

error is a single-word integer indicating the error return as follows:

0	SHEMA successfully attached
−1	The calling program is not Large or Extended model EMA/VMA
−2	The attachment address is invalid
−3	The named SHEMA is already initialized for a different model
−4	The pagesize parameter is invalid or larger than the existing SHEMA
−5	An attachment EMA segment is already in use; *startaddr* is wrong
−6	Insufficient free XSAM for a new SHEMA table or SHEMA Association Block (SAB)
−7	Named SHEMA is already in use by 255 programs
−8	The reserved partition number in *flags* is out of range
−9	The RTE memory manager failed to allocate memory for the SHEMA
−10	SHEMA existence conflicts with *flags* specified
−11	The SHEMA is initialized for a different EMA segment (Large model only)
> 0	The A- and B-Registers contain a 4-character abort code returned by the RTE-A memory manager when memory allocation for the SHEMA was attempted. Usually this is the "SC09" error − insufficient memory for the requested SHEMA size.

flags is a double-word integer with the following bit definitions:



where:

LK = 1	if the SHEMA partition should be "locked", such that the partition is not released when the in-use count = 0.
DC = 1	if a new SHEMA should not be created; error −10 is returned if the named SHEMA does not exist.
DR = 1	if an existing SHEMA should not be reused; error −10 is returned if the named SHEMA already exists.
ES = 1	if your program wishes to adopt an existing SHEMA size. If the SHEMA partition has already been allocated, then the same number of EMA data pages for which the partition has been created will be claimed from the calling program's EMA space. In this case, the value of *pagesize* is ignored, and the existing size is returned in that parameter. If the partition has not yet been allocated then it will be created now; the value of *pagesize* dictates the size of the new partition as usual.

A SHEMA table entry for the named SHEMA is created if none exists. The SHEMA partition is allocated and the page tables in that SHEMA partition are initialized if not already.

Usually, the name of the first variable in a common block relocated to the proper EMA segment is given for the *startaddr* parameter. For more information, see the discussion of the ES command in the *RTE-A LINK User's Manual*, part number 92077-90035.

A program calling RteAllocShema must have a Large or Extended model "primary" EMA or VMA defined, which may itself be a shared EMA. The attachment EMA/VMA address must be above the last address used by the primary area. For example, a program that uses the default VMA size of 8192 pages must attach any secondary SHEMAs at EMA segment 9 or above.

A SHEMA attached in this manner is automatically "returned" to the system if the calling program aborts. The program may also request that the SHEMA be detached programmatically by calling the RteReturnShema routine.

When RteAllocShema returns, user map page 31 is undefined.

RteReturnShema (Detach a Secondary SHEMA)

The RteReturnShema routine is called by a Large or Extended model program to detach a secondary SHEMA that has been allocated via RteAllocShema. The calling sequence is:

error = `rtereturnshema(`*startaddr*`,`*flags*`)`

```
integer*4 startaddr
integer*2 flags, error
```

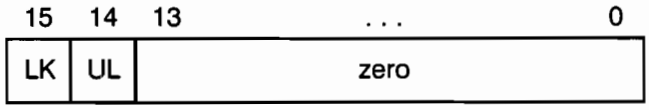
where:

startaddr is the double-word starting EMA address at which the SHEMA was attached in your program's EMA address space.

error is a single-word integer indicating the error return as follows:

0	SHEMA successfully detached
−1	The program is not Large or Extended model EMA/VMA
−2	The attachment address is invalid or does not have a secondary SHEMA attached to it

flags is a single-word integer with the following bit definitions:



where:

LK = 1 if the SHEMA should be "locked", such that the partition is not released when the in-use count equals 0 and the SHEMA table entry is not released when the in-system count equals 0.

UL = 1 if the SHEMA should be "unlocked", such that the partition is released when the in-use count equals 0 and the SHEMA table entry is released when the in-system count equals 0.

RteReturnShema detaches the SHEMA from each EMA segment of the calling program's EMA/VMA address space to which it was attached, freeing those EMA segments for reuse in a subsequent RteAllocShema call, if desired. After this call successfully completes, any attempt to use an EMA/VMA address that formerly referenced the returned SHEMA will cause an EM82 or VM82 error. If the same SHEMA has been attached multiple times to the calling program, then

only the attachment specified by *startaddr* is returned. However, any data items in the SHEMA that were mapped into your program's MSEG/VSEG area before the call are unmapped. Therefore, they must be remapped using the alternate attachment point before use.

The in-use and in-system counts for the returned SHEMA are decremented. If the SHEMA is unlocked and the in-system count reaches zero, the SHEMA partition is deallocated. If the SHEMA is unlocked and the in-use count reaches zero, the SHEMA table entry is deallocated.

When RteReturnShema returns, data map page 31 is undefined. Additionally, any pages in the program's data map that pointed to the partition for the SHEMA being returned are unmapped and are set to be read and write protected.

RteRenameShema (Rename SHEMA Label)

This routine renames an existing SHEMA label to a new name. The SHEMA may be in use at the time it is renamed. The calling sequence is:

```
error = RteRenameShema(oldname,newname,flags)

character oldname*16, newname*16
integer   flags, error
```

where:

oldname is a character string containing the name of the SHEMA to be renamed.

newname is a character string containing the new name to be assigned to the SHEMA.

flags is a single-word integer, which currently must be zero.

Error returns are as follows:

0	SHEMA successfully renamed
−1	Old SHEMA name not found
−2	New SHEMA name is already in use (note that it is not an error to "rename" a SHEMA to its own existing name)

This routine changes the label in the SHEMA table entry for the named SHEMA. Once a SHEMA is successfully renamed, all further access to the SHEMA must be made under the new name. This routine may be called by non-EMA/VMA programs.

RtePrimeShInfo (Return Primary SHEMA Information)

The RtePrimeShInfo routine returns primary SHEMA information about a SHEMA-only program. This routine may be called by a Normal, Large, or Extended model program.

```
error = RtePrimeShInfo(label,locked)

character*16 label
logical*2    locked
integer*2    RtePrimeShInfo,error
```

where:

label returns the 16-character label of the primary SHEMA.

locked returns .TRUE. (100000b) if the SHEMA is locked, otherwise returns .FALSE. (0b).

error returns:
- 0 if no errors occurred.
- −1 if the program is not a SHEMA-only program.

The B-Register returns the size of the Primary SHEMA partition, including all PTE pages and EMA data pages.

VMA File Subroutines

The demand-paged virtual memory system transparently manipulates the working set and backing store file for you; however, VMA file subroutines are available to allow you to create, open, or close the virtual memory backing store file with various options. This allows your program to 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 you do not specify a VMA backing store file, one will be created for you. The directory on which it is created is chosen in the following manner.

1. If the system scratch cartridge number ($SCRN) is not zero, the VMA backing store file is created there.

2. If the scratch cartridge number is zero and the directory /SCRATCH/ exists, the VMA backing store file is created in the directory /SCRATCH/.

3. If the scratch cartridge number is zero and the directory /SCRATCH/ does not exist, the VMA backing store file is created in your current working directory unless your current working directory is zero. If your working directory is zero, the VMA backing store file is placed on the first FMGR cartridge having sufficient space.

Default VMA backing store files on CI volumes are named as follows:

VM*cnnn*.VMA

Default VMA backing store files on FMGR cartridge are named as follows:

*nnnc*VM

where:

nnn is the ID segment number of the VMA program.

c is the CPU number (value of the $CPU entry point).

The default backing store file is type 2, with 256-block increments. The backing store file is automatically purged.

Using the VMA file subroutines, you can declare an existing file as the backing store file. This provides you with the option of having initialized virtual memory (refer to the VMAOPEN subroutine). In addition, you can declare the backing store file to be read only in order to prevent inadvertent changes to the file. The VMA file cannot be created via DS on a remote node.

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.

VMAOPEN (Open a VMA Backing Store File)

The VMAOPEN subroutine opens the named backing store file with the specified options.

```
CALL VMAOPEN(ierr,name,ioptn)
```

where:

ierr is error return. A zero is returned to indicate a successful call.

name is the filedescriptor, a character string of 63 or fewer characters. Any DS information is ignored.

ioptn are file options. A character string list of one-letter options (uppercase or lowercase) selected from the following set:

R Open for reading
W Open for writing
O OK to open an existing backing store file
C OK to create a new backing store file
S Open shared
U Open in update mode
X OK to access an/or create extents
T File is temporary
V Defer create/open until required

If the backing store file already exists, it must be a type 2 file, or an error is returned. If the backing store file does not exist and the C option is specified, a type 2 file is created with the record length equal to 1024 words. If the X option is also specified, as many extents as necessary are created to contain the VMA array. Files opened by the VMAOPEN subroutine should be closed with the VMACLOSE subroutine. If the contents of the working set is to be written to the file, use the VMAPOST or VMACLOSE subroutine described later in this chapter. A backing store file may be opened only once within a program; after the backing store file has been closed or purged, any further attempts to use VMAOPEN cause the program to abort.

To create a backing store file that is automatically purged after the program terminates, the temporary and create options must be specified (TC). If the T option is specified, the shared open option (S) cannot be specified. Other options may be used.

If the create (C) option and the temporary file (T) option are specified, and the name is all blanks, then a default VMA backing store file is created as specified in the last section. To force the creation of a default backing store file when the program starts instead of deferring its creation until the working set is full, use the blank name without the defer (V) option.

If the extents (X) option is not specified, the size of the backing store file is created with sufficient size to hold the number of pages specified by the program's virtual size (VS). If you specify a file size parameter, it must be at least equal to the program's virtual size.

If the extents option is specified, the initial size of the default backing store file equals the number of blocks required for 1/256th of the number of pages specified by the program's VS size, or 32 pages, which ever is larger. If you specify a file size, it must be at least this large.

Note that if the backing store file is created on a FMGR cartridge, it is automatically purged only if VMAOPEN is allowed to form the name.

Temporary files on a CI volume closed by VMACLOSE are not automatically purged. You can make a temporary file permanent by opening the file without specifying the T option.

To clean up after a system failure, you can use the masking T option with the PU command (PU @.@.T).

The backing store file is considered to be initialized only if opened in the update mode (the U option of the IOPTN parameter), thus allowing you 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 option was not specified, 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 disk.

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 shows up when creation of the backing store becomes necessary.

Error Returns: When an error occurs during the subroutine call, a negative error code is returned in the *ierr* parameter and in the A-Register.

Example: Open the VMA backing store file called TSTDTA when required for updating. If the file does not exist, create it.

```
$EMA (BIG,0)
      PROGRAM VMTS1
      COMMON/BIG/IELEMB (65535)
      CHARACTER INAM*63, IOPTN*4
      INTEGER*4 IELEMB
      DATA INAM/'TSTDTA::VMADIRECT'/
C
C   OPEN THE VMA FILE.
C   SET IOPTN FOR UPDATE MODE AND
C   CREATE (WITH EXTENTS) IF NECESSARY
C
      IOPTN = 'CUXO'
      IERR = 0
C
      CALL VMAOPEN(IERR,INAM,IOPTN)
      IF (IERR.NE.0) GOTO 300

C   MANIPULATE THE VMA ARRAY IN THE FILE TSTDTA

      300                        ! error message
```

VMAPURGE (Purge VMA Backing Store File)

The VMAPURGE subroutine can be called to purge the backing store file.

```
CALL VMAPURGE
```

VMAPURGE purges a file opened or created and opened by VMAOPEN, or a backing store file created by the system.

If called prior to the opening of a backing store file, VMAPURGE will take no action.

VMAPURGE can be used only on an open backing store file. Once VMAPURGE has been called after the opening of the backing store file, further VMA accesses are illegal (the program will abort when the system attempts to access the backing store file).

Example: Purge the VMA file created by the VMAOPEN call.

```
$EMA (BIG,0)
      PROGRAM VMTS2
      COMMON/BIG/IELEMB (65535)
      CHARACTER INAM*63
      INTEGER*4 IELEMB

C   CREATE VMA FILE
      .
      .
C   MANIPULATE FILE
      .
      .
      .
      CALL VMAPURGE     ! PURGE THE SCRATCH VMA FILE
                        ! NO FURTHER VMA ACCESSES ARE LEGAL
```

VMAPOST (Post Working Set to Disk)

The subroutine VMAPOST 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 (all virtual pages in memory will still be in memory). If the file is opened with the read-only option, no posting will occur.

```
CALL VMAPOST
```

If you opened or created your own backing store file using the VMAOPEN subroutine, the VMAPOST or VMACLOSE 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 disk.

VMACLOSE (Close the VMA Backing Store File)

The VMACLOSE subroutine posts all pages of the working set in memory to the VMA backing store file on disk and executes an FMPCLOSE on the VMA backing store file. If the default VMA backing store file is opened or created on the system scratch cartridge or your working directory, it is purged after program completion.

```
CALL VMACLOSE
```

If you opened or created your own backing store file using the VMAOPEN subroutine, the VMAPOST or VMACLOSE 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 disk. If the file is opened with the read-only option, no posting occurs.

VMACLOSE can be used only on an open backing store file. Once VMACLOSE has been called after the opening of the backing store file, further VMA accesses are illegal and will cause the calling program to abort.

Example: Close the VMA file opened by the VMAOPEN call.

```
$EMA (BIG,0)
     PROGRAM VMTS3
     COMMON/BIG/IELEMB (65535)
     CHARACTER INAM*63
     INTEGER*4 IELEMB

     .                    ! OPEN THE VMA FILE
     .                    ! UPDATE THE FILE
     .
     CALL VMACLOSE        ! CLOSE THE VMA FILE
     .
     .
```

VMAREAD (Read Data from a File into VMA/EMA)

The VMAREAD subroutine allows your program to read records from a data file into a VMA or EMA array. This subroutine is similar to the FMPREAD call.

ilen = VMAREAD(*idcb*, *ierr*, *iarray*, *idl*)

where:

ilen is the data length read (in bytes). A one-word variable in which the actual number of bytes read or a negative error code is returned. If more than 32767 bytes were read, the returned length will be negative; in such cases, the error variable should be checked.

idcb is the 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 is 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
-243 = request parameter error
-244 = VMA/EMA mapping error
  <0 = FMP error
```

iarray is the 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 VMAREAD is made and the array is in the VMA/EMA area. This value must be positive.

idl is the data length requested (in bytes). A one-word variable that specifies the number of bytes to be read. If the file is not type 1, the length of the request can not exceed the minimum of the size of MSEG and the size of the working set. This parameter is the same as the maxlength parameter of the FMPREAD call.

Type 1 files and large MSEGs 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 EMA array of the program.

```
$EMA (AREA,0)
      PROGRAM VMTS4
      COMMON/AREA/IARRAY (200,200)   !IARRAY IS AN EMA ARRAY
      CHARACTER INAM*63
      DIMENSION IDCB (144)
      DATA INAM/'FNAME'/
C
C   OPEN FILE FNAME
C
      FTYPE = FMPOPEN (IDCB,IERR,INAM,'RO')
      IF (IERR.LE.0) GO TO 100
      :
C
C   READ FROM FILE INTO IARRAY
C
      ILEN = VMAREAD(IDCB,IERR,IARRAY,256)
      IF (IERR.NE.0) GO TO 100
      :
C
C   CLOSE FILE FNAME
C
      IERR = FMPCLOSE(IDCB,IERR)
      IF (IERR.NE.0) GO TO 100
      :
      STOP
  100 WRITE (1,900) IERR             !ERROR HANDLER
  900 FORMAT ("ERROR CODE = ",I5)
      :
```

VMAWRITE (Write Data from VMA/EMA to a File)

The VMAWRITE subroutine allows your program to write a record of data from a VMA or EMA array into a data file. This subroutine is similar to the FMPWRITE call.

ilen = VMAWRITE(*idcb*, *ierr*, *iarray*, *idl*)

where:

idcb is the 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 is the 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 are returned in *ierr*:

0	= normal return
−243	= request parameter error
−244	= VMA/EMA mapping error
< 0	= FMP error

iarray is the 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 VMAWRITE is made and the array is in the VMA/EMA area. This value must be positive.

idl is the data length requested. A one-word variable that specifies the number of bytes to be written. If the file is not type 1, the length of the request cannot exceed the minimum of the size of MSEG and the size of the working set. This parameter is the same as the maxlength parameter of the FMPWRITE call.

ilen is the number of bytes actually transferred, or a negative error code. If more than 32767 bytes are transferred, the returned length will be negative; in such cases, the error variable should be checked.

Type 1 files and large MSEGs 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 array, IARRAY, to the file FNAME.

```
$EMA (AREA)
      PROGRAM VMTS4
      COMMON/AREA/IARRAY (20000) !IARRAY IS A VMA ARRAY
      CHARACTER INAM*63
      DIMENSION IDCB (144)
      DATA INAM/'FNAME'/
C
C  OPEN FILE FNAME
C
      CALL FMPOPEN(IDCB,IERR,INAM,'RWOX',1)
      IF (IERR.LE.0) GO TO 100
      :
C
C  WRITE VMA IARRAY TO FILE
C
      ILEN = VMAWRITE(IDCB,IERR,IARRAY,256)
      IF (IERR.NE.0) GO TO 100
      :
C
C  CLOSE FILE FNAME
C
      CALL FMPCLOSE (IDCB,IERR)    !CLOSE FILE FNAME
      IF (IERR.NE.0) GO TO 100
      :
      STOP
 100  error processing
```

Example 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,0)
       PROGRAM VMAEX (),Illustrates VMA File Manipulation

C   This program opens the existing file VMDATA (type 2, 1024 words
C   per record) as the backing store file, and allows you to
C   add data from a data file (type 1) to the VMA backing store file.
      COMMON/BIG/IARRAY (2048),JARRAY (2500)
      DIMENSION IDCB (144)
      CHARACTER INAM*63, JNAM*63, IOPTN*5
      DATA INAM/'VMDATA'/, JNAM/'DATA'/
C
C   Open the backing store file in update mode
C
      IOPTN = 'RWUOX'
      CALL VMAOPEN(IERR,INAM,IOPTN)
      IF (IERR .NE. 0) GO TO 1000
C
C   Open the data file
C
      IOPTN = 'ROS '
      CALL FMPOPEN(IDCB,IERR,JNAM,IOPTN,1)
      IF (IERR .LE. 0) GO TO 1000
C
C   Read from the data file into the VMA array JARRAY
C
    50 ILEN = VMAREAD (IDCB,IERR,JARRAY,256)
      IF (IERR .EQ. -12) GO TO 100       ! check for end of file
      IF (IERR .NE. 0) GO TO 50
C
C   Here the end of the data file was reached.
C   Close the backing store and data files
C
    100 CALL VMACLOSE
      CALL FMPCLOSE (IDCB,IERR)
      IF (IERR .LT. 0) GO TO 1000
      STOP
C
C   Error processing here
C
 1000 WRITE (1,'("VMA file error ", I6)') IERR
      STOP
 1200 WRITE (1,'("FMP error ", I6)') IERR
      END
```

FMGR VMA File Routines

The following VMA file routines allow your program to manipulate the backing store file. These calls apply only for FMGR files. Note that these calls cannot be used in Large or Extended model programs. They are provided for compatibility with existing Normal model programs only.

CREVM (Create a VMA Backing Store File)

Your 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)
```

where:

name is the file name. A three-word array containing the ASCII name of the file to be created.

ierr is for error return. A one-word variable that contains a zero for successful calls.

ioptn specifies the file options. The file options can be set as follows:

Bit 0 = 0 *name* and bit 1 are ignored.

Bit 0 = 1 A non-scratch file (file *name*) is to be created and used as the backing store file.

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 is the file security code.

icr is the file cartridge reference number (CRN).

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 only.

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 negative error code is returned in the *ierr* parameter and in the A-Register. The returned error code may be in the negative 200 range because VMA is able to access CI volume files.

OPNVM (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.

```
CALL OPNVM(name,ierr,ioptn,isc,icr)
```

where:

name is the file name. Three-word array containing the ASCII name of the file to be opened.

ierr is error return. A zero is returned to indicate a successful call.

ioptn specifies the file options. The file options can be set as follows:

If bit 0 = 1, the file is to be opened for non-exclusive use.

If bit 1 = 1, the file is to be opened for update (refer to the comments below).

If bit 2 = 1, the file open is to be deferred until required.

If bit 3 = 1, the file extents are not to be addressed or created.

If bit 4 = 1, the disk file is opened for read-only access.

isc is the file 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 is the file cartridge reference number (CRN).

The backing store file opened by the OPNVM subroutine must be a type 2 file, or an error will be returned. Files opened by OPNVM 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 chapter.

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 your program 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 disk.

Normal and Error Returns: Normal and error returns are the same as those for the CREVM subroutine.

PURVM (Purge VMA Backing Store File)

The subroutine PURVM can be called to purge the backing store file.

```
CALL PURVM
```

PURVM purges a file opened by OPNVM, a file opened or created and opened by CREVM, or a backing store file created by the system.

If PURVM is called prior to the opening of any backing store file, the subroutine takes no action.

Once PURVM has been called after the opening of the backing store file, further VMA accesses are illegal (the program will abort when the system attempts to access the backing store file).

PSTVM (Post Working Set to Disk)

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 (all virtual pages in memory are still in memory). If the file is opened with the read-only option, no posting occurs.

```
CALL PSTVM
```

If you opened or created your own backing store file using 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 disk.

CLSVM (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 disk 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 after program completion.

```
CALL CLSVM
```

If you opened or created your own backing store file using 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 disk. If the file is opened with the read-only option, no posting will occur.

VREAD (Read Data from a File to a VMA/EMA)

The VREAD subroutine allows your program to read records from a FMGR data file into a VMA or EMA array. This subroutine is similar to the FMGR READF call.

```
CALL VREAD(idcb,ierr,iarray,idl[,ilen[,inum]])
```

where:

idcb is the 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 is the 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 are returned in IERR:

```
   0  = normal return
-243  = request parameter error
-244  = VMA/EMA mapping error
 < 0  = FMP error
```

iarray is the 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.

idl is the 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 minimum of the size of MSEG and the size of the working set. For type 1 files only, the *idl* parameter is considered unsigned (no sign bit) to allow for a maximum data length of 65535.

ilen is the 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 is the 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 READF call.

Type 1 files and large MSEGs 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.

VWRIT (Write Data from VMA/EMA to a File)

The VWRIT subroutine allows your program to write a record of data from a VMA or EMA array into a FMGR data file. This subroutine is similar to the FMGR WRITF call.

```
CALL VWRIT(idcb,ierr,iarray,idl[,inum])
```

where:

idcb is the 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 is the 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 are returned in *ierr*:

0	= normal return
−243	= request parameter error
−244	= VMA/EMA mapping error
< 0	= FMP error

iarray is the 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.

idl is the 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 minimum of the size of MSEG and the size of the working set. This parameter is the same as the IL parameter of the WRITF call for values greater than or equal to zero. For type 1 files only, the *idl* parameter is considered unsigned (no sign bit) to allow for a maximum data length of 65535.

inum is the 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 WRITF call.

Type 1 files and large MSEGs 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.

VMA/EMA Mapping Management Subroutines

This section 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 9-3 summarizes the VMA/EMA mapping subroutines described in this chapter. These subroutines are in firmware on RTE-6/VM and RTE-A Systems.

As discussed previously, 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 logical map as shown in Figure 9-3. The software mapping routines use the MSEG to map data into logical memory as shown in Figure 9-2.

The VMA/EMA Mapping Management Subroutines may modify the A-, B-, E-, O-, X-, and Y-Registers because the microcode uses these registers to pass page fault information to the VMA handler. The computer reference manual for your CPU contains details about how the registers are used.

Table 9-3. 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.

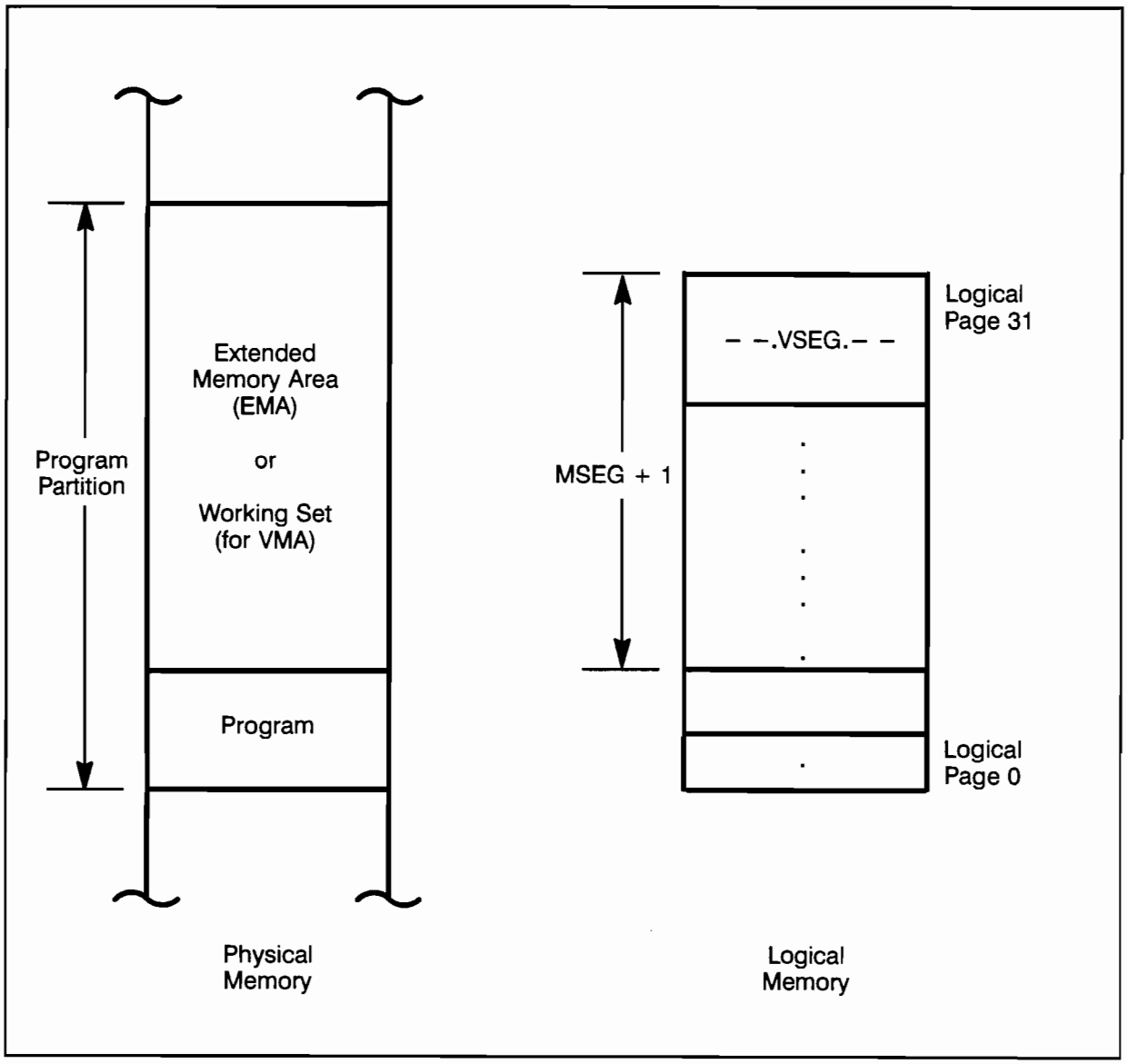


Figure 9-2. VMA/EMA and Memory Structure

.IMAP

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.

.IMAP maps in the page containing the element and the following page, if the following page is in the VMA or EMA. This allows for multi-word array elements. If the element is in the last page of VMA or EMA, that physical page will be mapped through the first page of the VSEG, and the second page of the VSEG will be set read/write protected.

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 undefined on return.

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 16-31) in offset word 1 and the low bits (bits 0-15) 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

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 VMA 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.

Any error detected causes the program to be aborted. Possible errors are described in Appendix A.

.JMAP

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. The possible errors are 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 is undefined on return.

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 software is called to swap 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

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.

Computer Museum

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 bit in the A-Register, least significant bit in the B-Register).

The TABLE for .JRES has the same contents and structure as the TABLE for .JMAP.

MMAP

MMAP is a subroutine that maps a buffer in VMA/EMA 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 10-19 in the 36th word of the ID segment). The maximum buffer size that may be requested with MMAP is the size of MSEG (31 minus the starting logical page of MSEG). If the working set size is less than this value, then the working set size minus 1 becomes the maximum buffer size that can be mapped.

MMAP will map one more page than the number of pages requested. This is done so that to map in a buffer, you need know only the VMA/EMA page number in which the buffer starts, and the size of the buffer rounded up to an integer number of pages. Thus, a two-page buffer starting at any location within a page in VMA/EMA will be guaranteed to be completely mapped in by an MMAP call specifying the starting page and NPGS value of 2.

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 0, MMAP will map in one page.

You 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 FORTRAN calling sequence is:

```
CALL MMAP(ipgs,npgs)
```

where:

ipgs is VMA/EMA page holding start of buffer to map (where the first page in VMA or EMA is page 0).

npgs is the number of pages (rounded up) in buffer 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* is 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

The .ESEG subroutine maps 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.)
RTN+1     normal return     Normal return point.
```

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 are mapped into logical memory and the B-Register equals the logical address of the starting page of MSEG. Any error causes the program to abort.

The maximum number of pages that can be mapped with .ESEG is MSEG size + 1.

.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 is the double-integer pointer (high word first) 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 is the double-integer pointer containing the virtual address.

ofset is the 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 *pontr* + *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 pre-RTE-6/VM EMA programs. The preferred method of doing I/O to VMA/EMA in RTE-A 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 MACRO/1000 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.

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 undefined.

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 is the name of the program.

address is 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).

10

CDS Programming

With the VC+ Enhancement Package (HP 92078A), the RTE-A Operating System can manage large programs. A Code and Data Separated (CDS) scheme is used to allow up to 7.9 MBytes of program code.

CDS Programs

The source code of a large CDS program does not appear segmented. In order to create a CDS program, this source code must be compiled or assembled with the CDS Compiler option. The CDS compiler option signals the compiler or assembler to create CDS relocatable code which contains separable code and data structures.

The CDS relocatable code is then linked. LINK creates a type 6 file for the program with the code and data separated. If there is more than 32K words of code, LINK can automatically create code segments. Data segments are always created, whether or not >32K. Procedure modules are placed in the different code segments. Each CDS program can have up to 128 code segments and one data segment. The maximum size of a code or data segment is 31 pages.

The code and data for a CDS program reside in different partitions. The code contains the machine instructions that are executed by the computer; the data is the information that is acted upon by the computer, according to the code instructions. The code does not change as the program executes; the data does.

When the program is executed, the code segments are loaded into a code partition and the data segment is loaded into the data partition. Two different Dynamic Mapping System (DMS) map sets are used for CDS programs: one for the code and one for the data. The VMA/EMA area is in the data partition, except for shareable EMA which is in its own partition.

Non-CDS programs (or programs that were compiled on other versions of RTE) can be reloaded for RTE-A and will execute in a data partition. Programs can often be changed to run in CDS mode without change to the source program, except the addition of the CDS compiler option.

CDS programs can be shared by several users at the same time. The code partition of a shared program is shared, while each copy of the program has its own data partition. Programs are specified as shared programs at link time via the LINK SP command.

Code Partition

The code partition of a program is an area of memory reserved by RTE-A for the program's code segments. See Figure 10-1 for an illustration of a code partition. The size of the code partition is fixed when the program is first dispatched and cannot be changed while the program is running.

The content of the code partition never changes during the execution of a program, but the data partition does. The code that is stored in memory is identical to the disk copy from which it was loaded. Because code partitions never change, they can be overlaid at any time and later restored by copying from the original on the disk. Data partitions, on the other hand, are subject to constant change during program execution, and must be saved on disk if their memory is needed by another program.

The code partition may contain only one code segment or all of the code segments for the program. The code partition is divided into code blocks. The size of each code block is at least the size of the largest code segment. One code segment from the type 6 disk file executes in each code block of the code partition. The default partition size is equal to the sum of all of the code blocks in the program. The LINK CD command or the CI CZ command can be used to modify the number of code blocks in the code partition. A CDS program may have more code segments than there are code blocks in its partition.

Each code partition has one page (page 0) dedicated for use by the system. This primarily contains information about which code segments are in memory, where they are in memory, and which code segments are on disk and where they are on disk. Refer to the *RTE-A System Design Manual*, part number 92077-90013, for a detailed discussion of the Code Partition Page 0.

If a program consists of five code segments that are 5, 5, 4, 3, and 2 pages in length, the code block size is 5 pages, large enough to hold the longest code segment. As a result, the program partition contains 5 code blocks of 5 pages each, plus one page of system information, or 26 pages. The number of code blocks in the program can be set when the program is linked or after the program is RP'd. Memory is used more efficiently if all of the segments are of roughly equal size.

If the code partition does not contain a code segment when it is needed, the segment must be loaded from the disk. If all of the code blocks in the partition are already full, one of them is overlaid by the segment that is loaded. For a shared program, all of the code segments must be present in the partition at all times during program execution. (Using the LINK SP command sets a bit in the ID segment which causes this to happen.)

At link time, you can declare any of your code segments to be memory locked. This means that these code segments will never be overlaid by another code segment. There must be at least one free code block for the non memory-locked segments.

Modules (for example, FORTRAN subroutines) are not permitted to cross segment boundaries. A subroutine can call another subroutine regardless of what segment the other subroutine is in. All parameters are passed through the stack.

The number of code blocks does not affect the logical operation of the program, but it does affect execution speed. The more code blocks a program has (up to the number of code segments it has) the faster it can run. This is because it is much slower to bring a code segment in from disk than to remap a code segment already in memory. The only advantage of having fewer code blocks than code segments is the saving of physical memory.

Data Partition

All of the data that a program uses is mapped into its logical data space. In addition, any non-CDS code that a CDS program references resides in the data partition. System common is mapped into the data partition logical memory, but does not physically reside in the data partition. Refer to Figure 10-1 for an illustration of a data partition.

Stack & Heap Area

The stack is that portion of the data partition used for parameter passing and procedure local variables. A procedure local variable is one which is used only within a particular procedure and is placed on the stack (procedure is synonymous with subroutine and function). Each time a CDS program makes a procedure call, a stack frame is created for the procedure to hold its parameters and variables.

The initial value of these kinds of variables is indeterminate. That is, upon entry to a procedure, space is allocated for them on the stack, and the initial value of the stack variables is whatever value happened to be in that location. This has implications for programmers who assume that local variables will have the same value upon re-entry to a subroutine. Some implementations of a language may support this and some may not.

For CDS programs, local arrays which are passed cannot exceed the stack frame address space of 1018 words. You must get these arrays out of the limited address space of the stack frame. In Pascal, this means the arrays should be in global space, while in FORTRAN, either labeled common or the SAVE statement will work. Also in FORTRAN, if the local array is completely above the 1K address limit of the stack frame, the compiler will compute and pass the data relative address to the subroutine. To do this, equivalence the data arrays in such a way that they are preceded by 1024 words. For example:

```
integer   fill(1024)
real      data(2000)
equivalence (fill(1024), data(1))
```

The fill array may be used for other functions as long as it is not passed into the subroutine.

The stack frame also contains links to the calling procedure. When a procedure returns to the calling procedure, its stack frame is returned to free space and the calling procedure resumes execution at the location following the call.

Heap space is that area of memory between the last memory location allowed for the stack and the end of memory (or MSEG, if any). This area is accessed from Pascal as the Heap 1 area; it is allocated by "new" statements and released by "dispose" statements. It is not recommended that this space be accessed from FORTRAN, however, its location can be determined using the LIMEM subroutine. FORTRAN programs that used LIMEM in earlier RTE's will, however, continue to work.

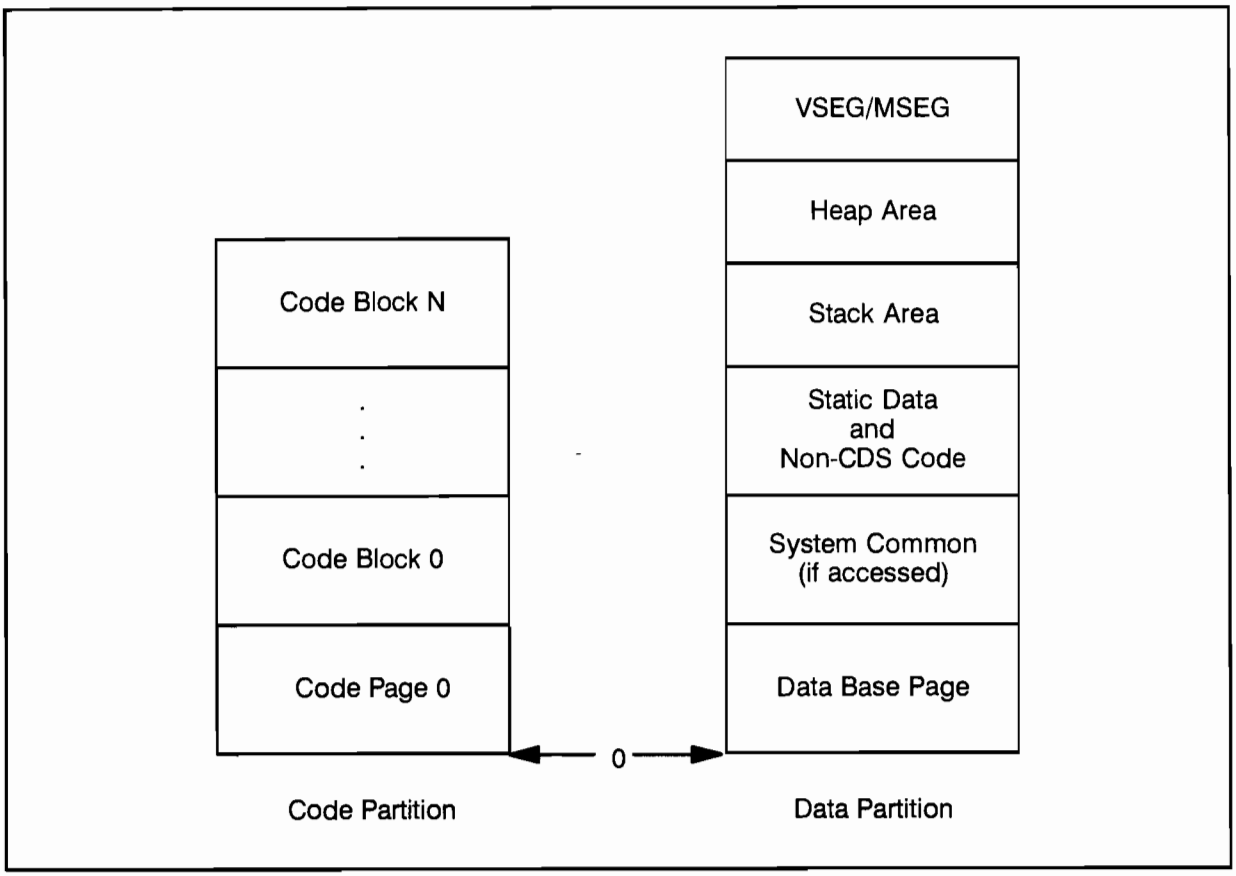


Figure 10-1. A CDS Program in Logical Memory

Refer to the *RTE-A System Design Manual*, part number 92077-90013, for a detailed description of the contents of the Code and Data Partition.

Mixing CDS Code and Non-CDS Code

CDS and non-CDS code can be mixed in the same FORTRAN program. However, certain restrictions apply. CDS code can call non-CDS code, but non-CDS code can never call CDS code. Once the non-CDS code is called, it cannot call CDS code, only other non-CDS code. When the non-CDS code completes, control returns to the CDS code.

One implication of this restriction is that programs must be converted from non-CDS to CDS in a top-down fashion. That is, the main must be converted before any subroutines which it calls, a subroutine must be converted before converting any subroutines which it calls, and so on.

Pascal programs must contain either all CDS or all non-CDS code.

Converting Programs to CDS

This section explains the main concerns with converting a FORTRAN or Pascal program to CDS. In most cases, the conversion is not difficult.

General Considerations

The compiler must be given the instructions to produce CDS relocatable code. This is in the form of the compiler option $CDS ON. Refer to the language reference manual for details.

Before starting conversion, assemble a set of test inputs and results which are known to be good and on which the program can be run after conversion to ensure compliance.

No Automatic Conversion

Conversions do not always work automatically, even though the program is written in standard FORTRAN or Pascal. Programmers often make use of implementation dependent features of the machine or language which they are using. This is because standards for languages allow the implementors of those languages to make certain decisions for themselves as to how features will work.

The behavior of local variables must be carefully considered for conversion to CDS. Local variables are those defined within a subroutine or function. They are not available to the main program or other subroutines. In FORTRAN, according to the ANSI standard, upon return from a subroutine, all the local variables become undefined unless specified in SAVE statements. When the subroutine is entered again, the value of these local variables depends on whether the program is CDS or not. In a non-CDS program on an HP 1000, they contain the same values that they had when the subroutine ended. In a CDS program, they contain whatever value is left on the stack. Also, in a CDS program, local variables are not necessarily initialized to zero, as they are in non-CDS programs.

FORTRAN Conversion

If the program to be converted relies on values being saved between invocations of subroutines, it is safest to take precautions. The precaution to take is to put a 'SAVE' statement in every suspect subroutine. This causes all the variables to be placed in static data space and means that their values will be saved between invocations of the subroutine. This precaution has the disadvantage of permanently allocating space for these variables; if there is not enough data space, this could be a problem. The best solution is to 'SAVE' only those variables whose values are relied on to be the same.

If the program to be converted relies on values being initialized to zero, it is best to use a DATA statement to explicitly initialize them.

Programs that use assigned GOTO statements to enter or exit a subroutine should be changed by replacing the GOTO statements with alternate returns from the subroutine.

If the program opens files using a DCB on the stack, it is best to explicitly set the first sixteen words of the DCB to zero before opening the file.

If there are subroutine calls within the program that do not specify all the subroutine parameters, the omitted variables are not initialized, so they should be explicitly set to zero.

Pascal Conversion

If the program to be converted relies on values being saved between invocations of procedures, those variables should be made global. If the program also relies on the variables being initialized to zero, they should be made global and explicitly initialized to zero.

No More Data Space

A CDS program may have up to 31 pages available for storage of data if EMA is not used. Data space is a limited resource for CDS programs. When a CDS program runs out of data space, either LINK reports that there is no more memory in the data segment, or the system reports a run-time error due to a stack overflow for the program.

Make sure that the stack overflow is not caused by a programming error, such as a runaway recursive routine, and check that the data partition is 32 pages long, including the data base page. If the stack overflow is caused by a programming error, correct the error and run the program again. If the stack overflow is caused by a data partition of less than 32 pages, increase the data space. To increase the data space, use the CI DT command documented in the *RTE-A User's Manual*, part number 92077-90002, or the LINK "SZ" or "HE" command documented in the *RTE-A LINK User's Manual*, part number 92077-90035.

If the memory or stack overflow is not eliminated, there are four strategies to eliminate it:

1. Move as much data as possible into EMA.
2. Move as much data as possible from static storage to the stack.
3. Convert non-CDS code (which resides in the data partition) to CDS code (which does not).
4. Rewrite the program so that it requires less data.

The first strategy is usually very effective in programs which have large arrays not already in EMA. Placing arrays in EMA removes them from the logical address space and frees it up for static variables, stack space, and so on. The disadvantage of this strategy is that there is some loss of execution speed. The advantages are that it is effective and easy. In Pascal, using EMA requires the Heap 2 compiler option; refer to the *Pascal Reference Manual*, part number 92833-90005, for details. In FORTRAN, using EMA requires the EMA compiler option; refer to the *FORTRAN 77 Reference Manual*, part number 92836-90001, for details.

The second strategy can be effective on FORTRAN programs which were converted with the SAVE compiler directive. The technique of the second strategy is to remove some of the SAVE directives so that fewer variables are stored in the static space. The SAVE directives that are deleted must be carefully chosen, because SAVEd variables retain their value between subroutine calls, but local stack variables do not.

The third strategy is very effective if the source code is available to be converted to CDS. Refer to the Converting Programs to CDS section of this chapter.

11

Programmatic Spooling

This chapter applies only to operating systems with the VC+ Enhancement Package. Spooling using the CI SP command is described in the *RTE-A Print and Spooling Reference Manual*, part number 92077-90248.

A series of EXEC schedule calls are supported to allow a program to control or communicate with the spool system. These calls schedule the SMP program and pass a command parameter, a spool LU, and optionally, extra data to qualify a command. This chapter describes each call in detail.

Spooling of an LU is flagged in the system by linking a 'spool node' to the DVT connected to the LU. If the LU is not assigned a DVT, the DVT for LU 0 is used. Thus, all unassigned LUs are available as spool LUs. The 'spool node' contains information to show the node to which the LU belongs. If several LUs point to the same DVT (as is the case for unassigned LUs), the system can properly direct the current caller's request.

The 'spool node' also contains a reference to the User ID entry for the user who set up the node. Thus only programs in the requesting session will 'see' the 'spool node'. When a session is moved to the background as a result of an EX command to CI with active programs in the session, the 'spool nodes' move to the background session. When the session program count goes to zero, any remaining 'spool nodes' are removed and the controlling program (usually SPGET) is notified. The net effect is that an SP,OF is done for each LU with a 'spool node' in the terminating session.

Spool System EXEC Calls

The following explains the supported EXEC schedule calls to control or communicate with the spool system. The meanings of the return parameters are explained in the section following this.

The Start Spooling on Logical Unit and Output File to Logical Unit calls described on the following pages allow you to supply a program name (in words 32 through 34 of CONBUF) to be scheduled when spooling completes. This specified program is scheduled after the spool system purges the spool file (if it is a default file) but before the spool system purges the spool file's entry from its control file. Since non-default spool files are not automatically output, the SP,OF option or its programmatic equivalent, would schedule the program.

SPGET (running in the system session) RP's the program using FmpRpProgram. This means that the program must already exist in the system session, must exist with the given name on a FMGR cartridge, or must exist with the .RUN type extension on the /PROGRAMS directory.

Before scheduling the program, SPGET attaches to the CRT (ATCRT) of the session that set up the spool. This LU is then available to the program for messages and error reporting on LU 1. Because the program is RP'd using FmpRpProgram with no options, it will have the same

five-character name as the program and its ID segment will be released when it terminates. The program is scheduled with queue and without wait and the 41-word spoolinfo control file record is passed to the program as a runstring. The format of the spoolinfo record is given at the end of this section.

The value 0 may not be passed in an LU parameter to refer to LU 0. The value −32768 may be passed instead to refer to LU 0.

Start or Redirect Spooling on Logical Unit

To begin spooling on a logical unit or redirect spooling to another LU, use the following calls:

```
CALL EXEC(23,ismp,icommand,islu,opts,lud,0,conbuf,35)
CALL RMPAR(status)
```

where:

icommand equals 0.

ismp is a 3-word integer array containing the name 'SMP'.

islu is the spool logical unit, that is, the LU the program will address.

opts is a single-word bit mask. The bits are numbered from 0 to 15 (right to left) and are defined as follows (the options to the SP ON and LI commands that correspond to these functions are given in parenthesis):

bit	meaning when set
0	no carriage control (NC)
1	no form feed (NF)
2	keep control headers (KC)
3	purge file when done (PU)
4	print banner page (BP)
5	suspend before and after printing (SS)
6-14	reserved; set to zero
15	do not chain LU redirections (DC)

lud non-zero specifies logical unit redirection to this LU (word 0 of *conbuf* must also equal zero).

conbuf is a 35-word integer array that contains spool information as follows:

Words 0-31 = the file name descriptor blank filled. If Word 0 = 0, a default spool file is used.

Words 32-34 = the program name to schedule when spooling completes. If Word 32 = 0, no program name is specified.

status is a 5-word integer array. Word 0 contains the spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

You may wish to ensure that there is no I/O activity involving the LU to be spooled before issuing these calls.

After performing these calls, use an EXEC 14 call to retrieve the name of the spool file assigned to you. This name is needed to perform some of the other programmatic spooling calls. The effect of these calls is the same as the CI command SP,ON. See the *RTE-A Print and Spooling Reference Manual* for a description of the SP command.

Stop Spooling on Logical Unit

To stop spooling to an LU and begin output, use the following calls:

```
CALL EXEC(23,ismp,icommand,ilu[,isess])
CALL RMPAR(status)
```

where:

icommand equals 4.

ismp is a 3-word integer array containing the name 'SMP'.

islu is the spool logical unit.

isess is an integer containing the session number, or zero for your session.

status is a 5-word integer array. Word 0 contains the spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

The effect of these calls is the same as the CI command SP,OF (see the *RTE-A Print and Spooling Reference Manual*).

Output File to Logical Unit

To queue a file for output to an LU, use the following calls:

```
CALL EXEC(23,ismp,icommand,islu,opts,0,0,conbuf,37)
CALL RMPAR(status)
```

where:

icommand equals 11.

ismp is a 3-word integer array containing the name 'SMP'.

islu is the spool logical unit.

opts is a single-word bit mask. The bits are numbered from 0 to 15 (right to left) and are defined as follows (the options to the SP ON and LI commands that correspond to these functions are given in parenthesis):

bit	meaning when set
0	no carriage control (NC)
1	no form feed (NF)
2	keep control headers (KC)
3	purge file when done (PU)
4	print banner page (BP)
5	suspend before and after printing (SS)
6-14	reserved; set to zero
15	do not chain LU redirections (DC)

conbuf is a 37-word integer array that contains spool information. The contents of *conbuf* are as follows:

Words 0-31 = the file name descriptor blank filled.

Words 32-34 = the program name to schedule when output completes. If Word 32 = 0, no program name is specified.

Words 35-36 = the number of lines in the file.

status is a 5-word integer array. Word 0 contains spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

The effect of these calls is the same as the CI command SP,LI (see the *RTE-A Print and Spooling Reference Manual*).

Initialize the Spool System

To initialize the spool system, use the following calls:

```
CALL EXEC(23,ismp,icommand)
CALL RMPAR(status)
```

where:

icommand equals 12.

ismp is a 3-word integer array containing the name 'SMP'.

status is a 5-word integer array. Word 0 contains the spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

The effect of these calls is the same as the CI command SP,IN (see the *RTE-A Print and Spooling Reference Manual*).

Terminate the Spool System

To terminate the spool system, use the following calls:

```
CALL EXEC(23,ismp,icommand[,iok])
CALL RMPAR(status)
```

where:

icommand equals 13.

ismp is a 3-word integer array containing the name 'SMP'.

iok is an integer not equal to 0 if the spool system should be shut down even if spool files are active.

status is a 5-word integer array. Word 0 contains the spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

The effect of these calls is the same as the CI command SP,QU (see the *RTE-A Print and Spooling Reference Manual*). The calling program must be scheduled by a superuser.

Purge a Spool File

To stop a spool file's activity and purge it if it is a default spool file, or if the PU option was specified when the spool file was created, use the following calls:

```
CALL EXEC(23,ismp,icommand,islu,0,0,0,conbuf,32)
CALL RMPAR(status)
```

where:

icommand equals 14.

ismp is a 3-word integer array containing the name 'SMP'.

islu is the spool logical unit.

conbuf is a 32-word integer array containing spool information. The content of *conbuf* is as follows:

Words 0-31 = the file name descriptor blank filled.

status is a 5-word integer array. Word 0 contains the spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

The effect of these calls is the same as the CI command SP,PU (see the *RTE-A Print and Spooling Reference Manual).*

Restart a Spool File

To stop a spool file's activity and restart it from the start of the file, use the following calls:

```
CALL EXEC(23,ismp,icommand,islu,r,0,0,conbuf,32)
CALL RMPAR(status)
```

where:

icommand equals 15.

ismp is a 3-word integer array containing the name 'SMP'.

islu is a spool logical unit.

r is a positive/negative records to restart (relative to current file position) or 0 to restart at the start of file.

conbuf is a 32-word integer array containing spool information as follows:

Words 0-31 = the file name descriptor blank filled.

status is a 5-word integer array. Word 0 contains the spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

These calls provide the same capability as the SP,RE command (see the *RTE-A Print and Spooling Reference Manual).*

Retrieve Spool File Status

To acquire status of a spool file, use the following calls:

```
CALL EXEC(23,ismp,icommand,islu,0,0,0,conbuf,32)
CALL RMPAR(status)
```

where:

icommand equals 16.

ismp is a 3-word array containing the name 'SMP'.

islu is a spool logical unit.

conbuf is a 32-word integer array containing spool information. The contents of *conbuf* are as follows:

Words 0-31 = the file name descriptor blank filled.

status is a 5-word integer array where:

Word 0 = the spool error code (0 is normal) (See Returned Parameters section in this chapter.)

Word 1 = bit 15 = 1 if file is OUTSPOOLxx; otherwise 0.
bit 14 = not applicable.
bit 13 = 1 if number of pending lines is greater than 65535.
bits 12-8 = status, where:
0 = outputting
1 = queued for outputting
2 = active
3 = to be purged
4 = to be restarted
5 = shutdown spool system
6 = waiting on downed device
bits 7-0 = spooled logical unit.

Word 2 = bits 15-8 = link (if queued).
bits 7-0 = Session ID entry number of originating session.

Word 3 = the number of lines left in the file to output.

Word 4 = the file status, in the same format as Word 40 of the SPOOLINFO.SPL record format defined at the end of this chapter.

The effect of these calls is the same as performing the CI command SP,ST for a single file; this capability is not available in the interactive mode. See the *RTE-A Print and Spooling Reference Manual* for a description of the CI SP command.

Retrieve Line Length of all Files

To retrieve the number of lines pending on a particular LU, use the following calls:

```
CALL EXEC(23,ismp,icommand,islu)
CALL RMPAR(status)
```

where:

icommand equals 17.

ismp is a 3-word integer array containing the name 'SMP'.

islu is the spool logical unit.

status is a 5-word integer array that contains the following:

Word 0	=	the spool error code (0 is normal) (See Returned Parameters section in this chapter.)
Words 1-2	=	the number of lines left to output queued on the spool logical unit.

This capability is not available at the interactive level.

Start/Stop Error Logging

To initiate error logging to a file or terminate error logging, use the following calls:

```
CALL EXEC(23,ismp,icommand,state,0,0,0,conbuf,32)
CALL RMPAR(status)
```

where:

icommand equals 19.

ismp is a 3-word integer array containing the name 'SMP'.

state equals 0 to end error logging or 1 to begin error logging.

conbuf is a 32-word integer array that contains spool information. The contents of *conbuf* are as follows:

Words 0-31	=	the file descriptor blank filled to receive error logging. This name must be supplied even if *state* = 0.

status is a 5-word integer array. Word 0 contains the spool error code. See Returned Parameters section in this chapter for a listing of possible error codes.

The effect of this call is the same as the CI command SP,LO (see the *RTE-A Print and Spooling Reference Manual*). The calling program must have originated from a superuser account.

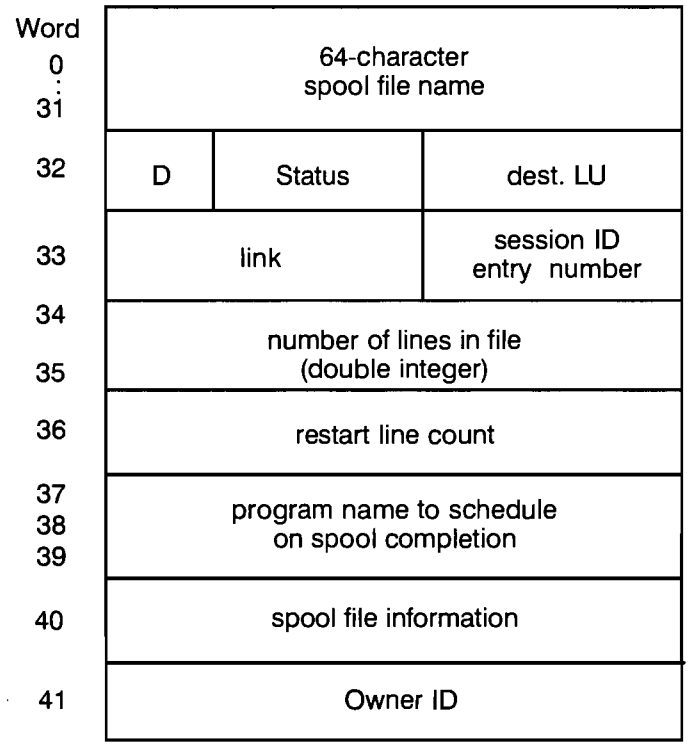
Returned Parameters

SMP returns a status from each EXEC schedule call to indicate success or failure. This status is accessible by a RMPAR call immediately following the EXEC schedule call. The error code may be found in the first parameter and has the following meaning:

parameter =	0	No error. Logical unit/spool operation was acceptable.
	1	Invalid parameter of the ON command.
	2	Spool system not initialized.
	3	SPGET is not dormant! Use OF,SPGET.
	4	Internal spool file could not be RP'd!
	5	A type 6 file could not be opened.
	6	Too many spool files allocated already.
	7	Can't open/purge spool file. No action taken!
	8	No such spool logical unit found.
	9	Filename/logical unit already active.
	10	Invalid parameter of the PU command.
	11	Invalid parameter of the OF command.
	12	Invalid parameter of the RE command.
	13	No such file found.
	14	Invalid parameter of the LI command.
	15	No SAM available for spool node.
	16	LU redirection already active on this LU.
	17	Illegal logical unit, which includes LU 1, your session terminal LU, and any LUs for device types 30b through 37b (disks and HP-IB bus controller).
	18	No such spool command.
	19	Spool system already initialized.
	20	No class/rn number available.
	21	SPGET must be sized larger!!
	22	Parameter exceeds 64 characters.
	23	Invalid parameter of the LO command.
	24	Error logging already active.
	25	Error logging is not active.
	26	You must be a superuser to perform this command.
	27	Cannot shut down spool until all spool activity stops; use "QU OK" to override.
	28	Cannot create SPOOL directory.
	29	Reserved
	30	Reserved
	31	Spool module not generated into operating system.
	32	Spooling on LU locked to a program.
	33	An operation is already pending on this file.
	34	LU to be set up is locked to a program in the caller's session.
	35	Unknown error from the system spool set up call.
	36	Session not found.
	37	Attach (and the command) failed because SP exists in the target session.
	100	Unknown/illegal command parameter.
	101	Illegal CONBUF length.
	102	Unable to schedule SP program.
	103	Spool system down.
	104	Unknown file name or logical unit.

SPOOLINFO.SPL Record Format

The SPOOLINFO.SPL record is passed to the program that is scheduled when spooling completes. The program can be specified by Start Spooling on Logical Unit or Output File to Logical Unit calls.



Words 0-31 = 64-character spool filename. This name is created in one of two ways:

- The user specifies it in the SP,ON command.
- The default file name OUTSPOOLxx is used where xx is the entry number (record number) in the SPOOLINFO.SPL file. Note that record 1 in SPOOLINFO.SPL is used for global spooling information, therefore, the records for the spool files start at record 2 and go up to record $n+1$, where n is the maximum number of spool files.

Word 32 = bit 15 (D-bit) = 1 if the file name is a default name (OUTSPOOLxx.SPL)

bits 14-8 = the status of the spool file where:

0 = outputting
1 = queued for outputting
2 = active
3 = to be purged
4 = to be restarted
5 = reserved
6 = waiting on down device

bits 7-0 = the spooled LU, that is, the LU to which the spool file will be output.

Word 33 = bits 15-8 = the output queue information where:

0 = end of queue
nonzero = next SPOOLINFO record in queue (only valid if status = 0 or 1)

bits 7-0 = the session ID entry number of the session (system session ID is 0).

Words 34-35 = the number of lines (records) in the file (this number refers to the number of I/O requests made to the file).

Word 36 = the number of lines to restart the spool file if status = 4 (to be restarted).

Words 37-39 = program name to schedule upon spool output completion (used by programmatic interface).

Word 40 = file type information:

bit 15 = 1 if the file was listed (LI,FILENAME).

bit 14 = 1 if the NC option was specified.

bit 13 = 1 if an operation is already pending on a spool file, for example, someone did a SP,PU,file. This allows the spool system to process one command without someone entering another.

bit 12 = 1 if the NF option was specified.

bit 11 = 1 if the KC option was specified or this is a default spool file.

bit 10 = 1 if the PU option was specified or this is a default spool file.

bit 9 = 1 if the BP option was specified.

bit 8 = 1 if the SS option was specified.

Word 41 = Owner ID

12

Privileged Operation

In normal operation, RTE-A protects all user programs from one another and protects the operating system from user programs. User programs do not interfere with each other; that is, your program cannot inhibit another user program from executing nor can it access memory occupied by any other user program. User programs do not have the capability to interfere with the operating system's execution and cannot access and modify the system tables. If any user program attempts to violate this protection, the operating system aborts the program and reports a Memory Protect (MP) violation.

In certain unusual circumstances a very powerful user program may find it necessary to disable RTE-A's normal protection mechanisms in order to accomplish a particular task. Executing with protection disabled is called privileged operation.

An example of a program that operates in privileged mode is WH, the system status utility. In order for WH to present a consistent view of system status, it momentarily disables RTE and all other user programs from executing and takes a snapshot of system tables. If WH executed in normal operation mode, the system tables could change while being read by WH, which would make the status information unreliable.

You must use the privileged operation mode with care because disabling the protection mechanism can create problems in your system. Privileged programs that are coded improperly can easily corrupt RTE-A and other user programs, and crash the system. Even properly coded programs can degrade system performance seriously if RTE-A is disabled for long periods of time.

GOPRV and UNPRV

GOPRV disables the normal Memory Protect mechanism and allows your program to write into protected memory areas or execute privileged instructions. Any program that modifies system tables or alters Dynamic Mapping System registers can use this call. (DMS registers and privileged instructions are explained in the hardware reference manuals.) Normal operation is resumed by calling UNPRV.

When using GOPRV, you must exercise caution if your program executes I/O instructions because your program can still be interrupted. An I/O interrupt causes the Global Register to be enabled with the select code of the I/O card that just interrupted. Therefore, after the interrupt is serviced, it is likely that the Global Register will be set to a different select code than the one that the user wants. It is recommended that you use $LIBR for privileged code that executes I/O instructions.

Calling GOPRV does not stop RTE-A or other programs from executing. Normal system operation is affected only if your program deliberately uses its privilege to do so. For example, normal system operation is affected if your program alters the list of scheduled programs or executes a halt instruction.

The privileged program is still under the control of RTE-A and, therefore, is subject to time-slicing, swapping, and other operating system controlled operations. EXEC calls and other system calls can be made while GOPRV privileged operation is in effect.

A CDS program should not execute any subroutine calls while GOPRV is in effect because GOPRV disables the stack overflow detection feature for CDS programs. If your CDS program calls GOPRV and then causes a stack overflow, the error cannot be detected.

GOPRV and UNPRV are callable from MACRO as well as high-level languages:

from MACRO:

```
(non-CDS)

JSB GOPRV
DEF *+1
 .
 .
 .
JSB UNPRV
DEF *+1
```

```
(CDS)

PCAL GOPRV,0,0,0
 .
 .
 .
 .
PCAL UNPRV,0,0,0
```

from FORTRAN:

```
CALL GOPRV()
 .
 .
 .
 .
CALL UNPRV()
```

DispatchLock/DispatchUnlock

DispatchLock allows your program to prevent all other user programs from executing. The most common reason for this is allocating/using an unprotected resource for which many programs may be competing simultaneously. Such a section of code is called a critical section, which protects a critical resource. RTE-A also provides other mechanisms, such as resource numbers, for protecting critical resources.

A program is dispatch locked when it prevents the execution of all other programs. In RTE-A, a program can use either .ZPRV or DispatchLock to become dispatch locked. .ZPRV is valid only in subroutines located in system common. Functionally, .ZPRV and DispatchLock are identical. Chapter 8 of the *RTE-A System Design Manual*, part Number 92077-90013, contains a description of .ZPRV.

Only one program at a time can be dispatch locked because after one program becomes dispatch locked, all other programs are not eligible for execution and, therefore, cannot make DispatchLock requests.

A dispatch locked program cannot access protected memory or execute privileged instructions. RTE-A is not disabled and continues to control the dispatch locked program.

Because a critical section should not be interrupted, off (OF) and suspend (SS) commands do not immediately affect dispatch locked programs — the program is not aborted or suspended until it becomes dispatch unlocked. Therefore, the only way to stop a dispatch locked program that is looping infinitely is to reboot the system.

A dispatch locked program can make EXEC calls and other system calls. However, because it is the only program RTE-A will consider for execution, a dispatch locked program can never become suspended. If it did, there would be no program eligible for execution.

RTE-A aborts a dispatch locked program and issues an SR error if the program makes a request that would normally cause it to become suspended. There are many causes for suspension; for example, the program issues an EXEC 7 call, a device is down or locked, or the program requests to wait for I/O to complete. A non-CDS dispatch locked program should not make an EXEC 8 call and a multi-segment dispatch locked program CDS program should not execute PCALLs because the program would be suspended waiting for an overlay or CDS segment, respectively, to be loaded from disc.

DispatchLock and DispatchUnlock are callable from MACRO as well as high-level languages:

from Macro: **from FORTRAN:**

```
(non-CDS)                (CDS)

JSB  Dispatchlock     PCAL  DispatchLock,0,0,0     Call  DispatchLock()
def  *+1               .                            .
 .                     .                            .
 .                     .                            .
 .                     .                            .
JSB  DispatchUnlock   PCAL  DispatchUnLock,0,0,0   Call  DispatchUnlock()
def  *+1
```

$LIBR/$LIBX

$LIBR provides the ultimate level of privilege. A program that calls $LIBR controls the system; RTE-A, other user programs, normal I/O interrupts, and Memory Protect are all disabled. Privileged I/O interrupts are serviced. Any TBG ticks occurring after a program calls $LIBR are not serviced until the program calls $LIBX to terminate its privilege.

EXEC calls and other system calls cannot be made while a program controls the system because $LIBR disables RTE-A.

When a shared subroutine in system common calls $LIBR, the subroutine is called a level 1 subroutine. Shared subroutines are discussed in chapter 8 of the *RTE-A System Design Manual*, part number 92077-90013.

The calling sequence for $LIBR and $LIBX is as follows:

From Macro (CDS or non-CDS):

```
        jsb $LIBR
        dec 0
          .
          .
          .
        jsb $LIBX
        def retadr
retadr  def return
return  . . .
```

Guidelines for Privileged Operation

The following guidelines should be observed when privileged routines are used:

All privileged programs should be coded with care because an improperly coded program can easily crash the system.

Because of the impact on system performance, I/O interrupts, and other programs, privileged operation should be limited to 1 millisecond.

If you use Symbolic Debug/1000 to debug programs that contain privileged sections, setting breakpoints or attempting to monitor program execution in the privileged section will cause the system to crash.

If the working map is changed while in a privileged routine, it must be restored to its previous value before entering the operating system or using FORTRAN I/O; otherwise, the system can crash.

When using GOPRV, you must exercise caution if your program executes I/O instructions because your program can still be interrupted. It is recommended that you use $LIBR for privileged code that executes I/O instructions.

The system calls to invoke the various privilege levels can be nested if you observe several restrictions. If any of the following restrictions are violated the program is aborted with an SR (subroutine) error:

- Any privileged call can be nested with calls of the same type.
- A dispatch locked program can make EXEC, CLRQ, and LURQ calls, and call GOPRV and $LIBR.
- After calling GOPRV, a program can make EXEC, CLRQ, and LURQ calls, and call DispatchLock and $LIBR.
- After calling $LIBR, a program cannot call EXEC, CLRQ, LURQ, DispatchLock, or GOPRV.

13

RTE-A Signals

Introduction to Signals

Signals are the software parallel to hardware interrupts. They provide a mechanism for the operating system or other programs to communicate to a program. This is accomplished by separating the program that is to receive signals into two parts: the main program and a signal handler. Typically, the main program executes until a signal is delivered. At that time, the operating system, or another program, calls the signal handler by using signal routines, passing it the signal number and a block of data that is signal dependent. The handler uses this information to determine what action to take and it informs the operating system when it completes the action. When the signal handler was invoked, the operating system supplied it with information about the interrupted environment of the main program. This information is used to restore the environment of the main program and resume the main program's execution.

If it is not possible to service a signal because the main program is in a critical region, the signal handler can set a flag that the main program can check when it is beyond the critical region.

Available Signals

There are five signals supported in RTE-A. Three of the signals correspond to program violations, I/O completions, and timer completions. The other two supported signals are user definable.

Table 13-1 summarizes the main characteristics of each signal. Note that for each signal there is a number, a mnemonic, and a default action. The signal number is the value passed to the signal subroutines to represent a particular signal, while the mnemonic represents the signal within documentation. A program may define constants with the mnemonic names whose values are the signal numbers. The default action is the action taken when a program without a signal handler receives a signal. With the exception of SglAlrm, data is sent along with each signal indicating additional information about the signal.

Note that higher priority is given to signals with lower signal numbers. Thus, if signals are blocked and a SglAlrm signal and a SglIO signal have been queued, the signal handler will receive the SglAlrm signal first when the signals are released.

Table 13-1. Signal Types

Type	Num	Mnemonic	Default Action	Data Sent by the Operating System	Priority
Program Violation	4	SglVio	Abort	Violation Type	High
Timer Completion	14	SglAlrm	Abort	None	.
User Definable	16	SglUsrl	Abort	User Definable	.
User Definable	17	SglUsr2	Abort	User Definable	.
Class I/O Completion	22	SglIO	Ignore	I/O Request Data.	Low

Program Violation – SglVio

When a program has generated a violation such as a memory protect, the operating system sends the SglVio signal to that program. The default action for this signal is to abort the program. This means that if a SglVio is sent by the operating system to a program without a handler, the program aborts with a message such as:

```
KRSTN  aborted at address   3203 Reason is  MP   Current segment=    0
A= 40556  B= 62145  X= 71163  Y= 67556  E= 0  0= 0  WMAP= 106002
Instruction=102077  Z=     0  Q=     0  CS mode=OF
MP = Memory protect (I/O instruction or store/jump to protected memory)
```

If this signal is sent by the operating system to a program with a signal handler, the handler receives a signal number of four (SglVio) and the signal dependent data is two words containing four ASCII characters. These characters are used to identify the type of error that occurred. Some possible values are: MP followed by two spaces, SC04, or LU00. Note that these are the same abbreviations that RTE-A reports when it aborts a program that does not have a signal handler. See the *RTE-A Quick Reference Guide*, part number 92077-90020, or Appendix A of this manual for a list of possible errors. A program violation signal is delivered whenever RTE-A attempts to abort a program that has a signal handler, except when the program is aborting because of a parity error (PE), load error (LD), or a swap error (SW). In these cases, the program is aborted in the same manner as if it did not have a handler.

If this signal is sent by a user program to a program with a signal handler, the handler will receive a signal number of four and the signal dependent data will be determined by the sending program.

If a SglVio is sent by a user program to a program without a handler, the program to which the signal was sent aborts with an SG07 violation:

```
DL     aborted at address  52516 Reason is  SG07 Current segment=    0
A= 52506  B=   123  X=     0  Y=   357  E= 0  0= 0  WMAP= 106002
Instruction=177777  Z=     0  Q=     0  CS mode=OF
SG07 = Unexpected signal received.
```

Timer Completed – SglAlrm

The operating system sends a SglAlrm signal when an interval timer expires. The timer intervals are set by an application using the timer services documented later in this chapter.

The default action for this signal is to abort the program. If either the operating system or a user program sends SglAlrm to a program that does not have a signal handler, the receiving program is aborted with an SG07 error, in the manner described in the discussion of SglVio. If SglAlrm is sent to a handler, signal 14 is indicated, there is no signal dependent data sent by the operating system.

User Definable – SglUsr1 and SglUsr2

These signals are usually used when one program wants to send a signal to another program. The operating system does not send either SglUsr1 or SglUsr2. A typical application for SglUsr1 and SglUsr2 is program to program communication.

For example, assume the program CNSMR wants the program SERVR to retrieve information from a database and return it to CNSMR. CNSMR and SERVR would be written with signal handlers. When CNSMR wanted information, it would send SglUsr1 (signal number 16) to SERVR. The signal dependent data contains a request for the information, the name of the requesting program (CNSMR), and the requesting program's session number. CNSMR sends its name and session number so that SERVR knows where to send the requested data. Upon receipt of the request, SERVR acquires the information from the database and then sends a SglUsr1 (signal number 16) to CNSMR with the requested database information in the signal dependent data.

The default action for this signal is to abort the program. If a user program sends SglUsr1 or SglUsr2 to a program that does not have a handler, the receiving program is aborted with a SG07 error. The discussion of SglVio contains details of default actions.

Class I/O Completion – SglIO

SglIO is sent by the operating system to a program when a class I/O operation initiated by a program is complete.

The default action for this signal is to ignore it. This means that if SglIO (signal number 22) is sent by either the operating system or a user program to a program that does not have a handler, no action is taken by the operating system, that is, the program is not aborted. If this signal is sent by the operating system to a program with a signal handler, the signal dependent data consists of six words containing information about the original class request.

The formats of the original class requests are:

For EXEC 17,18 and 20:

EXEC (*ecode* , *cntwd* , *bufr* , *bufln* , *pram3* , *pram4* , *class* [, *uv* [, *keynum*]])

and for EXEC 19:

EXEC (19 , *cntwd* , *pram1* , *class* [, *pram2* [, *pram3* [, *pram4* [, *uv* [, *keynum*]]]]])

The following describes the six words of signal dependent data:

Word	Contents	Corresponds to
1	class number	*class* parameter, the top three bits are undefined
2	transmission log	length of completed request, corresponds to the value returned in the B-Register for EXEC 21 (Class get)
3	*pram3*	*pram3* parameter in EXEC call
4	*pram4*	*pram4* parameter in EXEC call
5	request type	dependent upon ecode in EXEC call: 1 if *ecode* was 17 (class read) 2 if *ecode* was 18 (class write) 3 if *ecode* was 19 (class control) 1 if *ecode* was 20 (class write/read) In addition, the sign bit will be set if the request is being flushed.
6	user defined value	*uv* parameter in EXEC call

For more detail on class requests, refer to the chapter on Class I/O.

Signal Service Subroutines

This section describes the subroutines provided for interfacing to the signal services. The use of the signal service subroutines are demonstrated in subsequent sections that describe the signal handler and discuss the sending and blocking of signals.

The calls are described in the format of normal subroutines, which have the advantage of being readable. However, EXEC 37 calls may also be used. EXEC 37 calls have the advantages of speed, reentrancy, and the no-abort and no-suspend options. Values for all of the parameters must be present in the call. Integers are 16 bits and double integers are 32 bits.

Note that the signal service subroutines that perform equivalent EXEC 37 calls do not set the no-abort or the no-suspend bits. The EXEC call format must be used if either of these functions are desired.

If a signal service subroutine call or an EXEC 37 call without the no-abort bit set encounters an error, the program receives a violation. If the no-abort bit is set and an error occurs, the 4-character error code is returned in the A- and B-Registers. If no error occurs, zero is returned in the A-Register.

Table 13-2 lists the signal subroutines and the equivalent EXEC 37 calls.

A description of the signal error codes is located in Appendix A, Group III errors.

Table 13-2. Signal Subroutines

Subroutine	Equivalent EXEC 37 Call
SglHandler(handler)	(37,1,handler)
SglLimit(lo,hi,rtlo,rthi)	(37,2,lo,hi,rtlo,rthi)
SglKill(name,ses,sig,buf,len)	(37,3,name,ses,sig,buf,len)
SglSetMask(mask)	(37,4,1,mask)
SglBlock(mask)	(37,4,2,mask)
SglPause(mask)	(37,4,3,mask)
SglLongJmp(env)	(37,5,env)
SglSetJmp(env)	(37,6,env)
SglAction(sig)	none

SglAction

SglAction, when it is passed a signal number in *sig*, returns an integer value signifying the action to be taken.

```
action = SglAction(sig)
```

where:

action is an integer specifying the action to take.

sig is an integer signifying the signal number.

Usually, a signal handler cannot handle all the possible signal types it might receive. When an unexpected signal arrives, the handler needs to know the action to take. SglAction may be called to determine the action. If the bottom bit of the returned value is set, the meaning is that the RTE-A operating system is capable of delivering this signal. If the top bit is set, the meaning is that the default action is to ignore the signal, that is, simply to return to the interrupted environment instead of aborting the program.

For example, SglAction will return with the top bit set (that is, ignore) for a class completion signal but it would be clear (that is, abort) for a program violation signal.

There is no equivalent EXEC call for SglAction.

SglBlock

SglBlock is a double integer function that returns the previous set of masked signals and blocks signals in addition to the current signals.

```
oldmask = SglBlock(mask)
```

where:

oldmask is a double integer that returns the previous set of masked signals.

mask is a double integer that contains a mask of signals that are to be blocked from delivery.

This call is equivalent to making the set of currently blocked signals equal to the existing set logically OR'ed with Mask. See SglSetMask to set the blocked mask to Mask without OR'ing mask with the current mask.

The equivalent EXEC call sequence is (37,4,2,*mask*).

Note SglVio signals should not be blocked under normal circumstances. Doing so will cause an excessive amount of XSAM to be used for queuing SglVio signals if RTE-A tries to abort the program.

SglHandler

SglHandler sets the signal handler address.

```
CALL SglHandler(handler)
```

where:

handler is a pointer to the handler routine.

If an error occurs, none of the signal environment is initialized.

The equivalent EXEC call sequence is (37,1,*handler*).

SglKill

SglKill sends a signal to a program.

```
CALL SglKill(name,ses,sig,buf,len)
```

where:

name is a three-word integer buffer that specifies the name of the program that is to receive the signal.

ses is an integer identifying the session the program is in.

sig is an integer specifying the signal that is to be sent. If *sig* is zero, error checking is performed on the other parameters. The following are valid signal numbers.

Signal Number	Signal Type
4	Program Violation
14	Timer Completed
16	User Definable
17	User Definable
22	Class I/O Completion

buf is an integer buffer containing the signal dependent buffer that should be passed to the program.

len is an integer identifying the length of the signal dependent buffer in bytes.

If *sig* is invalid (less than one or greater than 32), an SG02 error results.

If *name* and *ses* define a program that does not exist, an SG08 error occurs. If they point to a program that is not part of your session and your session does not have the capability to send signals across session boundaries, then an SG05 error occurs.

If there is insufficient XSAM to satisfy the request, the sending program is suspended unless the nosuspend bit was set. If this is the case, an SG04 error is returned.

If the signal buffer limit has been reached for the receiving program, the sending program will be signal buffer limit suspended unless the nosuspend bit was set, if the nosuspend bit was set, an SG06 error occurs.

The equivalent EXEC call sequence is (37,3,*name*,*ses*,*sig*,*buf*,*len*).

SglLimit

SglLimit sets the signal buffer limits.

```
CALL SglLimit(lo,hi,rtlo,rthi)
```

where:

lo	is an integer specifying the lower buffer limit for the program receiving signals.
hi	is an integer specifying the upper buffer limit for the program receiving signals.
rtlo	is the same as *lo* but applies to real-time programs (priority ≤ 40).
rthi	is the same as *hi* but applies to real-time programs (priority ≤ 40).

If a program calls SglKill to deliver a signal and if queuing that signal causes the total amount of XSAM used for queuing signals for the receiving program to be greater than *hi*, then all future SglKill calls will be blocked by the signal buffer limit, suspending the sending program or returning SG06 errors if the nosuspend bit was set. This state remains until the total amount of XSAM used is less than *lo*, then all suspended programs are resumed.

Each signal uses 2 words of XSAM for a header plus the amount of XSAM required to store the signal dependent data. If *hi* and *lo* are zero, buffer limits are removed and all of XSAM may be used for signals, this should not normally be done because all of XSAM could conceivably be used up.

If there is an error in one of the limits, such as $hi < lo$, for example, an SG02 error code is returned. Note that the operating system stores the values in a compressed format. Because of this compression, the values are rounded, the lower limit becomes *lo* div 16 * 16 and the upper limit becomes (*hi* − *lo*) div 16 * 16. This may cause nonzero buffer limits to be rounded to zero. As a result, the following conditions result in an SG02 error:

$hi > 6112$, or, $hi - lo > 2032$

The default amounts are based on the amount of XSAM in the system. These values are approximately:

lo:	1/32 of XSAM
hi:	1/64 of XSAM + *lo*
rtlo:	1/8 of XSAM
rthi:	1/16 of XSAM + *rtlo*

The equivalent EXEC call sequence is (37,2,*lo*,*hi*,*rtlo*,*rthi*).

SglLongJmp

SglLongJmp jumps to supplied environment.

```
CALL SglLongJmp(env)
```

where:

env is a three-word integer array containing information set up by a SglSetJmp call or by the operating system calling a signal handler.

SglLongJmp preserves the value of all registers except the PC-, C-, and Q-Registers.

The equivalent EXEC call sequence is (37,5,*env*).

SglPause

SglPause waits for a signal to be delivered to the program. SglPause returns when a signal is delivered and the signal handler returns to the interrupted code.

```
CALL SglPause(mask)
```

where:

mask is a double integer specifying the signals that should be blocked from delivery while the program is paused.

The *mask* parameter allows you to unblock a set of signals and then wait for at least one of those signals to arrive as an "atomic" operation. This means that no signals may be delivered in between the operations of unblocking and waiting for a signal, guaranteeing that no signals will be "missed" by the waiting operation.

When SglPause returns, the signal mask is restored to the value it had before SglPause was called.

The equivalent EXEC call sequence is (37,4,3,*mask*).

SglSetJmp

SglSetJmp sets an environment.

rtntype = `SglSetJmp(`*env*`)`

where:

rtntype is an integer that returns 0 when SglSetJmp returns after setting an environment, or returns the contents of the A-Register at the time that environment is jumped to via SglLongJmp.

env is a three-word integer array returning information about the present environment including the PC with a CDS mode indicator, the active code segment, and the current stack register (Q).

When a SglLongJmp is executed that jumps back to the environment saved by SglSetJmp, the effect is as if the SglSetJmp call returns again. The contents of the A-Register at the time the SglLongJmp is executed is returned in the *rtntype* variable. If the A-Register is non-zero at the time of the SglLongJmp call, then the code following the SglSetJmp call can distinguish which type of return is made: a return from the SglSetJmp call or from a SglLongJmp call. See the section, "Program Example Using SglSetJmp" later in this chapter for an example.

The equivalent EXEC call sequence is (37,6,*env*).

SglSetMask

SglSetMask is a double integer function that blocks signals and returns the previous set of masked signals.

oldmask = `SglSetMask(`*mask*`)`

where:

oldmask is a double integer that returns the previous set of masked signals.

mask is a double integer that contains the mask of the signals that are to be blocked.

Unlike the SglBlock call, which adds signals to be blocked to the current set of blocked signals, the SglSetMask call sets the blocked signals mask to *mask*. That is, any signals not specified in *mask* will no longer be blocked.

The equivalent EXEC call sequence is (37,4,1,*mask*).

Note SglVio signals should not be blocked under normal circumstances. Doing so causes an excessive amount of XSAM to be used for queuing SglVio signals if RTE-A tries to abort the program.

Signal Handler

A signal handler must be a non-CDS routine written in Macro. The main program informs the operating system of the address of a signal handler using the subroutine SglHandler or the equivalent EXEC (37,1,*handler*) call. The operating system then calls the handler when a signal is being sent to a program. The signal may be from the operating system itself or from another program.

The handler determines what actions are to be taken depending on the signal. The handler must also determine if the action can be taken immediately. For instance, the main program may be in a critical region and can take delivery of the signal only when it is beyond that region. This means the handler must set a flag that the main program can check once it is beyond the critical region.

Buffer Descriptors

The signal handler has two buffer descriptors preceding the entry point. The first buffer descriptor contains the description of the save area for the program's environment at the time it was interrupted. This allows for resumption of the program at the point of interruption. The second buffer descriptor describes the save area for the signal dependent data.

The following is an example of the declaration of buffer descriptors:

```
EnvBuf      equ *
PC          bss 1       ;Environment
Q           bss 1       ;
CodeSeg     bss 1       ;Save Area (43 Words)
PTE Page    bss 1       ;
DataMaps    bss 32      ;
StartPage   bss 1       ;
Status      bss 1       ;
IDTemps     bss 5       ;(to save the ID temp words)

SigBuf      equ *       ;Signal Buffer
SigNum      bss 1       ;Signal Number
Length      bss 1       ;Signal Dependent Data byte length
Data        bss 6       ;Signal Dependent Data

; The following are the buffer descriptors.
; They must immediately precede the signal handler's entry point.

            def EnvBuf  ;pointer to environment buffer
            dec 43      ;length of env buf in words
            def SigBuf  ;pointer to signal buffer
            dec 8       ;length of signal buffer in words

Handler     dst AB      ;beginning of handler...
```

Environment Buffer

The environment buffer describes the interrupted environment at the time the signal is delivered and is made up of seven parts:

1. The program counter, with the CDS indicator in the sign bit (set if CDS mode is on);
2. The Q register (the stack pointer);
3. The code segment number;
4. The physical page number of the PTE;
5. The program's map set registers for the data segment;
6. The starting physical page of the data partition;
7. The program's state.

The first three words of the environment buffer consisting of program counter, the stack pointer, and the code segment are essential to allow the main program to resume execution at the point it left off. This information is passed in an EXEC call equivalent to the SglLongJmp subroutine when the handler returns execution to the main program. If the extra five words (words 39 through 43) of the environment buffer are declared, RTE-A also restores the $TMP1 through $TMP5 words of the ID segment when the program resumes. The remaining information is not essential and is available for the signal handler to use if desired.

Signal Buffer

The signal buffer contains the signal dependent data and consists of three parts:

1. The number of the signal that is being handled.
2. The length of the signal dependent data that could be returned, in bytes.
3. The signal dependent data.

The length in bytes of the signal dependent data that could be returned is stored in the second word of the signal buffer. This length tells the handler how much of the signal dependent data buffer is valid.

The length value stored here could be greater than the length of the buffer. For example, assume the length set in the buffer descriptor is 40 bytes (20 words), and the signal wants to send 39 bytes of signal dependent data. Only the first 36 bytes of data would be in the buffer because two words (four bytes) of the buffer are used for the signal number and the signal length. Even though the length word in the signal buffer would still be set to 39, only as much data as was specified in the buffer descriptor will be transferred.

Hardware Status Saving

In addition to having the environment and signal buffer descriptors, the signal handler must save the state of the hardware registers. This ensures that when execution returns from the handler to the main program, the contents of the A-, B-, E-, O-, X-, and Y-Registers match the contents at the time of interruption. Saving of the register contents is performed at the beginning of the handler, as in the following example:

```
Handler dst ABSave    ;beginning of handler
                      ;Save A,B,E,O,X,Y
        era,als
        soc
        ina
        sta EOSave
        stx XSave
        sty YSave ...
```

Reentrant Subroutines

The signal subroutines are not reentrant. The signal handler should be written in the equivalent EXEC 37 calls instead of normal subroutines because the EXEC calls are reentrant.

Note Calling non-reentrant subroutines from a signal handler may cause loss of data or worse. Problems can occur when the main program is executing within a non-reentrant subroutine and it is signalled by its signal handler which invokes the same subroutine.

Exiting the Signal Handler

To exit the signal handler use the EXEC (37,5,*env*) call after restoring the program's hardware register status.

```
          .
          .
return    ldy YSave          ;Restore Y
          ldx XSave          ;Restore X
          ldb EOSave         ;Restore E & O
          clo
          slb,elb
          sto
          dsd ABSave         ;Restore A & B

          jsb                ;Long Jump
          def *+4            ; back to
          def =37            ; interrupted
          def =5             ; environment
          def EnvBuf
          .
          .
```

The *env* parameter should contain the first three words of the environment buffer that was saved when the signal handler was called, this enables the main program to resume at the point it was interrupted. Until the handler has been exited by using the EXEC (37,5,*env*) call, no new signals are given to the handler.

If a signal has interrupted an EXEC call, the EXEC call is restarted when the handler executes a SglLongJmp. One exception to this is when a signal has interrupted a time suspend request, the SglLongJmp subroutine causes the time suspension to be terminated. A signal is not delivered if a program is paused, operator suspended, program wait suspended, or I/O suspended until the program is scheduled.

Blocking Signals

The main program must be able to continue uninterrupted in certain critical periods of execution. To do this, the program needs the capability to block the delivery of signals.

The subroutines SglSetMask and SglBlock and their EXEC call equivalents are available to specify which signals are to be blocked from delivery. The signals to be blocked are specified in the *mask* parameter of the subroutines. Signal number *i* is blocked if bit $i-1$ is set in *mask*, where the bits are numbered from 0 to 31, bit 0 being the least significant bit. The *mask* parameter is stored in standard double integer format, with the first word in memory containing the most significant bits. See Table 13-1 for a list of signal numbers. The following FORTRAN statement sets the bit for signal *i*:

```
mask = ibset(mask,i-1)    (where mask has been declared as an integer*4)
```

SglPause can be used to unblock and wait for a signal. Previously blocked signals may be unblocked by setting *mask* to zero in a SglSetMask call. The subroutine SglBlock(*mask*) is used to add signals, by using a logical OR statement, to an existing set of blocked signals.

The operating system queues the blocked signals in XSAM until the signals are unblocked. Once unblocked, the signals are released for delivery in FIFO order according to their priority. For example, the operating system releases all signal 4's, then all signal 14's and so on.

Sending Signals

There are two ways a signal can be sent to a program. First, the operating system can send a signal to a program because an asynchronous event has occurred, and second, a program can send a signal to itself or to another program. The following sections describe these two ways of sending signals.

A Simple Use of Signals

This section lists two very short programs, one with a signal handler, and one without. The programs create an RQ violation by making an EXEC call without parameters. The comparison of the two programs demonstrate the use of signals and the handler. The first version, rq1, has no handler.

```
ftn7x,q,c
      program rq1

C     Create an RQ violation
      call exec(0)

      end
```

When the above program is executed, the following message is issued to the screen:

```
  RQ aborted at address   2003 Reason is  RQ   Current segment=       0
  A=     0  B= 55025  X=    40  Y= 55737  E= 1  O= 0  WMAP= 106002
  Instruction=100700  Z=     0  Q=     0  CS mode=OF
  RQ = Bad or too many EXEC parameter(s)
```

The following example illustrates a simple but effective use of signals with a signal handler. The signal handler code follows the discussion of the rq2 program.

```
ftn7x,q,c,
      program rq2
        implicit none
      external handler

C     Set up a signal handler
      call exec(37,1,handler)

C     Create an RQ violation
      call exec(0)

      end
```

After rq2 sets up a signal handler and makes an EXEC call without any parameters, a SglVio signal is sent from the operating system to rq2's handler. When rq2 is run, the following message is printed:

```
  Unexpected fatal signal received.
  Signal number =  4.
```

Because the handler for rq2 is written to expect a SglUsr1 signal, the SglVio signal is unexpected. The handler prints a message and reports that it received signal number four.

The following typical handler is used by the rq2 program.

```
macro,q,c,s

      nam handler
      ent handler
      ext exec, SglAction, Kcvt

; Pointers to the buffers for RTE-A.  Note that these must
; immediately precede the entry point to the handler.
      def EnvBuf                  ;Pointer to the environment buffer
      abs EnvEnd-EnvBuf           ;Length of the environment buffer
      def SigBuf                  ;Pointer to the SDD buffer
      abs SigEnd-SigBuf           ;Length of the SDD buffer
handler equ *                     ;Signal Handler entry point

      dst ABSave                  ;Save A and B

; Note that in this handler, X, Y, E and O don't really need
; to be saved, because they are not used anywhere (SglAction
; only modifies A and B).  This code is only here in case
; code is later added that does make use of these registers.
      era,als                     ;Save E & O
      soc
      ina
      sta EOSave
      stx XSave                   ;Save X
      sty YSave                   ;Save Y

      lda SigNum                  ;Get the signal number.
      cpa =d16                    ;Was it SglUsr1?
      jmp expected                ; Yes.
      jmp unexpect                ;Unexpected signal.

; Received an expected signal, take care of it.
expected nop

return ldy YSave                  ;Restore Y
       ldx XSave                  ;Restore X
       ldb EOSave                 ;Restore E & O
       clo
       slb,elb
       sto
       dld ABSave                 ;Restore A & B

       jsb Exec                   ;Long Jump
       def *+4                    ; back to
       def =d37                   ; interrupted
       def =d5                    ; environment
       def EnvBuf

; Received an unexpected signal, find out appropriate action
unexpect jsb SglAction
       def *+2
```

```
         def SigNum
         ssa                  ;If the top bit is set,
         jmp return           ; ignore the signal.

         jsb Kcvt             ;Convert signal number to ASCII.
                              ;Although Kcvt is not reentrant,
                              ; the handler is going to terminate
                              ; and execution is not going to
                              ; return to the handler.
         def *+2
         def SigNum

         sta AscNum

         jsb Exec             ;Write out error.
         def *+5
         def =d2
         def =d1
         def Msg
         def MsgLen

         lda SigNum           ;Was the signal a SglVio?
         cpa =d4
         rss
         jmp NotVio

         dld SigData          ;Yes, copy the error code
         dst Reason           ; into the message.

         jsb Exec             ;Write out why we were aborted.
         def *+5
         def =d2
         def =d1
         def Msg1
         def MsgLen

NotVio   jsb Exec             ;Quit
         def *+3
         def =d6
         def =d0

Msg      asc 17,Unexpected fatal signal received.
         oct 6412
         asc  8,Signal number =
AscNum   bss 1
         asc  1,.
         oct 6412
MsgLen   abs MsgLen-Msg
Msg1     asc 8,Type of error: "
Reason   bss 2
         asc 1,".
         oct 6412
Msg1Len  abs Msg1Len-Msg1
```

```
; Define the buffer for the environment to be placed in.
EnvBuf  equ *
PC      bss 1
Q       bss 1
CodeSeg bss 1
EnvEnd  equ *

; Define the buffer for the signal-dependent data (SDD).
SigBuf  equ *
SigNum  bss 1
SigLen  bss 1
SigData bss 2
SigEnd  equ *

ABSave  bss 2
EOSave  bss 1
XSSave  bss 1
YSave   bss 1
        end
```

Signals Sent from User Program to User Program

The two programs that follow use signals for inter-process communication. The main programs are written in FORTRAN. Each uses a signal handler written in Macro. These examples use signal subroutines to set up, block, release, send, and wait for signals. Only the important portions of the handlers are presented.

The program CNSMR reads data from the terminal and sends it to another program SERVR. SERVR processes the data and returns the result to CNSMR. The data and the result are sent from program to program within the buf parameter of either the SglKill subroutine or the equivalent EXEC call.

Both CNSMR and SERVR programs begin by blocking SglUsr1 using SglSetMask so that at the appropriate time, they can employ SglPause to receive the SglUsr1 signal when ready.

Next, each program employs the SglHandler subroutine to set up a signal handler, the handlers are written to perform the default action for all signals except for SglUsr1, the expected signal. Both handlers define the buffers for the interrupted program environments. They also define the buffers for the signal dependent data (SDD) sent as a parameter in the SglKill subroutine along with the signal number and buffer length.

CNSMR prompts the user to interactively enter values for the three integer variables: operand, data1, and data2. These values become the signal dependent data carried in the buf parameter of the SglKill subroutine along with the name of the program SERVR and its session. The program SERVR is the program to receive the signal and to process the signal dependent data. After calling SglKill, CNSMR calls SglPause to unblock the SglUsr1 signal and wait to receive the result of the processed data from SERVR.

```
ftn7x,q,s
      program cnsmr
      implicit none
      integer SglUsr1
      parameter (SglUsr1=16)
      external handler

      integer*4 mask, oldMask, SglSetMask
      integer SglPause, SglHandler, UsNum
      logical running
      integer result, MyName, Session, operand, data1, data2, buf(7)
      common /result/ result
      equivalence (buf(1),MyName), (buf(4),Session)
      equivalence (buf(5),operand), (buf(6),data1), (buf(7),data2)

C     Block signal SglUsr1
      mask = 2j ** (SglUsr1 - 1)
      oldMask = SglSetMask(mask)

C     Set up a signal handler
      if (SglHandler(handler) .ne.  0) then
          stop 'CNSMR: SglHandler failed, terminating.'
      end if
```

```
      Session = UsNum()
      call PName (MyName)
      running = .TRUE.
      do while (running)
          write (1,*) '> '
          read (1,*) operand, data1, data2
          if (operand .eq.  0) then
               running = .FALSE.
          else if (operand .eq.  -1) then
               running = .FALSE.
               call SglKill (5hSERVR,Session,SglUsr1,buf,14)
          else
               call SglKill (5hSERVR,Session,SglUsr1,buf,14)
C             Wait for a SglUsr1
              if (SglPause(oldMask) .ne.  0) then
                  stop 'CNSMR: SglPause failed, terminating.'
              end if
              write (1,*) 'Result =', result
          end if
       end do
       end
```

CNSMR waits for the result sent by SERVR in a SglUsr1 signal. The following is a segment of CNSMR's signal handler that will receive and handle the expected signal.

```
          .
          .
          lda SigNum       ;Get the signal number.
          cpa =d16         ;Was it SglUsr1?
          jmp expected     ; Yes.
          jmp unexpect     ;Unexpected signal.

; Received an expected signal, take care of it.
expected lda SigData
          sta Result
          .
          .
```

CNSMR's handler defines a common area for result so the main program can access it.

```
          .
          .
          result   alloc common,1
          .
          .
```

The program SERVR defines a common data area, SigBuf, which it shares with its signal handler and into which is placed the signal dependent data sent in the buf parameter by CNSMR.

```
ftn7x,q,s
      program servr
      implicit none
      integer SglUsr1, noabort
      parameter (SglUsr1=16, noabort=100000b)
      external handler

      integer*4 mask, oldMask, SglSetMask
      integer SglPause, SglHandler
      logical running, trySignal
      integer result, a, b
      character*2 aStr, bStr
      equivalence (a,aStr), (b,bStr)
      integer signum, siglen
      common /SigBuf/ signum, siglen
      integer sender(3), session
      common /SigBuf/ sender, session
      integer action, data1, data2
      common /SigBuf/ action, data1, data2

C     Block signal SglUsr1
      mask = 2j ** (SglUsr1 - 1)
      oldMask = SglSetMask(mask)

C     Set up a signal handler
      if (SglHandler(handler) .ne.  0) then
          stop 'SERVR: SglHandler failed, terminating.'
      end if

      running = .TRUE.
      do while (running)
C         Wait for a SglUsr1
          if (SglPause(oldMask) .ne.  0) then
              stop 'SERVR: SglPause failed, terminating.'
          end if

          if (SigLen .lt.  14) then
              write (1,*) 'SERVR: Not enough data'
          else
              trySignal = .TRUE.
              if (action .eq.  -1) then
                  running = .FALSE.
                  trySignal = .FALSE.
              else if (action .eq.  1) then
                  result = data1 + data2
              else if (action .eq.  2) then
                  result = data1 - data2
              else if (action .eq.  3) then
                  result = data1 * data2
```

```
                else if (action .eq.  4) then
                    result = data1 / data2
                else
                    write (1,*) 'SERVR: Bad request.'
                    trySignal = .FALSE.
                end if

                if (trySignal) then
                    call exec
                         (37+noabort,3,sender,session,16,result,2,*100)


                    goto 200
100                 call abreg(a,b)
                    write (1,*)
     +                   'SERVR: SglKill failed, reason = ', aStr, bStr

200                 end if
                end if
            end do
        end
```

After SERVR has set up a signal handler, it uses the SglPause function to unblock and wait for a SglUsr1 signal. When its handler receives the expected signal, control passes to SERVR. Notice that the handler does not have to take any action because the operating system has already placed the information in the common area. More information on the common area will follow. The following is a segment of SERVR's signal handler that receives and handles an expected signal.

```
            .
            .
            lda SigNum      ;Get the signal number.
            cpa =d16        ;Was it SglUsr1?
            jmp expected    ; Yes.
            jmp unexpect    ;Unexpected signal.

; Received an expected signal, SDD data already transferred, return.
expected nop
            .
            .
            .
```

SERVR's handler also defines the buffer for the signal dependent data, compare this section to the part of SERVR where the common variables are declared.

```
            .
            .
            ; Define the buffer for the signal dependent data (SDD).
            SigBuf    alloc common,9
            SigNum    equ SigBuf+0
            SigLen    equ SigBuf+1
            Sender    equ SigBuf+2     ;The name of the sender,
```

```
Session  equ SigBuf+5     ; their session number,
Action   equ SigBuf+6     ; and what to do.
Data1    equ SigBuf+7
Data2    equ SigBuf+8
SigEnd   equ SigBuf+9
.
.
```

SERVR obtains the signal dependent data from the common data area and computes a result. Then SERVR returns the result as the signal dependent data parameter in the EXEC call equivalent to the SglKill subroutine. SERVR uses an EXEC call so that the noabort option bit can be set.

CNSMR prints the result to the screen and returns with the prompt. The user can enter −1 as the operand variable (in addition to any other values for data1 and data2) to stop CNSMR and SERVR.

Program Example Using SglSetJmp

Signals can be used to develop a program that allows a series of executable steps to be attempted until one is successful. The following FORTRAN program example, ft, and its associated signal handler demonstrate this. The SglSetJmp subroutine is used in conjunction with the EXEC call equivalent of the SglLongJmp subroutine.

For example, if the first program step should generate an error, such as by making an EXEC call without parameters, a SglVio signal is sent to the program's handler. The handler returns control to the program to attempt the next program step.

```
ftn7x,q,s
      program ft
      implicit none
      external handler

      integer SglSetJmp, SglHandler
      integer env(3)
      common /env/env

C     Set up a signal handler
      if (SglHandler(handler) .ne.  0) then
          stop 'SERVR: SglHandler failed, terminating.'
      end if

C     Save the environment
      if (SglSetJmp(env) .eq.  0) then
          write (1,*) 'Trying method 1'
          call exec(0)
          write (1,*) 'Method one succeeded'
      else
          write (1,*) 'Trying method 2'
```

```
          write (1,*) 'Method two succeeded'
      end if
      end
```

The example program begins by setting up a signal handler. The SglSetJmp function returns a value of zero so the program proceeds to make an exec(0) call that results in the SglVio signal being sent by the operating system.

The handler is written to expect a SglVio signal. The handler alters the contents of the A-register so that when the program resumes execution, the SglSetJmp function appears to return a non-zero value causing execution to resume at the else statement.

```
          .
          .
          lda SigNum       ;Get the signal number.
          cpa =d4          ;Was it SglVio?
          rss              ; Yes.
          jmp unexpect     ;Unexpected signal.
          lda SigData      ;RQ error?
          cpa =s'RQ'
          rss              ; Yes.
          jmp unexpect

; Received an expected signal, take care of it.
expected lda =d1           ;Put a nonzero value in A, so the main
                           ; can tell that we aren't SglSetJmp
          jsb Exec         ;Return back to the saved environment.
          def *+4
          def =d37
          def =d5
          def Env
          .
          .
```

The handler for ft also defines env to be found in the common area.

```
          .
          .
env       alloc common,3
          .
          .
```

When the expected SglVio signal is received, the EXEC equivalent to SglLongJmp is performed without first restoring the interrupted environment and without specifying the environment buffer that was given to the handler when it was called. Instead, the environment buffer that was saved by SglSetJmp is specified. This allows transfer of control to another part of the program.

Timer Signals

The timer signal provides each program with one interval timer. This timer acts as a sort of alarm clock which, at the end of the specified interval, notifies the program that its timer has expired. Each individual program's timer interval is absolute. A program's timer is still active when the program is in a suspended state.

Signal Handler for Timer Signals

The signal handler is the routine to which the operating system passes control when the timer interval has expired. If no handler exists when a timer expires, the system aborts the program with an SG07 (Unexpected Signal) error code.

Functional Characteristics

The timer signal gives the user the following functionality:

1. The ability to set a timer for a program.
2. The ability to examine the amount of time remaining for the current interval.
3. The ability to modify the timer interval of a program, that is, to reset the interval.
4. The ability to cancel the current interval before the signal handler is entered.

Using Timer Signals

The first step in using timer signals is to inform the operating system of the address of a signal handler that is to be set up. Using EXEC (37,1,*handler*) or the SglHandler(*handler*) subroutine establishes the handler.

Once the signal handler is set up, a timer with the desired interval can be set up using the SetTimer subroutine. The program's execution continues until the timer has expired, whereupon the operating system passes control to the signal handler. The timer can be reset from the signal handler with an EXEC 38 call.

Timer Subroutine Calling Sequences

This section describes the interfacing routines to the timer signal. In the following section, an integer is assumed to be 16 bits, a double integer is assumed to be 32 bits.

SetTimer

SetTimer is an integer function that establishes a new timer or resets an existing timer. SetTimer returns zero if no error occurs, −1 if no interval is specified.

```
error = SetTimer(interval)
```

where:

interval is a double integer indicating the number of ticks between 0 and $2^{32}-1$ before the signal is generated. Each timer tick takes 10 ms. An interval of zero causes a timer signal immediately.

A program can have only one active timer in effect at any time. The caller will be XSAM suspended if there is insufficient XSAM to create a new timer. You can use the EXEC call interface with the no-suspend bit set to prevent XSAM suspension.

KillTimer

KillTimer is an integer function that cancels the current timer for the calling program. It returns zero if the timer is canceled, −1 if no timer exists for the calling program.

```
error = KillTimer( )
```

QueryTimer

QueryTimer is an integer function that returns the number of ticks remaining before a timer signal is to be generated for the calling program. It returns zero if no error occurs, −1 if no timer exists for the calling program.

```
error = QueryTimer(ticks)
```

where:

ticks is a double integer that, upon return, contains the number of ticks remaining for the calling program's interval. Each tick has a value of 10 ms.

EXEC 38

An EXEC 38 call sets up and modifies an interval timer for the calling program. The system sets, resets, queries or kills an interval timer for the caller.

```
CALL EXEC (ecode , option , interval)
```

where:

ecode is 38 for all interval timer related system calls.

option is an integer variable that specifies the type of timer operation to perform. The values of *option* are as follows:

0 = SetTimer/ResetTimer
This indicates that the caller wishes to set up a new timer or reset an existing timer. If no interval timer exists when the call is made, a new interval timer is created. Otherwise, the existing timer is reset.

1 = QueryTimer
This specifies that the caller wishes to know how many ticks remain before an interval timer expires (if one exists).

2 = KillTimer
This specifies that the current interval timer be terminated (if one exists) without entering the signal handler.

interval is a double integer that is set to the duration (number of ticks) of the timer in the case of SetTimer calls. It should be set to zero for KillTimer calls. QueryTimer calls return the number of ticks until the timer expires in this variable.

Parameter Relationships

When a SetTimer operation is performed (*option* = 0) and no XSAM is available, the caller is XSAM suspended (state 63b) until enough XSAM becomes available to establish the timer. As in all EXEC calls, the no-abort/no-suspend options are available by setting the appropriate bits of the *ecode* parameter.

If no XSAM is available and the no-suspend bit was set, the call aborts with an SG04 error. If any of the three parameters are missing (*ecode*, *option*, or *interval*), the call aborts with an RQ00 abort code.

A SetTimer call with a length of zero (*option* = 0, *interval* = 0), generates a timer signal instantly. No timer is set up. Instead, the signal handler is entered immediately.

A- and B-Registers

For successful EXEC 38 calls, the A-Register and B-Register contents are unchanged. For unsuccessful calls, the A- and B-Registers return error information. This information is described in the EXEC Error Processing section in the Introduction chapter in this manual.

Interval Timer Example

The following example illustrates the use of timer signals. The program prints a stream of X's to the terminal (LU 1) until the timer expires. Once the timer expires, a timer signal is generated and the signal handler is entered. Here the timer is reset and the handler prints the letter 'O' to the screen. Below is the code for the example described above. The main routine is coded in FORTRAN. The signal handler must be coded in Macro.

```
ftn7x,q,s,m,t

      program example

      implicit integer (a-z)
      external Clock
      logical ifbrk

c     Set up a signal handler for the timer signal. When a timer
c     expires, it will enter the handler at the entry point called
c     CLOCK.

      call Exec (37,1,Clock)     ! Set up a signal handler

c     Set an interval timer for 100 ticks.  Each tick takes 10ms so
c     the interval here is 100*10ms or 1 second.

      call Exec (38,0,100j)      ! Set a timer with 1 sec.  delay

c     At this point, the program goes into a loop printing X's to
c     the screen until the timer expires ( 1 sec.  ) later.  Once
c     the timer expires, the signal handler CLOCK is entered.  Here
c     the timer is reset and an O is printed to the screen.


      do while ( .not.  ifbrk() )
          call Exec (2,2001b,1hX,-1)    ! Print a stream of X's
      end do

      end

*
*     This section starts the Signal Handler used by the above
*     FORTRAN example program.
```

```
*
macro,q,c,s
         nam clock
         ent clock
         ext Exec

EnvBuf   equ *          ;Place to save environment while in signal
PC       bss 1          ;handler.
Q        bss 1
CodeSeg  bss 1

SigNum   bss 3

         def EnvBuf
         dec 3
         def SigNum
         dec 3

*
* Enter here when timer expires.
*

Clock    dst ABSave     ;Save A, B, E, O, X, Y
         era,als
         soc
         ina
         sta EOSave
         stx XSave
         sty YSave

         lda SigNum     ;Make sure its a timer signal
         cpa =d14       ;If timer,
         rss            ; do it.

         hlt 77b        ;Generates unexpected signal.

         jsb Exec       ;Reset the timer for 1 sec.
         def *+4
         def =d38
         def =d0
         def dd100

         jsb Exec       ;Print out an 'O' to the screen
         def *+5
         def =d2
         def =b2001
         def MSG
```

```
            def =d1

            ldx XSave       ;Restore the registers before returning
            ldy YSave
            ldb EOSave
            clo
            slb,elb
            sto
            dld ABSave

            jsb Exec        ;Long Jump back to the main program
            def *+4
            def =d37
            def =d5
            def EnvBuf

dd100       dec 0           ;double integer value of 100
            dec 100
MSG         asc 1,O

ABSave      bss 2
EOSave      bss 1
XSave       bss 1
YSave       bss 1

            end
```

14

Programmatic Environment Variable Access

Environment variables allow programs within a session to share variables with CI and other programs. Setting environment variables from CI is described in detail in the *RTE-A User's Manual*, part number 92077-90002. This chapter describes how these environment variables can be accessed programmatically with an EXEC 39 call. Refer to the *RTE-A User's Manual* for a list and definition of predefined environment variables.

EXEC 39 Call

User programs can access individual environment (exported) variables by name by issuing an EXEC 39 call. The following are the four subfunction codes (the second parameter of the EXEC 39) and are described in the following sections:

code = 1	Get the value of an environment variable.
code = 2	Set an environment variable to a given value.
code = 3	Delete an environment variable.
code = 4	Retrieve the modification count.

Environment variable names are described using FORTRAN string descriptors in parameters *name* and *value* in the EXEC 39 call. The variable name starts at the first byte and ends at the character before the first space, the thirty-second character, or the length given in the string descriptor, whichever comes first. For the *name* parameter, a zero-length string descriptor or a name that starts with a space causes an EV02 error (illegal parameter value). When used as a name, the entire string descriptor may not be used.

Getting the Value of a Variable

An EXEC 39 call with the second parameter equal to 1 retrieves the value of an environment variable that has been set by a previous EXEC 39 call (see the Setting a Variable section) or by the SET command from CI.

```
CALL EXEC(39,1,name,value)
```

where:

name is a FORTRAN string descriptor that specifies the name of the variable to be retrieved.

value is a FORTRAN string descriptor that receives the blank-filled value of the named variable.

Returns:

A-Register = status information; see the A-Register Return section at the end of this chapter.

B-Register = if the call was successful, the actual length, in characters, of *value* is returned. Otherwise, it equals zero.

Setting a Variable

An EXEC 39 call with the second parameter equal to 2 sets an environment variable to a given value.

```
CALL EXEC(39,2,name,value)
```

where:

name is a FORTRAN string descriptor that specifies the name of the variable to be defined.

value is a FORTRAN string descriptor that specifies the value that should be assigned to the named variable.

Returns:

A-Register = status information; see the A-Register Return section at the end of this chapter.

B-Register = undefined.

Caution CI also stores aliases and functions in the Environment Variable Block (EVB). The aliases and functions are distinguished from environment variables by a ':' after the alias or function name. Using an EXEC 39 call to set a variable name that has a colon in the name, produces unpredictable results in CI.

Deleting a Variable

An EXEC 39 call with the second parameter equal to 3 deletes an environment variable that has been set by a previous EXEC 39 call (see the Setting a Variable section) or by the SET command from CI.

```
CALL EXEC(39,3,name)
```

where:

name is a FORTRAN string descriptor that specifies the name of the variable to be deleted.

Returns:

A-Register = status information; see the A-Register Return section at the end of this chapter.

B-Register = undefined.

Retrieving the Modification Count

An EXEC 39 call with the second parameter equal to 4 returns the modification count in the B-Register.

```
CALL EXEC(39,4)
```

Returns:

A-Register = status information; see the A-Register Return section at the end of this chapter.

B-Register = the modification count.

The modification count is incremented when a program begins to write into the environment variable block and again when it finishes. This is useful for a program that keeps track of the values of a large number of environment variables. The program would get the modification count before it gets the value of an environment variable and then can get the modification count again later to see if any values have changed. If the value of the modification count is different, one or more variables have been changed and/or deleted. The program should then retrieve values again for all environment variables of interest.

An even modification count indicates a stable environment. If the modification count is odd, this indicates that the environment variable block has been locked (via the resource number) by a program in order to change it.

A-Register Return

After an EXEC 39 call, the A-Register contains status information. The possible values of the A-Register and their definitions are:

0	Operation successful.
1	Operation successful except that the buffer for *value* provided by the calling program was not large enough to hold the entire value of the environment variable. The B-Register contains the length of the actual value of the environment variable that resides in the environment variable block.
2	Named variable not found.
3	Session does not contain an environment variable block.
4	There is no space left in the environment variable block to store the variable. If EXEC(39, 2, ...) is called to set an environment variable and this error occurs, the variable retains its prior value.
2hEV	An EXEC error has occurred. The remainder of the error code is in the B-Register.
2hOP	%ENVRN is not generated into the system. The B-Register contains 2h39.

There are five possible EXEC errors:

OP39	%ENVRN is not generated into the system.
EV00	Invalid environment variable block.
EV01	Incorrect number of parameters.
EV02	Illegal parameter value.
EV04	Environment variable block is busy (no-suspend bit set).

These errors can be trapped by the calling program by setting the no-abort bit. The alternate return from EXEC will be taken and the A- and B-Registers will contain the first two and last two ASCII characters of the EXEC error code respectively.

If a program is already accessing the environment variable block when an EXEC 39 call is made by a second program, the second program will be resource number suspended. When the first program finishes accessing the environment variable block, the second program will be scheduled again. To avoid being suspended, both the no-suspend and no-abort bits in the *ecode* parameter should be set. This causes the alternate return to be taken if the program would normally have been suspended. In this case, the A- and B-Registers contain the error code EV04.

A

Error Messages

This appendix contains EXEC call, operating system, and FMP error codes. For language or subsystem error codes, refer to the manual for that language or subsystem.

When the system discovers an Executive (EXEC) error, it normally terminates the program, releases system resources assigned to the program, issues an error message to the system console and to the error log file if used, then proceeds to execute the next program in the scheduled list.

The user may specify the no-abort bit for some EXEC error conditions. See the parameter *ecode* in an EXEC request description for a detailed discussion of this option.

The error messages described below are those that may occur while accessing the Executive. They are grouped according to type.

The format of an EXEC error message is as follows:

```
PROG aborted at address 26606 Reason is RQ Current segment= 0
A=100700 B=     0 X=     0  Y=      0  E=      1 O=      0 WMAP=106002
Instruction=177777 Q=      0  Z=       0  CS mode=OF
RQ = bad or too many EXEC parameters
```

This error message gives the program name, location, reason the program aborted, and contents of the program registers.

Group I error messages are errors returned by an operator command or MESSS call. These errors are described in the *RTE-A User's Manual*, part number 92077-90002.

Note: Displayed values of X, Y, Z, and Q-Registers and CS-Mode for LD, SW, and parity errors are not valid.

Group II Errors

Group II errors are not affected by the no-abort bit in the *ecode* parameter in EXEC calls.

An EXEC request that has an illegal request code or that has more than eight parameters is rejected. The message:

```
PROG aborted at address 26606 Reason is RQ Current segment=    0
A=      3  B=  1401 X=      0 Y=      0 E=      1 0=    0 WMAP= 102100
Instruction=177777 Z=      0 Q=      0 CS mode=OF
RQ = bad or too many EXEC parameters
```

This error message gives the program name, location, reason the program aborted, and contents of the program registers.

An RQ error also can occur if the no-abort is not set and a program issues an REIO or XREIO request to a disk LU without specifying the track and sector parameters.

If an instruction in a user program is executed which the computer does not recognize, the program is aborted, and an error message in the above format is printed on the user terminal. The error message is in the format shown above, except for the reason code and text:

```
UI unimplemented instruction in user area
```

The execution of an unimplemented instruction is most commonly caused by one of the following problems:

1. The program was coded for an HP 1000 computer with a greater instruction set. (Many of these instructions can be simulated by software subroutine calls as described in the *Macro/1000 Reference Manual*, part number 92059-90001.)

2. The program has a logic error causing it to execute data.

If a disk read or write request fails repeatedly (5 times consecutively) on a program load or swap, the program is aborted and the message: (in the format shown under RQ, with the following reason code and text)

LD disk tried five times to load program/segment and failed
or
SW disk tried five times to swap program and failed

is displayed on the user terminal. LD indicates that the program load (possibly a segment load or swap in) failed. SW indicates that the program swap out failed.

Memory Protect Violations

The operating system is protected by a hardware memory protect. Any instruction that attempts to modify memory on a write-protected page, is rejected.

This causes a memory protect interrupt and the program is aborted and the following message is displayed on the system console.

```
PETST  aborted at address  26606 Reason is MP Current segment = 0
A=     3  B=  1401 X=     0 Y=     0 E=    1 0=     0 WMAP= 102100
Instruction=177777 Z=     0 Q=     0 CS mode=OF
```

where:

`MP`	=	memory protect error; I/O instruction or store/jump to protected memory
`address`	=	the offending program counter value
`segment`	=	the number of the most recently loaded segment, or zero for non-segmented programs.

A memory protect abort also occurs when a program attempts to execute a privileged instruction (STC, HLT, OTA, etc.) without calling $LIBR first.

Attempting to cross store (JSB or .XSA) before calling $LIBR also results in a memory protect abort. The routine IXPUT calls $LIBR before performing its cross store, so you do need not to call $LIBR before using IXPUT.

SR Errors

In this section, the following definitions apply:

A Level 1 ($LIBR) routine operates with the interrupt and memory protect system off and is generally used to modify the operating system tables. (Modifying the operating system is not recommended.) A level 1 routine is sometimes called a privileged routine. Normally, execution time should be limited to a few milliseconds.

A Level 2 routine calls the routine .ZPRV or DispatchLock to keep other programs from calling the routine while it is executing. A level 2 routine is sometimes called a privileged routine. Normally execution time should be limited to a few milliseconds.

A Level 3 routine calls .ZRNT to keep other programs from interfering with the routines execution. A level 3 is sometimes called a re-entrant routine. It must be coded in assembly language as a type 6 module. One copy of the routine may be placed in system common (when the system is generated) to be shared by all the programs in the system. Level 3 routines cannot be called from CDS code.

A Level 4 routine is not shareable, and must be appended to each program that calls it. A level 4 routine may be coded in assembly language, FORTRAN, or Pascal as a type 7 module. A level 4 routine is sometimes called a utility routine. Main programs are also considered to be level 4.

The following list shows system subroutine levels; level 4 subroutines are not listed.

Subroutine	Level	Subroutine	Level
CLRQ	3	IXPUT	1
CNUMD	2	KCVT	2
CNUMO	2	LIMEM	1
DTACH	2	LURQ	3
EQTRQ	1	PARSE	2
EXEC	3	PRTM	1
IDCLR	1	PRTN	1
INPRS	2	RNRQ	3
IPUT	1	SAVST	1
IXGET	1	TMVAL	1

If a program attempts to call a routine that is on a higher numbered level than the currently executing routine, the program is aborted (exception: level 1 routines may call level 2 routines) and the following message is displayed:

```
PROG aborted at address 26606 Reason is SR  Current segment=     0
A=     3  B=  1401   X=     0  Y=     0 E=    1 0=    0 WMAP= 102100
Instruction=177777   Z=     0  Q=     0 CS mode=OF
SR = privileged subroutine call error
```

Incorrect coding of shareable subroutines can also cause this error.

An SR error also occurs if a level 1 or 2 CDS routine causes a code segment fault while it is privileged.

Dispatching Errors

Two errors can occur when you load a program into memory in order to run it. The system aborts the program and prints one of the following error messages to the scheduling terminal.

SC09 Program too large to fit in largest usable block of dynamic memory ever available.

EM90 Shareable EMA size for program is larger than the shareable EMA area that has already been allocated.

The second message indicates that another program using the same shareable EMA has already run and that the shareable EMA was allocated according to its size need. To avoid this problem, specify the same EMA size to all programs using the same shareable EMA area, either to LINK or by operator command.

Group III Errors

Errors in this group are affected by the no-abort (NA) bit in the EXEC request that caused the error as follows:

1. NA bit clear: the program is aborted. The following message is printed on the invoking terminal:

```
PROG aborted at address 26606 Reason is RN02 Current segment= 0
A=      3    B=  1401 X=     0 Y=     0 E=     1  0=    0 WMAP=102100
Instruction=177777  Z=     0 Q=     0 CS mode=OF
RN02 = undefined resource number
```

2. NA bit set: The program is not aborted. A four-character ASCII message is returned in the A- and B-Registers. The first two characters will be in the A-Register and the second two in the B-Register. The possible error message is shown in the pages following. The return address to the program is specified in the word following the JSB EXEC. (If no error occurs, the return address is one greater than the address specified in the call.) No message is displayed on the console. In FORTRAN the modified error return is handled as described in the EXEC Call Error Returns section in Chapter 1.

The following lists relate errors to the program calls that may cause them.

Error	Meaning	EXEC Calls
SC01	Not enough parameters for EXEC schedule call	11,14
SC02	Illegal parameter value in EXEC schedule call	12,14
SC03	SECURITY VIOLATION detected:	
	Insufficient capability to schedule program	9,10,12,23,24
	or	
	Attempted to access an LU that is not in the session LU access table	1,2,3,17,18,19 20

Error	Meaning	EXEC Calls
SC04	Illegal buffer or not a son in EXEC schedule call.	6,8,9,10,11, 12,14,23,24
SC05	EXEC schedule call; program PROGA not found.	8,9,10,12, 23,24
SC06	Overlay attempted from a CDS program.	8
SC09	Program is too large to fit in memory.	
SC10	Not enough SAM to pass string parameter.	9,10,14,23,24
SC15	Not enough memory (SAM) to pass string. (No-suspend bit set)	9,10,14,23,24

The following messages apply to all programs:

Error	Meaning	Call
RN00	No option bits set in call;	RNRQ
RN02	Undefined resource number	RNRQ
RN03	Can not clear RN not locked to a program or invalid RN.	RNRQ
LU02	Illegal logical unit:	LURQ
CL01	Illegal class number or no class table	CLRQ
CL02	Illegal class number or no class table	CLRQ

CDS (Code and Data Separation) errors:

Error	Meaning
CS00	CDS software not installed.
CS01	Segment load requested by instruction in data segment.
CS02	CST index out of bounds.
CS03	Invalid SST entry.
CS04	Too many indirects.
CS05	CDS program is corrupt, internal error found.
CS06	Stack overflow. Link program with more stack space.

Environment Variable Block Errors: The following errors can be caused by EXEC 39 calls which are discussed in Chapter 14:

Error	Meaning
EV00	Invalid environment variable block.
EV01	Incorrect number of parameters.
EV02	Illegal parameter value.
EV04	Environment variable block is busy (no-suspend bit set).

Signal Errors: The following errors are used for signals which are discussed in Chapter 13:

Error	Meaning
SG01	Illegal number of parameters on an EXEC call.
SG02	Illegal parameter on an EXEC call.
SG03	Cannot use SglPause while in handler.
SG04	Not enough XSAM at this time. May occur when calling SglHandler.
SG05	Not enough capability to deliver the signal when using SglKill.
SG06	Buffer limit suspended when using SglKill.
SG07	Unexpected signal received.
SG08	No such program as identified by a call to SglKill.
SG09	Illegal buffer descriptors when the operating system attempted to deliver a signal. Not possible to trap with no-abort.

Option Errors

Depending on the installation and application, the operating system can be generated with a number of optional modules. These modules enable certain EXEC calls to become executable. If the system returns a no EXEC capability regenerate RTE-A with the specified module. Refer to the *RTE-A System Design Manual*, part number 92077-90013, for details on the specified module.

Requests to these optional EXEC calls which have not been included in the system will result in an error message of the following form:

```
No EXEC nn capability
```

where:

nn is the number of the EXEC called.

These errors can be trapped by the program by setting the no-abort bit in the ECODE parameter of the EXEC call. If the no-abort bit is set, the error will be returned to the program, instead of causing the program to be aborted.

I/O Errors

The errors below cause the program to be aborted unless the no-abort option was used in the EXEC call.

IO00 Illegal EXEC call Class Number. Either the user has not yet received a valid class number by setting the CLASS parameter to zero or the class number was passed to another program, and the passer has been aborted, or the system has no Class Table.

IO01 Not enough parameters in EXEC call, I/O call, or illegal access to the disk, or illegal disk subfunction specified for a non-disk device.

IO02 Illegal logical unit or Class call to disk in EXEC call or illegal class request to disk.

IO04 Illegal buffer address in EXEC I/O call. If the buffer address and its length are either above or below the program area then the address is an illegal address. This error also occurs if the Class request is for more memory than will ever be available in SAM.

The Z-bit in a rethread request does not match the Z-bit setting in the original request.

The rethread buffer length is greater than the buffer length in the original request. This applies to the Z-buffer length as well.

No rethread buffers exist when the rethread request was made. The class defined by the parameter OCLAS was not valid.

IO07 Driver has rejected EXEC call. The driver was requested to perform an action on a device that was incorrect for the device or driver.

IO10 Illegal Class Get; two gets by two programs at once. This condition can happen when two different programs both attempt a Class Get on the same class number at the same time. The first request is satisfied while the second is aborted.

IO11 Attempt to input from spooled logical unit. While spooling, a program made an EXEC 1, 17, 20, or 3 (with a subfunction of 6; that is, a dynamic status request) request to the spooled device.

IO12 I/O request denied; session user has attempted to access an LU that is not listed in the session user's LU access table defined by GRUMP.

IO13 LU is locked.

IO14 LU is down.

Group IV Halt Errors

Halt (HLT) instructions indicate a serious violation of the integrity of the operating system. Sometimes they indicate that the CPU 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 halts occur, check out your hardware and software with the appropriate diagnostics.

When a halt is executed, the Virtual Control Panel (VCP) is invoked. It displays the message:

```
Pxxxxxx Axxxxxx Bxxxxxx Mxxxxxx T1020nn
```

where *nn* is the halt number.

The following HLTs may occur:

HLT 0 Either BOOTEX has not initialized the system or else there are more than 8 pages of OS/Driver partition space. You should run BOOTEX to initialize the system properly or else decrease the size of the OS/Driver partition space.

HLT 1 The system has not been initialized by the BOOTEX program.

HLT 2 A privileged routine has executed location 2. This may occur if data is executed or a link is set to zero.

HLT 3 A Group II or III error has occurred when no program was executing, or when executing location 5 (parity error handling) when a parity error has not occurred.

HLT 4 A CPU power failure has occurred and the power fail driver was not included in the system.

HLT 5 If the CPU is an A400, A600, A600+ or A700, a CPU parity error has occurred in the system, system common areas of memory, or unused partition. If the CPU is an A900 or A990, any parity error causes a HLT 5. If the parity error handling module (PERR) was not specified at generation time, any parity error will cause a HLT 5. The B-Register will contain the page number, and the A-Register will contain the parity error location.

HLT 26 BOOTEX cannot boot the system. This may be caused by a missing system or snap file. The halt message will most likely be preceded by a message that gives more information.

HLT 27 Memory management or program management tables have been corrupted, and the system cannot continue. The system must be rebooted.

HLT 42 An internal error occurred when a port map for a driver was being allocated or deallocated. The system must be rebooted.

Group V Interrupt Errors

When an interrupt occurs on a channel for which there is no driver, the interrupt flag is cleared, and the message is displayed (xx is the select code):

```
Illegal interrupt from SCxx
```

This message is reported when an interrupt occurs from a select code which is not in RTEs table of select codes that are supposed to interrupt. This usually indicates that there was a generation error in select code specification.

At generation time, a program can be set up to handle an interrupt directly. Whenever the interrupt occurs, RTIOA attempts to schedule the program. If the program is already busy, the message:

```
Unable to schedule program PROGN on interrupt to driver
```

is reported. This indicates the RTEs interrupt table specifies a program to run on interrupt but that program was not dormant when the interrupt came in.

Group VI Device Driver Errors

The following I/O errors can occur when a particular device driver detects an error condition. The format of the error message is:

```
I/O Device Error on LU xx error_description
```

where:

xx = the logical unit number

error_description = one of the following:

```
I/O request error
Device not ready
Device timed out
End of tape detected
Transmission error
Device is write protected
Addressing error
Serial poll error
Group poll error
Drive fault
Data communication error
Device information specified at generation time is wrong
```

`Special Driver Defined Error =` *yy*
where *yy* is driver specific. This error is reported when the system does not recognize the error reported by the driver. Refer to the section for your driver in the *RTE-A Driver Reference Manual*, part number 92077-90011, for specific error information.

`Request has been flushed`
Reported in addition to the above messages when the offending I/O request will not be retried.

Group VII Parity Errors

If a CPU parity error occurs in the mapped part of memory (in a partition not in the system common nor in the system area), the system will abort the program that encountered the parity error and issue the following message:

```
Parity Error occurred at physical page 132 page address 63
Pages actually marked bad (downed) = 2
PE = Parity Error in User code or data space
```

If a parity error occurs in the system or common area or out of a user partition which has I/O outputting from it, or if the parity error handling module (PERR) was not specified when the system was generated, a HLT 5 will occur with B = bad page and A = location of the PE on the bad page.

Group VIII VMA/EMA Errors

There are several types of VMA/EMA errors. In the first category are run-time errors generated by a VMA/EMA subroutine/microcode call which always result in a program abort. In the second category are run-time errors generated by a VMA/EMA subroutine/microcode call which will always return to the user with some kind of error indicator. In a third category are run-time errors which are no-abort-bit sensitive; VMAIO is the only VMA/EMA routine of this type. Finally, there are dispatching errors (EM90, EM91) that can occur when the dispatcher is attempting to run a program that uses shareable EMA.

VMA/EMA errors that cause the program to abort have the same format as the MP error and look like:

```
VMxx or EMxx
```

where:

xx is an FMP error number (if xx is less than 80). FMP reports the error as a negative number and VMA/EMA reports the last two digits of that number.

It is possible that VMA/EMA will report different FMP errors by the same VMA/EMA error; for example, either FMP−006 (No such file) or FMP−206 (Directory read protected) is reported as VM06. The VMxx error reported depends on the context. Examine the current status of the system to determine which of the several possible FMP errors may have been reported.

VM01 Disk error.

VM02 Duplicate file name.

VM05 Backing store file created with less than 256 blocks of memory or file extent cannot be created when read only access has been specified to the VMA file. X-register equals the requested page ID that caused the problem.

VM06 File not found or directory read protected.

VM07 Illegal security code or illegal write on LU 2 or 3 or directory write protected.

VM08 File open or cartridge containing file is locked.

VM09 No such directory (FMP-209).

VM12 File extent cannot be created when read only access has been specified to the VMA file. X-Register equals requested page that caused problem.

VM13 Specified cartridge is locked.

VM14 Directory full.

VM15 Illegal file name.

VM19 Illegal access on a system disk.

VM20 An array is specified with incorrect subscripts.

VM21 MSEG in the $EMA or $VMA directive is not specified correctly.

VM22 The program is not an VMA/EMA program.

VM32 Cartridge not found.

VM33 Not enough room on cartridge.

VM46 Greater than 255 file extents on the VMA file.

VM80 VMA system is corrupt. Self checks on the data structure did not pass a sanity check, therefore the system aborted. Probable cause is by a program exceeding a non-VMA/EMA array boundary.

VM81 Program is not a VMA program. Relink the program using the LINK VM command.

VM82 The requested page is beyond the maximum page specified for VMA or the disk file is not big enough. A VMA array boundary has been exceeded in your program. The X-reg is requested page number (in octal), Y-reg is logical address to map in the requested page, abort address is address of the instruction causing the problem, where first page of VMA/EMA is page 0.

VM83 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 operator WS command.

VM84 File type not = 2 or record length not = 1024 words. The file specified in the OPNVM or CREVM call was not type 2 with record length = 1024 words.

VM85 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.

VM86 Access to VMA system after the VMA file has been closed. Revise program to not access the VMA area after a CLSVM or PURVM call has been made.

VM87 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 or the number of pages specified to be mapped was zero. The program must be revised to use less MSEG area or make the MSEG bigger.

VM88 Cannot respecify the VMA file. Revise the program to call the OPNVM or CREVM routine no more than one time, if the first time was successful.

VM89 Transfer too large for VMAIO.

VM90 Shareable EMA size for program is larger than the shareable EMA area already allocated.

VM91 Program and shareable VMA/EMA area are assigned to the same partition; or, a program is assigned to a reserved partition in which the program's VMA/EMA area has already been allocated.

VM92 VMA or VMAIO not available.

EM80 EMA system is corrupt. Self checks on the data structure did not pass a sanity check, therefore the system aborted. Probable cause is by a program exceeding a non-EMA array boundary.

EM81 This program is not an EMA program. Link the program again using the LINK EM command.

EM82 The requested page is beyond the maximum page specified for the EMA system. An EMA array boundary has been exceeded. Probable bug in your program. The X-reg = requested page number (in octal), Y-reg = logical address to map in the requested page, abort address = address of the instruction causing the program, where first page of VMA/EMA is page 0.

EM87 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 or the number of pages specified to be mapped was zero. The program must be revised to use less MSEG area or make the MSEG bigger.

EM89 Transfer too large for VMAIO.

EM90 Shareable EMA size for program is larger than the shareable EMA area that has already been allocated. This is an error detected on an attempt by the dispatcher to run the program; some other program using the same shareable EMA area has already been run, and the shareable area was allocated according to its size need. On detection, the program is made dormant and the error message given.

EM91 Program and shareable EMA area are assigned to same partition or program is assigned to a reserved partition in which program's shareable EMA area has already been allocated.

EM92 EMA not available.

FMP Error Codes

−001 Disk error!

The disk is down; try again and then report it to the system manager of facility.

−002 File already exists

A file already exists with specified name; repeat with new name or purge existing file.

−003 Backspace illegal

Attempt was made to backspace a device (or type 0 file) that cannot be backspaced, check device type.

−004 Record size illegal

Attempt to create a type 2 file with a zero record length.

−005 Bad record length

Attempt to read or position to a record not written, or on update to write an illegal record length; check position or size parameters.

−006 No such file

Attempt to access a file that cannot be found. Check the file name or cartridge number.

−007 Incorrect security code

Attempt to access a file without the correct security code. Use the correct code or do not access file.

−008 File is already open

Attempt to open file already open exclusively or open to eight programs or cartridge containing file is locked; use CL or DL to locate lock.

−009 Must not be a device

Type 0 files cannot be positioned or be forced to type 1; check file type.

−010 Not enough parameters

Required parameters omitted from call; enter the parameters.

−011 DCB is not open

Attempt to access an unopened DCB. Check error code on open attempt.

−012 Illegal file position

Attempt to read or write or position beyond the file boundaries; check record position parameters, result depends on file type & call.

−013 Disk is locked

Cartridge is locked; initialize cartridge if not initialized, otherwise, keep trying.

−014 Directory is full

No more room in file directory; purge files and pack directory if possible, or try another cartridge.

−015 Illegal name

File name does not conform to syntax rules; correct name.

−016 Size = 0 or illegal type 0 file access

Wrong type code supplied; attempt to create or purge type 0 file or create 0-length file; check size and type parameters.

−017 Device I/O failed

Attempt to read/write or position type 0 file that does not support the operation; check file parameters, namr.

−018 Illegal LU.

Attempt to access an undefined LU.

−030 Value too large for parameter

Value is greater than legal maximum.

−032 No such cartridge

Specified cartridge is not mounted. Check disk specification in call.

−033 Ran out of disk space

Disk specified for a disk file has insufficient room for file create. Could occur during a WRITF if an extent is being created.

−034 Disk is already mounted

Disk is mounted as an FMGR or hierarchical volume.

−035 Already 63 disks mounted to system

Only 63 disk LUs may by mounted at one time.

−036 Lock error on device

A call to OPEN or OPENF specified exclusive use of a device which was already locked or no resource numbers were available. Try again or request nonexclusive use.

−037 Program is active

A request to purge an active type 6 file was requested by PURGE. The program must be offed before the file can be purged. The swap file cannot be purged if swapping is enabled.

−038 Illegal scratch file number

The legal range of scratch file numbers is 0-99. Check your program.

−046 More than 255 extents

An attempt to create more than 255 extents was made. Use a file with a larger initial size.

−049 Copy verify failed

The verify option of the COPYF routine detected a discrepancy while verifying a transfer of data. Check the file for correctness.

−050 No files found

A "−" was specified in a namr, but there were no files matching the mask. Check the mask for correctness.

−051 Directory is empty

The specified directory contains no files.

−053 Program assigned to bad partition

The program (for non-CDS programs) or the data partition (for CDS programs) is assigned to a reserved partition which is "bad" due to a parity error in the partition or a reserved partition which is undefined. Use the AS command to re-assign the program (or the AS command with the "D" option to re-assign the data partition) to a good partition.

−054 Partition too small for program

The program (for non-CDS programs) or the data partition (for CDS programs) is assigned to a reserved partition which is not large enough to hold the program or data partition. The program or data partition must be assigned to a larger reserved partition or dynamic memory.

−055 No room in shareable EMA table

Insufficient free XSAM exists to create SHEMA table entry.

−056 SHEMA assigned to non-existent partition

The shareable EMA area used by the program is assigned to a reserved partition which was not defined (by the AS or RV command) at system bootup time. The program must be reloaded to change the shareable EMA assign number or the system must be rebooted to define the partition. (Remember that the first program RP'd that uses a shareable EMA area determines where it is allocated. Perhaps another program that uses the shareable EMA area could be RP'd first.)

−057 Partition too small for shareable EMA

The shareable EMA area used by the program is assigned to a reserved partition which is not large enough to hold it. If all the programs that access the shareable EMA area do not specify the same shareable EMA size, this error could result.

−058 Program assigned to SHEMA partition

The program (for non-CDS programs) or data partition of the program (for CDS programs) is assigned to the same reserved partition as the shareable EMA area the program accesses. Both must be in memory for the program to run, so one must be re-assigned to a different reserved partition or dynamic memory. This error could result if the first program that uses that shareable EMA area assigns it to a reserved partition in which a second program that accesses it is assigned to run.

−059 Already 255 programs using SHEMA area

There are already 255 programs RP'd that access the shareable EMA area specified by the program.

−060 Code & data assigned to same partition

Both the code partition and the data partition are assigned to the same reserved partition. They must be in distinct partitions. This error applies only to operating systems with VC+ Enhancement Package.

−063 Code assigned to non-existent partition

The code partition of the program is assigned to a reserved partition which is "bad" due to a parity error in the partition or a reserved partition which is undefined. Use the AS command with the "C" option to re-assign the code partition to a good partition or dynamic memory.

−064 Partition too small for code segment

The program's code partition is assigned to a reserved partition which is not large enough to hold it. The code partition must be assigned to a larger reserved partition or dynamic memory.

−068 Code assigned to shareable EMA partition

The program's code partition is assigned to the same reserved partition as the shareable EMA area the program accesses. Both must be in memory for the program to run, so one must be re-assigned to a different reserved partition or dynamic memory. This error could result if the first program that uses that shareable EMA area assigns the area to a reserved partition in which the code partition of a second program that accesses it is assigned to run.

−099 D.RTR EXEC request aborted

D.RTR has tried something unreasonable, probably because the cartridge list has been corrupted.

−101 Illegal parameter in D.RTR call

Possible operator error; recheck previous entries for illegal or misplaced parameters.

−102 D.RTR not available

D.RTR is not RP'd or has been offed; system should be rebooted.

−103 Directory is corrupt

During a directory lock done by MC, DC, IN, PK, CR, or PU, the directory is scanned for internal consistency. If this occurs, copy the files to another disk, or just store the ones you need.

−104 Missing extent

A request was made for a file extent which was missing from the file. The file is probably corrupt. Purge the file.

−105 D.RTR must be sized up

D.RTR uses free space for open flags and global directories, and must be sized up when loaded.

−108 Illegal number of sectors/track

The disk LU being mounted has a defined number of sectors per track greater than 128.

−200 No working directory

Returned by FmpWorkingDir when there is no working directory established, and by some other calls when a file name is specified with no directory but no working directory exists.

−201 Directory not empty

Directories can only be purged when they are empty. To purge the directory, purge the remaining files (use a wildcard purge).

−202 Did not ask to read

This file is read-protected. Specify the R option in the open request.

−203 Did not ask to write

This file is write-protected. Specify the W option in the open request.

−204 File read protected

This file is read-protected or is a write-only device. Change the protection on the file.

−205 File write protected

This file is write-protected or is a read-only device. Either the file has write protection set (in which case you should change the protection on the file), or it has a positive security code which needs to be specified correctly in the open call.

−206 Directory read protected

One of the directories needed to access the file is read-protected. Change its protection.

−207 Directory write protected

The directory containing the file is write-protected, so you cannot change its properties, purge it, or rename it.

−208 Duplicate directory name

That name already being used. Be sure the directory is being created where you expect it to be.

−209 No such directory

One directory needed to find the file does not exist. Its name may be misspelled, or the working directory may be wrong.

−210 Unpurge failed

Disk space or a directory entry occupied by the purged file has been reclaimed, so the file cannot be unpurged. Not repairable.

−211 Directories not on same LU

Rename operations do not move data, and data must be on the same LU as the directory, so rename operations can only rename a file into a directory on the same LU as it was originally.

−212 Cannot change that property

Rename operations cannot change whether the file is a directory, nor can they change the file type, size, or record length.

−213 Too many open files

D.RTR has no room to record the open flag for this file. Close some files or dismount a volume for temporary relief; a long-term solution is to size D.RTR larger, open fewer files, or have fewer global directories.

−214 Disk not mounted

The indicated volume was not mounted, so it cannot be dismounted and directories cannot be created on it.

−215 Too many directories

D.RTR has no room to record this global directory; this error can occur on mount or directory create. Close some files or dismount a volume for temporary relief; a long-term solution is to size D.RTR larger, open fewer files, or have fewer global directories. Perhaps some global directories can be renamed as subdirectories.

−216 You do not own

Only the file owner can change its protection information, and only the directory owner can change the file owner. Superusers do not get this error; become a superuser to avoid this problem.

−217 Bad directory block

Tag fields in the directory do not match, indicating a corrupt disk or working directory pointer. Change working directories. If that fails, investigate the situation with the file system status utility.

−218 Must specify an LU

FmpCreateDir could not determine where to create this directory. Either supply an LU, or set the working directory to a directory on the LU where the new directory is to be created.

−219 No remote access

The passed name or DCB indicates that this file is located on a (possibly) remote system, so it must be routed through the DS transparency software before it is usable.

−220 DSRTR not available

The DS transparency source monitor is not RP'd, so DS transparency does not work. RP DSRTR.

−221 Files are open on LU

This LU cannot be dismounted because one or more files are open. The name of the first open file is printed by D.RTR.

−222 LU has old directory

This LU has an old directory, and FmpMount was not told to re-initialize old directories.

−223 Illegal DCB buffer size

DCB buffer sizes must be in the range one to 127 blocks, except for type zero and one files, which ignore the size. This error is also returned by routines such as FmpCopy when the passed buffer is too small.

−224 No free ID segments

Cannot restore the program, due to lack of ID segments. Remove programs that are no longer needed.

−225 Program is busy

FmpRunProgram reports that the program named in the XQ command is busy.

−226 Program aborted

The program was OF'd or aborted before it ran to completion.

−227 Program doesn't fit in partition (SC08/09)

The program is too big for available memory or the partition to which it is assigned. Unassign the program or assign it to a bigger partition.

−228 No SAM to pass string (SC10)

The system does not have enough SAM to pass runstrings. Rebooting may help if SAM is fragmented, or you may need to regenerate the system to allocate more SAM.

−229 Active working directory

Tried to purge a working directory or dismount a disk containing a working directory.

−230 Illegal use of directory

A directory was used illegally (e.g., to create a file).

−231 String is too long

A string longer than 256 bytes was passed to FmpReadString or FmpWriteString.

−232 Unknown for FMGR file

Requested unavailable information (e.g., time stamp) about an old file.

−233 No such user

User name not found by FmpSetOwner.

−234 Size mismatch on copy

Source and destination file sizes for FmpCopy are incompatible.

−235 Break flag detected

An FMP routine detected a break sent by the BR command.

−236 You are not a superuser

Normal user used a command reserved for the superuser.

−237 Must not be remote

A file was specified with a remote system name or account in a situation where such names are illegal. This error is reported even if the node specifies (or defaults to) the local system.

−238 Illegal program file

The file named is illegal because:

- It is not a program file.
- It accesses system entry points outside the table in %VCTR and is being RP'd to a system other than the one for which it is linked.
- It was linked with an incompatible version of %VCTR.

−239 Program name exists

Cannot RP program with that name because another program already has it. OF the old program with the ID parameter, or choose another name for the new program.

−240 Changed RPL checksum

The program file was linked with a snap file that specified different microcoded instructions (RPLs), or the same instructions in a different order. If the program uses nonexistent instructions, it aborts with the message UI (unimplemented instruction), and the program file changes to prevent this error from being reported again on the current system. (This change to the program file does not solve the problem, but the program may work anyway.)

−241 Can only run unshared

Shared program cannot be run because there is no room in the shared program table. Use LINK to make the program unshared, OF programs to make room in the shared table, or regenerate the system with more shared program entries.

−242 Disk I/O failed

D.RTR got an EXEC error when attempting to access a disk LU.

−243 Parameter error

An actual parameter has an unreasonable value.

−244 Mapping error

An error occurred while a VMA file routine was mapping VMA.

−245 System can't do CDS

Tried to RP a CDS program on a non-CDS system.

−246 System common changed

Tried to RP a program that defines system common differently than it is defined on the current system.

−247 UDSP not defined

The UDSP is not defined because of one the following reasons:

- None of the entries in the UDSP have been defined.
- The requested UDSP number and entry has not been defined.
- The given UDSP number and entry is beyond the bounds defined for the account.

−248 Invalid directory address found

UDSP tables are corrupt.

−249 SECURITY VIOLATION detected

The Security/1000 subsystem has detected an attempt by the user or program to perform a function for which the user or program has insufficient capability.

−250 D.ERR not available

The system program D.ERR, which is used to generate FMP error messages, cannot be scheduled because D.ERR was not RP'd or it was OF'd. You should RP D.ERR.

−251 Program name exists in another session

An attempt has been made to RP a program in the system session while a program of the same name is already RP'd in another session.

−252 Disk LU is down

D.RTR tried to access a disk LU that is down.

−253 Disk LU is locked

D.RTR tried to access a disk LU that is locked to another program.

−254 No such group

Group name not found by FmpSetOwner.

−255 User is not in group

The user is not in the group specified as the user group parameter for FmpSetOwner.

−256 No such session

From FmpRpProgram.

−257 No such program

From FmpRpProgram.

−258 No SAM for Proto ID

Not enough SAM/XSAM to create a proto ID segment. Rebooting may help if SAM/XSAM is fragmented, or you may need to regenerate the system to allocate more SAM/XSAM.

−260 Too many symbolic links in path

D.RTR traversed more than 8 symbolic links. This is probably a closed symbolic link loop.

−261 Symbolic link results in illegal path

Symbolic link interpretation yields a path name greater than 63 characters. Symbolic links that contain relative path names can be changed to absolute paths.

−262 Symbolic link must not be remote

An attempt was made to access a symbolic link on a remote system and the remote symbolic link referred to a remote file. Create a symbolic link on the local system that refers to the destination of the remote symbolic link being accessed.

−263 System does not support symbolic links

At attempt was made to create a symbolic link on a system that does not support symbolic links. A CDS version of D.RTR is required and programs accessing symbolic links must be linked with either $SFMP or $SCDS.

−270 Update time already current

From FmpCopy.

The following error codes reflect errors in DS transparency software.

−300 Illegal remote access

Usually means an internal error. Either an invalid connection number was specified, or an invalid request was routed to the DS transparency software.

−301 Too many remote connections

No more than 64 files can be open at remote systems at any one time. Each open file requires a connection. You can reclaim connections by closing files.

−302 No such node

The local system does not know anything about the node number or the name specified. It may not be in the NRV.

−303 Too many sessions

Cannot log on the remote system because too many other sessions are already logged on.

−304 No such account

No user has that name.

−305 Incorrect password

The correct password was not supplied.

−306 Can't access account

A logon error occurred that was not one of the above three.

−307 Transfer is too long

A request was made to DSRTR to transfer more than 1024 words.

−308 Connection broken

The remote system monitor TRFAS was restarted since the connection was open.

The following errors are reported by DS software; see the DS manuals for more details.

-310 DS is not initialized [DS00]

DS has not been started with DINIT

-311 DS link is not connected [DS01]

Hardware problem.

-312 Remote system doesn't respond [DS05]

Other system is probably down, or not running DS.

-313 No TRFAS at remote system [DS06]

Remote monitor TRFAS is not RP'd at remote system.

-315 DS error DSXX(X), node YY

Something happened not included in the above. The DS error code is reported.

FMP Errors 401 – 410 are related to native language support utilities. If your system does not have the native language support utilities, the following errors may still occur if the message catalog file on the system for a utility is not of the same revision as the utility itself.

-401 Message number not found in catalog

-402 Message too big for message buffer

B

Converting FMGR File Calls

This appendix describes a step-by-step procedure to convert FMGR calls to FMP calls. FMGR calls are equivalent to the FMP calls of other RTE operating systems. The conversion procedures have been written to assist in converting programs that may be unfamiliar to the user.

General Considerations

File system calls usually make up a small percentage of a program, so the conversion effort is minimal since in most cases the program logic should not have to be changed.

Many of the FMGR calls will still work although it is recommended that programs be converted to allow full usage of the enhancements available with the FMP calls.

Programs that are to be transportable to other RTE operating systems should follow a different approach that is discussed later in this chapter.

Note The FMP calls do not have optional parameters. All parameters in the FMP calls must be supplied.

File and Directory Names

File and directory names can contain up to 63 characters, allowing for a full name including all directories and the ASCII versions of type, size, and so on. For example:

```
/POPULATION/CITIES/CALIFORNIA/SANJOSE.TXT:::4:24      (48 characters)
```

File names should be stored in 32-word character buffers if they are supplied as input to the program. This ensures consistency between programs. Because names are passed as character strings, it is possible to use a smaller buffer for file names that are embedded in the program. FMP calls work with unparsed names, so the 32-word buffer replaces the 10-word namr used by FMGR.

Global directory names contain up to 16 characters, and can be stored in 8-word character buffers. A subdirectory is treated as part of the file name by the supplied parsing routines. The global directory name can be specified as a prefix, as in the following example:

```
SOURCE/CMDS::USER:3    or    /USER/SOURCE/CMDS:::3
```

Constructs such as /FILE::DIR produce undefined results.

The directory name can appear in either of two places: to the left of any subdirectories or after two colons to the right of the file name. Use the following conventions to determine where to print the directory name:

- If no subdirectories are specified, print the directory name after the two colons, as in GRIDLOCK.RUN::PROGRAMS.
- If one or more subdirectories are specified, print the directory name as a prefix to the subdirectory name, as in /FAMILY/GENUS/SPECIES.TXT.

It is recommended to use file names in FMP calls after the file is opened because many of the FMP calls work with file names. File names are also useful in reporting errors.

Namr Calls and Strings

Namr calls that parse file names must be replaced, but be careful not to change namr calls used for different purposes. Namr calls that are used only to set up calls to Open, Create, and Purge can be removed, as the new equivalents of these calls do not require parsed file names. Calls that break apart file names for purposes of examining individual components can be replaced with a call to FmpParseName in most cases. FmpParseName does not indicate what type the subfields were and does not parse up to a comma the way that Namr does.

FmpParseName does not completely replace Namr. Other useful routines include: SplitString, which divides a character string at a blank or comma, and DecimalToInt, which converts a character string to a single integer. Fparm does runstring parsing, returning the file names in a runstring as separated character variables. (Fparm is not available to Pascal users.) These routines can be found in the *RTE-A • RTE-6/VM Relocatable Libraries Reference Manual,* part number 92077-90037.

Examples:

Here is an example of code that opens two files whose names are passed in the runstring:

```
call getst(buffer,-80,len)
start = 1
if (namr(pbuf,buffer,len,start) .lt. 0) goto 900
type1 = open(dcb1,err,pbuf,0,pbuf(5),pbuf(6))
if (err .lt. 0) goto 920
if (namr(pbuf,buffer,len,start) .lt. 0) goto 900
type2 = open(dcb2,err,pbuf,0,pbuf(5),pbuf(6))
if (err .lt. 0) goto 920
```

This can be replaced by:

```
file1 = ' '
file2 = ' '
call fparm(file1,file2)
if (file1 .eq. ' ' .or. file2 .eq ' ') goto 900
type1 = fmpopen(dcb1,err,file1,'ro',1)
if (err .lt. 0) goto 920
type2 = fmpopen(dcb2,err,file2,'ro',1)
if (err .lt. 0) goto 920
```

Note that Namr was not used.

The next example shows a sequence without character strings. It illustrates constructing string descriptors, which are the double integer (integer*4) variables in the following example. The function STRDSC takes parameters of buffer, starting character, and number of characters, and returns a string descriptor. Here it is used to create string descriptors for the file name and option strings (a constant 'ROS'):

```
integer*4  strdsc,string,file1,file2,options
call getst(buffer,-80,len)
string=strdsc(buffer,1,len)
file1=strdsc(buffer1,1,64)
file2=strdsc(buffer2,1,64)
call  splitstring(string,file1,string)
call  splitstring(string,file2,string)
if (blankstring(file1) .ne. 0 .or. blankstring(file2) .ne. 0))
   goto 900
options = strdsc(3hROS,1,3)
type1 = fmpopen(dcb1,err,file1,options,1)
if (err .lt. 0) goto 920
type2 = fmpopen(dcb2,err,file2,options,1)
if (err .lt. 0) goto 920
```

String descriptors describe strings by identifying where they can be found and how big they are. Once a string descriptor is set up, it can be used indefinitely. The buffer it points to can be changed through the string descriptor or through direct changes. In the above example, 'splitstring' changes the referenced buffer, and 'blankstring' tests for an all blank string.

The file system assigns default values for type and size when the file is created. The following example shows how the user would change the type and size values. In the example, the code sequence, constructs the name of a debug file from the name of a type 6 file, according to the following rule: if the type 6 file name has a .RUN type extension, create a file with the same name and a .DBG extension; otherwise, create a file with the same name but insert an at sign (@) in front of it, because this is a FMGR file. Make the file type 1, block size 96:

```
character pname*64, name*64, dir*16, typex*4, ds*64

call fmpparsename(pname,name,typex,sc,dir,d,d,d,ds)
if (typex .eq. 'RUN') then
  call fmpbuildname(pname,name,'DBG',sc,dir,1,96,0,ds)
else
  call fmpbuildname(pname,'@'//name,typex,sc,dir,1,96,0,ds)
endif
```

Open and Openf Calls

All Open and Openf calls are replaced by FmpOpen calls. Considerations include file name parsing and character string handling, described above. Beyond that, the user should be aware of how options and buffer sizes are specified. For example, the FMGR call:

```
type = open(dcb,err,pbuf,0,pbuf(5),pbuf(6),256)
```

specifies exclusive open for reading and writing (assuming the security code matches), with no other unusual options. It uses a 256-word DCB buffer, so the DCB should be declared as 256+16=272 words.

To get the same effect with FMP calls, the call would be:

```
type = fmpopen(dcb,err,name,'rwo',2)
```

Note that character options 'rwo' have been specified. Reading and writing are specified by 'rw'. The 'o' option means it is okay to open a FMGR file, but not to create a new one. (This is discussed more under create.) Other options available and their octal equivalents in the option word of the old open call are:

1: shared access: 's'
2: update mode: 'u'
4: force to type 1: 'f'
10: supply subfunction: no equivalent, see FmpSetIoOptions
20: (not defined)
40: permit extents: 'x'

The option word must be specified in an FMP call. In upgrading a call, start with 'rwo', then add the other options as necessary to get the way the call used to look. For example, an option word 45B: open, permitting type 1 and 2 extents, forced to type 1 and shared, would be option word 'rwoxfs'. The option characters can come in any order. If it is known that the file will be used only for reading or only for writing, omit the 'w' or 'r' respectively. Use the shared option if the file will be read only; it should not be used for writing unless the user provides his or her own synchronization.

Note that the buffer size is specified in blocks, rather than in words. The buffer size to supply is the old one divided by 128:

```
type = fmpopen(dcb,err,name,options,256/128)
```

This parameter must be supplied; if the FMGR call did not supply a buffer size, use a value of 1.

FmpOpen works like Openf when a logical unit number is passed in, but the logical unit number must be a string. For example,

```
type = fmpopen(dcb,err,'6','wo',1)
```

is correct, but

```
type = fmpopen(dcb,err,6,'wo',1)
```

does not work, because the logical unit number is an integer, not an ASCII string. If the logical unit is non-interactive, FmpOpen will try a logical unit lock with wait unless the file is opened shared.

Note that FMP files can be opened to a large number of programs (more than 7), but there must be room in the internal table of D.RTR for the open flag. If there is not room, the open will fail. One program can have the same file open several times (if each open call specifies a shared open). Repeated, exclusive opens of the same program work on FMGR files, but not on FMP files.

Readf and Writf Calls

For sequential files (type 3 and above, and type 0), Readf calls are replaced by FmpRead calls, and Writf calls are replaced by FmpWrite calls. They work much as the FMGR calls work, except that lengths are passed in and returned as byte lengths, not word lengths. The length read is returned only as a function value, so calling FmpRead as a subroutine will probably not produce the desired results.

For example:

```
call  readf(dcb1,err,buffer,128,len)
if (err .lt. 0) goto 900
call  writf(dcb2,err,buffer,len)
if (err .lt. 0) goto 910
```

is replaced by

```
len = fmpread(dcb1,err,buffer,256)
if (err .lt. 0) goto 900
call fmpwrite(dcb2,err,buffer,len)
if (err .lt. 0) goto 910
```

Now len is in bytes. If the program is expecting to use words, you can either change the program to deal with byte lengths (including odd byte lengths), or you can convert len to words:

```
if (len .ne. -1) len = (len+1)/2
```

End-of-file is reported as err = 0, len = −1. Do not try to use FmpWrite with a length of −1 to write an explicit end-of-file, as this will write 0 bytes (see below).

For random access files (type 1 and 2), Readf and Writf calls are converted to an FmpPosition call followed by an FmpRead or FmpWrite call. The straightforward way to do this is to position via (double-integer) record number; this is requested by using an internal position parameter of (double-integer) −1. (See FmpSetPosition for details.)

For example:

```
call  readf(dcb1,err,buffer,len,dummy,rrec)
if (err .lt. 0) goto 900
call  writf(dcb2,err,buffer,len,wrec)
if (err .lt. 0) goto 910
```

is replaced by

```
integer*4 drec

drec = rrec
call  fmpsetposition(dcb1,err,drec,-1J)
if (err .lt. 0) goto 900
call fmpread(dcb1,err,buffer,len*2)
if (err .lt. 0) goto 900
drec = wrec
```

```
call  fmpsetposition(dcb2,err,drec,-1J)
if (err .lt. 0) goto 910
call  fmpwrite(dcb2,err,buffer,len*2)
if (err .lt. 0) goto 910
```

The user should be careful not to pass single integers to FmpSetPosition. A called subroutine cannot determine what kind of integer was passed, so FmpSetPosition will use the single integer as the upper half of a double integer.

Close Calls

Non-truncating calls to Close can simply be replaced by calls to FmpClose:

```
call close(dcb)    →    call fmpclose(dcb,err)
```

Pass the error parameter, even if no error can occur. Fmpclose stores a value through the error parameter.

Truncating closes require two or three calls, depending on whether or not the user knows the truncation size. The sequence to truncate a file at the current position used to be:

```
call  locf(dcb,err,rec,block,offset,size)
if (err .lt. 0) goto 900
tblocks = (size/2) - (block+1)
call close(dcb,err,tblocks)
if (err .lt. 0) goto 900
```

Now it is:

```
integer*4 record, position, newsize
call  fmpposition(dcb,err,record,position)
if (err .lt. 0) goto 900
newsize = (position+127)/128 for type 3, (position/128)+1
call  fmptruncate(dcb,err,newsize)
if (err .lt. 0) goto 900
call  fmpclose(dcb,err)
if (err .lt. 0) goto 900
```

Note that the old way specified the number of blocks to truncate, while the new way specifies the desired file size. The new way truncates extra extents, which was not possible before. All sizes are double integers. There is no call provided for this sequence because it is not common.

Note that truncating to zero size does not purge the file. It leaves a one-block file.

Creat and Crets Calls

All Creat and Crets calls are replaced by FmpOpen calls that specify the 'c' option, meaning it is okay to create the file. Refer to the description above about Open and Openf to get the basics. Additional size, type, and record length information is passed as ASCII, appended to the name; FmpBuildName is useful for creating ASCII strings. Refer to the example under file name parsing for more information.

Any options used in an open call can be specified when creating a file. Previously, 'creat' set up default options of nonshared update mode, so to create the old environment, use the string 'rwcu'.

For example:

```
call  creat(dcb,err,pbuf,pbuf(8),pbuf(7),pbuf(5),pbuf(6))
if (err .lt. 0) goto 900
```

is replaced by

```
call  fmpopen(dcb,err,name,'rwcu',1)
```

This will give an error −2 if the file exists. The user can specify both 'o' and 'c'; this will open the existing file, or create a new one if necessary. Note that this sequence of creates followed by opens can be replaced by a single FmpOpen call:

```
call  creat(dcb,err,pbuf,24,3,pbuf(5),pbuf(6),256)
if (err .eq. -2) then
   call open(dcb,err,pbuf,0,pbuf(5),pbuf(6),256)
endif
if (err .lt. 0) goto 900
```

is replaced by

```
call  fmpopen(dcb,err,name,'rwoc',2)
if (err .lt. 0) goto 900
```

Support for scratch files consists of a way to create a name that is unique and a bit that indicates that this file is not important. The program that creates the scratch file has the responsibility to purge it when it is done. The file system does not automatically purge scratch files, although a wildcard purge of all scratch files can be specified. This eases the problem of having scratch files disappear when they are closed briefly.

To create an extendible type 1 scratch file with a starting size of 24 blocks, the old sequence would have been:

```
call  crets(dcb,err,0,name,24J,1,sc,cr)
if (err .lt. 0) goto 900
call open(dcb,err,name,40b,sc,cr)
if (err .lt. 0) goto 900
```

This is replaced by:

```
call fmpuniquename('TEMP',name)
call fmpopen(dcb,err,name//':::1:24','rwctx',1)
if (err .lt. 0) goto 900
```

The 't' option specifies this is a scratch file. Note that this file goes on the working directory. This only causes a problem if the working directory is currently on a small or slow disk, when a larger or faster disk is available elsewhere. One possible solution is to create the file on directory SCRATCH or some such special name, then try again on the working directory if the special directory does not exist.

In this example, the unique name has a prefix 'TEMP'. This is of no special significance, except that some prefix must be supplied to keep the name from seeming to be a number. If there is a chance, the scratch file will go on an old cartridge, then the prefix should be short (one character) to keep from getting duplicate six-character names. In any case, the name must be kept around to purge the file.

Aposn, Locf, and Posnt Calls

File positioning is also discussed in the section covering random access Readfs and Writfs. Aposn and Locf position by internal file pointers, while Posnt positions by record number. These functions are performed with FmpPosition and FmpSetPosition for FMP files.

Two position pointers are maintained for open disk files: a record number and an internal file position. The internal file position is the word offset from the first word of the file. To record the current record number and internal file position, use FmpPosition. Note that it always returns double integer values, even if single integers were passed. For example, the call:

```
call locf(dcb,err,record,block,offset)
if (err .lt. 0) goto 900
```

is replaced with

```
integer*4 drecord, dposition
call  fmpposition(dcb,err,drecord,dposition)
if (err .lt. 0) goto 900
```

The new internal position value is related to the old value: position = block * 128 + offset.

Use caution when changing Locf calls. They contain a lot of information, and it is not always easy to tell what is used and what is not. FmpPosition only returns file position. Other Locf information includes:

- FmpSize returns the total size of the file in blocks, rather than the size of the main part of the file in sectors.
- FmpEof indicates how much of the file is being used.
- FmpRecordLength returns file record length.
- FmpOpen returns file type when it opens the file.

There is no FMP call to return the logical unit of a file, because the logical unit cannot be used in place of the directory name.

To restore file position to a place recorded with FmpPosition, use FmpSetPosition. For example:

```
call aposn(dcb,err,record,block,offset)
if (err .lt. 0) goto 900
```

is replaced by

```
integer*4 drecord, dposition
call  fmpsetposition(dcb,err,drecord,dposition)
if (err .lt. 0) goto 900
```

This works for any type disk file. FmpSetPosition knows to use the internal position recorded by FmpPosition because the passed position is a non-negative value. If the position is negative, it is ignored and positioning is done by record number (see below). The record number parameter is only used to set up the record number in the DCB for use later by calls that position by record number.

FmpSetPosition is also used to position files by record number. Positioning type 1 and 2 files has already been discussed under Readf and Writf. Positioning type 0 and type 3 and above files was formerly done by Posnt. Posnt could position to an absolute record number, or to a record number relative to the current position. FmpSetPosition always positions to an absolute record number, however, relative positioning can be achieved by first using FmpPosition to see where you are, then adding the offset to get the absolute record number. (FmpSetPosition always positions relative to the current record number in the DCB, so if this is wrong you will not end up at the right absolute record number.) Remember that positioning sequential files by record number can be very slow.

For example, to position to absolute record 100, then skip backward 10 records:

```
call posnt(dcb,err,100,1)
if (err .lt. 0) goto 900
 .
 .
 .
call posnt(dcb,err,-10,0)
if (err .lt. 0) goto 900
```

This is replaced by:

```
integer*4 drecord, dposition
call  fmpsetposition(dcb,err,100J,-1J)
if (err .lt. 0) goto 900
 .
 .
 .
call  fmpposition(dcb,err,drecord,dposition)
if (err .lt. 0) goto 900
call  fmpsetposition(dcb,err,drecord-10,-1J)
if (err .lt. 0) goto 900
```

The −1J parameter passed as the file position indicates only the record number is to be used for positioning, as with type 1 and 2 files.

Purge and Namf Calls

Purge calls are replaced by FmpPurge calls, and Namf calls are replaced by FmpRename calls. The FMP calls do not work if the file is open to anyone, including the caller, so the file should be closed first. These calls do not require the caller to pass in a DCB.

Examples:

```
call  purge(dcb,err,pbuf,pbuf(5),pbuf(6))
if (err .lt. 0) goto 900
```

is replaced by

```
call  fmpclose(dcb,err)
err = fmppurge(name)
if (err .lt. 0) goto 900
```

and

```
call  namf(dcb,err,pbuf,newname,pbuf(5),pbuf(6))
if (err .lt. 0) goto 900
```

is replaced by

```
call  fmpclose(dcb,err)
err = fmprename(oldname,err1,newname,err2)
if (err .lt. 0) goto 900
```

Extended Calls

Extended calls (calls that start with E, for example, Eread, Ewrit, and Ecrea) are replaced in the same way as their non-extended equivalents. The FMP calls work with large files as a standard feature.

The creation of a file larger than 32767 blocks is slightly complicated. The user must pass in an ASCII file size that is the negative number of 128-block "chunks" in the file, so that a 50000 block file would be represented as −(50000+127/128) = FOO:::−391. This will really create a 50048-block file. Maximum file size is 32767 * 128 blocks, which is about 4 million blocks or 1 billion bytes.

Other Calls

FMP calls exist that perform the functions done by Rwndf, Post, and Fcont. Their names are FmpRewind, FmpPost, and FmpControl, respectively. Their use is not illustrated here, but is described in the section of this manual covering FMP routines.

FMP Calls/FMGR Files

This section describes what happens when an FMP call refers to a FMGR file.

This combination provides the same level of service as is obtained with FMGR calls referring to FMGR files. The caller can open, create, purge, and so on, files on old volumes. This is straightforward if the cartridge is specified, and if there is no new directory with this name. The cartridge can be specified as +CRN or −LU.

The following happens in cases other than the above:

If there is a new directory with the same name as an old cartridge, that cartridge cannot be accessed via the FMP calls, although it can be with FMGR calls. (FMP calls first check new directories, while FMGR calls first check old cartridges.) In general, it is confusing to have new directories with the same name as an old cartridge, so it is not recommended (although it is allowed).

If the directory is not specified: FOO or FOO:::3, and the user has a working directory, only that directory is searched. If the directory is explicitly specified as 0: FOO::0 or FOO::0:3, or if the directory is unspecified and the user has no working directory, then all of the old cartridges mounted to this user are searched. This is the only way to get a multiple disk search with the FMP calls, and it only searches old cartridges.

Calls that specify a file name only work with FMGR files if the information is available in the old directory. Thus, a user can get the name of a FMGR file, but cannot get the timestamps or position of end-of-file. In the latter cases, the old cartridges are not even searched, even if an old cartridge name is specified. Here is a summary:

These calls pass file names and work with FMGR files:

FmpOpen, FmpProtection, FmpPurge, FmpRename, and FmpSize

These calls pass file names and do not work with FMGR files:

FmpAccessTime,	FmpCreateDir,	FmpCreateTime,	FmpEof,
FmpRecordcount,	FmpRecordLen,	FmpSetOwner,	FmpSetProtection,
FmpSetWorkingDir,	FmpUnpurge,	FmpUpdateTime	

These calls do not pass file names, but do not work with FMGR files:

FmpOpenFiles, FmpSetDirInfo

Other calls that do not pass file names work with FMGR files.

Note If the directory name is found on an old cartridge, then the old rules for parsing namrs apply. Periods (.) and slashes (/) in names are not significant on old directories. The name is truncated to six characters.

Accesses to old directories use the old rules for things like open flags and extent creation. The same protection checks (for example, security code) are made, although it is not guaranteed that

all invalid requests will be caught (such as illegal characters in file names.) Old error codes are returned when appropriate.

Calls that specify a DCB work regardless of whether the file is old or new, including read, write, position, and so on. This includes files with extents and files with odd byte length records.

Standard Type Extensions

The following is a list of the standard type extensions.

.c	C source file
.cmd	command file
.dat	data file
.dbg	debug file
.dir	directory of subdirectory entry
.doc	document file
.err	eror message file
.ftn	FORTRAN source file
.ftni	fortran source include file
.h	C include file
.hlp	help file
.lib	indexed library of relocatables
.lod	LINK command file
.lst	listing
.mac	Macro source file
.maci	Macro source include file
.map	loader map listing
.merg	merge file for relocatables without headers
.mlb	Macro library file
.mrg	library merge file for relocatables with headers
.mnf	manual numbering file
.pas	Pascal source file
.pasi	Pascal source include file
.rel	relocatable (binary) file
.run	program file
.snp	system snapshot file
.spl	spooling system file
.stk	command stack file
.sys	system file
.txt	text file

Computer Museum

FMGR Calls

In addition to the FMP calls described earlier in this manual, there are other calls which work only with FMGR files. They use six character names, 2 character CRN's, etc. This chapter gives a detailed description of the FMGR routines. They are included here for compatibility only, and are not recommended for use in new applications. The routines are given in alphabetical order.

For each routine the FORTRAN calling sequence is shown. In FORTRAN, routines called as functions should be declared as single integers unless otherwise stated. See the Program Calls section for the general form of the Assembly Language calling sequence. All parameters are assumed to be integer variables unless otherwise noted.

APOSN (Position a Disk File)

This routine is called to position any disk file to a specific record. The record location may be determined by a prior call to LOCF.

APOSN is intended to position sequential files with variable length records prior to a read or write request. It may be used to position random access files with fixed length records (types 1 and 2) but it must not be used to position non-disk files (type 0). POSNT may be used to position type 0 files.

```
CALL APOSN(idcb,ierr,irec,irb,ioff)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

irec is the next record; 1-word variable set to number of next sequential record; may have been determined by a previous call to LOCF. Next record number must be in the range 1 to 32767.

irb is the next block; optional 1-word variable set to next block number; may have been determined by a previous call to LOCF; omitted only for files with fixed-length records. Next block number must be 16383 or less.

ioff is the next word; optional 1-word variable set to number of next sequential word in block (DCB); may have been determined by a previous call to LOCF; omitted only for files with fixed-length records. Next word number must be in the range of 0 to 127.

The record position parameters (*irb*, *ioff*) determine the position within the file of the record *irec*. They contain the block number and the word offset within the block where the record begins.

irec must be set; if not set to a specific record number by user, it may be set to a value returned by a call to LOCF. The three values *irec*, *irb*, and *ioff* may all be retrieved through LOCF. This permits the resetting of the file location to its position when LOCF was called.

Note Whenever a file with variable-length records is positioned, the two optional parameters *irb* and *ioff* must be included.

CLOSE (Close a File)

To close a file after use, call the CLOSE routine. The file remains in the system available to other programs following the close; the Data Control Block is freed for association with other files. A disk file opened for exclusive use of the calling program may be truncated to its actual length.

```
CALL CLOSE(idcb,ierr,itrun)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is the error return; a 1-word variable in which negative error code is returned if truncation is unsuccessful; required only when *itrun* is specified.

itrun indicates truncation; optional 1-word variable containing integer number of blocks to be deleted from the file at closing. If omitted or zero, the file is closed without truncation; if negative, only extents are truncated. *itrun* must be less than 16384 blocks.

Type 0 files: If the file being closed is a type 0 file, CLOSE will attempt to unlock it.

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 disk space.

A file may be truncated only if:

- The file is a disk file.
- The current position is in the main file, not in an extent.
- The file is not open to another program.
- The number of blocks deleted is 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 a 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.

The value of *itrun* when positive can be calculated from information returned by a previous call to LOCF, assuming the current position is at the end-of-file. In this case, the last block number written or read (*irb*+1) is subtracted from the total blocks with which the file was created (*jsize*/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 RWNDF.

A zero value for *itrun* is exactly the same as omitting this parameter; a standard closing is performed with no truncation.

CRDC (Dismount a Cartridge)

This routine dismounts a cartridge from the system.

ierr = CRDC (*icr*)

where:

icr is the cartridge identifier; a 1-word variable containing a positive cartridge reference number or negative logical unit number of the cartridge to be dismounted; must be mounted.

ierr is the error return code; 1-word variable will contain zero if no error was detected, or the FMP or FMGR error which was detected.

The cartridge will not be dismounted if it has an active program file (type 6) residing on it, or if it is locked to another program. Note that CRDC can return both positive and negative error codes.

CREAT (Create a File)

CREAT creates a file. It makes an entry in the file directory for the file and allocates disk space for the data.

Following execution of CREAT, the file is left open in the update mode for exclusive use of the program performing the call. If you want the file open in any other mode or for more than one program, use the OPEN call. Note, however, that it is not necessary to change the open mode for sequential access to the created file.

```
CALL CREAT(idcb,ierr,name,isize,itype,isecu,icr,idcbs)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words where *n* is positive or 0.

ierr is the error return; 1-word variable in which a negative error code is returned. If there is no error, it is set to the number of 64-word sectors (twice the number of 128-word blocks) in the created file.

name is a 3-word array containing the file name.

isize is the file size, maximum 16383 blocks; a 2-word array with number of blocks in first word; if −1, rest of cartridge is allocated to file; second word, used only for type 2 files, contains record length in words.

itype is the file type; 1-word integer variable in range 1-32767; types 1-7 are defined by FMP (see following), higher types are special purpose files defined by the user.

isecu is the security code; optional 1-word variable in range −32768 through 32767; if omitted, value is set to zero and file is not protected; positive value sets write protection only; negative sets read and write protection.

icr is the cartridge reference; optional 1-word variable; if omitted, space for the file will be allocated on the first cartridge having enough room; if positive, cartridge is identified by its cartridge reference number, if negative, by its logical unit number.

idcbs is the DCB buffer size; optional 1-word variable; number of words in DCB buffer if larger than 128; if omitted, FMP assumes DCB size (control words + buffer) is 144 words regardless of *idcb* dimensions.

When the exact size of the file is not known, an indefinite size can be specified by setting *isize* to −1. The rest of the cartridge, but not more than 16383 blocks, is allocated to the file in this case. Any area that is unused may be returned by using the *itrun* parameter when the file is closed. (Refer to CLOSE.) Note that a file using all the remaining cartridge is not extendible since a file may not cross cartridge boundaries.

When any file of type 3 or greater is created, CREAT writes an EOF mark at beginning of the file. As records are written to the file, an EOF is automatically written following the last record.

CRETS (Create a Scratch Disk File)

CRETS creates a temporary or scratch disk file; that is, it creates a file with a unique name and makes an entry in the File Directory for the file and allocates disk space to the file. This call only creates scratch files. Two of the optional parameters are 32-bit integers.

Following execution of CRETS, the file is left open in the update mode for exclusive use of the program performing the call. When terminating access to the file, use PURGE to purge the file. If the information in the file is to be made available for normal use, change the name of the file using the NAMF call, or copy the information to another file.

The file directory manager automatically cleans up scratch files that are no longer in use. This happens after the creating program closes the scratch file or terminates. On the next search through the file directory, the directory manager removes the file.

```
CALL CRETS(idcb,ierr,num,name,isize,itype,isecu,icr,idcbs,jsize)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words where *n* is positive or zero.

ierr is for error return; one-word variable in which a negative error code is returned.

num is the scratch file number between 0 and 99.

name is the created file name, a 3-word array into which CRETS returns the name of the file it has created. Save this if you will be using NAMF or PURGE.

isize is the file size; optional array of two 32-bit integers, with number of blocks in first element; if −1, rest of cartridge is allocated to file; second element, used only for type 2 files, contains record length in words. If omitted, the file is created with 24 blocks.

itype is the file type; optional 1-word variable in range 1-32767; types 0-7 are defined by FMP (see following) higher types are user-defined files; if omitted, the file is created type 3.

isecu is the security code; optional 1-word variable in range −32768 to +32767; if omitted, value is set to zero and file is not protected; positive value sets write protection only; negative value sets read and write protection.

icr is the cartridge reference; optional 1-word variable; if omitted, space for the file will be allocated to to the first cartridge having enough room; if positive cartridge is identified by the cartridge reference number, if negative, by the logical unit number.

idcbs is the DCB buffer size; optional 1-word variable; number of words in DCB buffer if larger than 128; if omitted, FMP assumes DCB size (control words + buffer) is 144 words regardless of *idcb* dimensions.

jsize is the actual file size; optional 32-bit integer variable; actual created file size (in sectors) is returned here if call is successful.

When the exact size of the file is not known, an indefinite size can be specified by setting *isize* to −1. The rest of the cartridge is allocated to the file in this case. Any area that is unused may be returned by using the *itrun* parameter when the file is closed. (Refer to the CLOSE section.)

The default for this parameter is 24 blocks.

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.

CRMC (Mount a Cartridge to the System)

This routine mounts a cartridge to the system.

ierr = CRMC (*lu* , [*lstrk*])

where:

lu is the cartridge logical unit number; 1-word variable; the positive or negative logical unit number of the cartridge to be mounted; must not be zero or already be mounted.

lstrk is the last FMP track; optional 1-word variable; the last track available for FMP use on the cartridge; if omitted, the last physical track on device containing the cartridge is used.

ierr is for error return; 1-word variable in which the FMP or FMGR error code is returned.

This routine also checks to see if the cartridge being mounted has a corrupt directory. If a corrupt directory is detected, a FMGR-103 error is returned in the A-Register, and the cartridge will be mounted, but locked to the program.

Note that CRMC will return both positive and negative error codes. For example, a FMGR 012 error will be returned if a duplicate cartridge mount attempt is made.

EAPOS (Extended Range Positioning)

The EAPOS routine is the extended range version of APOSN.

```
CALL EAPOS(idcb,ierr,irec,irb,ioff)
```

where:

idcb	is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.
ierr	is for error return; 1-word variable in which negative error code is returned.
irec	is the next record; 32-bit integer variable; next sequential record number.
irb	is the next block; optional 32-bit integer variable; block number containing next sequential record; omitted only for random access files.
ioff	is the next word; optional 1-word variable; offset within block containing next sequential record; must be in the range 0 to 127; omitted only for random access files.

Refer to APOSN for a description of EAPOS parameters and its sequence of operation.

ECLOS (Extended Close)

The ECLOS routine is the extended range version of CLOSE.

```
CALL ECLOS(idcb,ierr,itrun)
```

where:

idcb	is the Data Control Block; an array of 144 + (n * 128) words, where n is positive or zero.
ierr	is for error return; 1-word variable in which negative error code is returned.
itrun	is the truncation value; optional 32-bit integer; number of blocks to be deleted from the main file at closing; if negative, only extents are truncated; if omitted or zero, the file is closed without truncation.

See the CLOSE section for discussion of truncation and sequence of operations.

ECREA (Extended File Create)

ECREA is the extended range version of CREAT.

```
CALL ECREA(idcb,ierr,name,isize,itype,isecu,icr,idcbs,jsize)
```

where:

idcb	is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.
ierr	is for error return; a 1-word variable in which a negative error code is returned.
name	is a 3-word array containing the file name.
isize	is the file size; array of two 32-bit integers; first element contains the number of blocks; if set to −1, the rest of the cartridge is allocated to the file; second element is used only for type 2 files and contains the record length.
itype	is the file type; 1-word integer variable in range 1-32767; types 1-7 are defined by FMP, higher types are special purpose files defined by the user.
isecu	is the security code; optional 1-word variable in range −32768 to +32767; if omitted, code is set to zero and file is not protected; positive value sets write protection only; negative value sets read and write protection.
icr	is the cartridge reference; optional 1-word variable; if omitted, space for the file will be allocated to the first cartridge having enough room; if positive, cartridge is identified by the cartridge reference number; if negative, by the logical unit number.
idcbs	is the DCB buffer size; optional 1-word variable; number of words in DCB buffer if larger than 128; if omitted, FMP assumes DCB size (control words + buffer) is 144 words regardless of *idcb* dimensions.
jsize	is the actual file size; optional 32-bit integer; actual file size created in sectors is returned here if call is successful.

ELOCF (Extended LOCF)

ELOCF is the extended range version of LOCF.

```
CALL  ELOCF (idcb , ierr , irec , irb , ioff , jsec , jlu , jty , jrec )
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

irec is the next record; 32-bit integer variable in which the next sequential record number is returned.

irb is the next block; optional 32-bit integer variable in which the block number containing the next sequential record is returned.

ioff is the next word; optional 1-word variable in which the word offset within the block containing the next sequential record is returned; not returned for type 0 files; range 0 through 127.

jsec is the actual file size; Optional 32-bit integer variable in which the file size in sectors is returned; not returned for type 0 files.

jlu is the logical unit; optional 1-word variable in which logical unit to which file is allocated is returned.

jty is the file type; optional 1-word variable in which file type at open is returned.

jrec is the record size; optional 1-word variable in which record size of type 1 or type 2 files or read/write code of type 0 files is returned; not applicable to files with variable length records (type 3 and above).

For disk files parameters *irec*, *irb* and *ioff* contain the current position within the file. These parameters may be passed directly to EAPOS whenever the user wants to position the file back to this location. Note that ELOCF and EAPOS must be used together because for each, the record number and relative block position are 32-bit integer variables.

For further discussion of ELOCF functions and its sequence of operation, refer to the section on LOCF.

EPOSN (Extended Range Positioning)

The EPOSN routine is the extended range version of POSNT.

```
CALL EPOSN(idcb,ierr,nur,ir)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

nur is the number of records; 32-bit integer variable specifying the number of records to position forward if positive, backward if negative; if IR is included as a non-zero value, *nur* specifies the record number to which the file is positioned.

ir is an optional 1-word variable set to indicate that *nur* is interpreted as a record number; if omitted or zero, *nur* is treated as number of records to space forward or backward. Refer to the section on OPEN.

For further discussion of EPOSN functions and sequence of operations, refer to the POSNT section.

EREAD (Extended Range Read)

The EREAD routine is the extended range version of READF.

```
CALL EREAD(idcb,ierr,ibuf,il,len,num)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

ibuf is the user buffer; array into which record is read; it should be large enough to contain the record.

il is the length in words; optional 1-word variable specifying number of words to be read; should not be omitted for type 0 files. For other files, one record is read if *il* is omitted; if *il* is specified, it is a good idea to make it the same size as *ibuf*. Refer to READF for details of *il* use.

len is the number of words read; optional 1-word variable in which actual number of words read is returned; set to −1 if end-of-file is read; if omitted, information not supplied.

num is the record number; optional 32-bit integer variable set to the record number to be read if positive, number of records to backspace if negative; used only for type 1 and type 2 files; if omitted, record at current position is read.

num specifies the number of the record to be read, or the number of records to be backspaced before the read, if negative. Records are numbered sequentially; the first record is number 1. For further discussion of parameters and read capabilities, refer to READF.

EWRIT (Extended File Write)

The EWRIT routine is the extended range version of WRITF.

```
CALL EWRIT(idcb,ierr,ibuf,il,num)
```

where:

idcb	is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.
ierr	is for error return; 1-word variable in which negative error code is returned.
ibuf	is the user buffer; array containing the record to be written; should be large enough to contain the largest record to be written.
il	is the length in words; optional 1-word variable specifying number of words to be written; if omitted, one record is written to type 1 and 2 files, zero-length record to all other file types. Refer to section WRITF for details of *il* use.
num	is the record number; optional 32-bit integer variable containing record number to be written if positive, number of records to backspace if negative; used only for type 1 and 2 files; if omitted, record is written to current file position.

For further discussion of write capabilities and sequence of operations, refer to the WRITF routine section.

FCONT (Type 0 File Control)

This routine controls input/output functions on a peripheral device. The on a peripheral device. The device must have been opened with OPENF. The call has no effect on other file types. It performs the same functions as the EXEC I/O CONTROL call (EXEC 3), such as backspacing, rewinding, and writing end-of-file on cartridge tape, and controlling line spacing and top-of-form on a line printer.

```
CALL FCONT(idcb,ierr,icnwd,iprm1,iprm2,iprm3,iprm4)
```

where:

idcb	is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.
ierr	is for error return; 1-word variable in which negative error code is returned, or zero if successful.

See the EXEC 3 call in Chapter 3 of this manual for a detailed description of the other parameters. Appropriate values for several of these parameters depend on the device; these values may be found in the *RTE-A Driver Reference Manual*, part number 92077-90011.

On completion of the FCONT call, the A-Register contains an EXEC error code and the B-Register contains device status. From FORTRAN or Pascal, these values are retrieved by a call to the system library routine ABREG.

FSTAT (Retrieve System Cartridge List)

This routine returns an array containing the system cartridge list. For each mounted cartridge, its logical unit number, last FMP track, cartridge reference number, and if a program has locked the cartridge, the program's open flag are returned.

ierr = FSTAT(*istat*, *ilen*)

where:

ierr is the error return code; 1-word variable in which negative error code is returned.

istat is the returned cartridge list; array in which the system cartridge list is returned, using 4 words per cartridge; see table below. *istat* must be at least as large as *ilen*.

ilen is the length in words of *istat*; Optional 1-word variable; if omitted, 125 words will be used. If *istat* does not contain zero as a terminator, then the entire directory may not have been returned.

FSTAT returns either the full cartridge list or the number of words specified by *ilen*, whichever is smaller.

The format of *istat* is shown in Table C-1.

Table C-1. The *istat* Parameter Format (FSTAT Call)

Word	Contents	Cartridge
1	Logical unit number	First cartridge in directory
2	Last FMP track	
3	Cartridge reference number	
4	Open flag of locking program or 0 if not locked	
5	Logical unit number or 0 if no more disks	Remaining cartridges
.	.	.
.	.	.
.	.	.
125	0 (terminates list)	

IDCBS (Retrieve Number of DCB Words)

This function returns the number of words in a Data Control Block actually used by the File Management Package for data transfer and file control.

isize = IDCBS(*idcb*)

where:

isize is the returned DCB buffer size; actual size of data control block array in use.

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

When a Data Control Block 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 (refer to the Data Control Block section). This call returns the actual Data Control Block size: the DCB packing buffer used plus 16 control words.

INAMR (Rebuild Namr String)

INAMR is the inverse of the NAMR routine. It builds a character string in the NAMR format from an input array (parameter buffer). See the NAMR routine for more information.

CALL INAMR(*ipbuf*, *ubuf*, *maxb*, *istart*)

where:

ipbuf is the parameter buffer; 10-word array in which the namr parameters are stored; same format as the parameter buffer for the NAMR routine.

ubuf is the returned user buffer; output array that will contain the character string generated by INAMR.

maxb is the *ubuf* length in characters; 1-word variable.

istart is the starting character position; 1-word variable; the position within the user buffer to start the namr string.

The value of *istart* is updated at the end of the INAMR routine. An empty *ubuf* array should start with *istart* = 0. Because *istart* is modified by this routine, it should be passed as a variable, not as a constant.

LOCF (Retrieve Information on Open File)

A call to this routine retrieves status and location information on an open file. The information is obtained from the Data Control Block control words for the file. The minimum information returned is the next record number; all other information is optional.

```
CALL LOCF(idcb,ierr,irec,irb,ioff,jsec,jlu,jty,jrec)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

irec is the next record; 1-word variable in which number of next sequential record is returned. Will be in range 1 through 32767.

irb is the next block; optional 1-word variable in which next block number is returned; not returned for type 0 files; includes extents if file was extended. Will be in range 0 through 16383.

ioff is the next word; optional 1-word variable in which number of next word in block (DCB) is returned; not returned for type 0 files. *ioff* will be in range 0 through 127.

jsec is the actual File size; optional 1-word variable in which number of sectors in file at creation is returned; not returned for type 0 files; *jsec*/2 provides number of blocks. Will be in range 2 through 32767 and will be even.

jlu is the logical unit; optional 1-word variable in which logical unit to which file is allocated is returned.

jty is the file type; optional 1-word variable in which file type at open is returned.

jrec is the record size; optional 1-word variable in which record size of type 1 and 2 files or read/write code for type 0 files is returned; not applicable to files with variable length records (type 3 and above).

Location Information

Together, *irec*, *irb*, and *ioff* provide the current position within a disk file; they are not set for non-disk files. The values in these parameters may be passed directly to APOSN (see its section) 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.

irec numbers records starting with 1 for the first record, 2 for the second, and so forth. *irec* alone is sufficient to find the location of type 1 files.

irb numbers the blocks of the file relative to the start of file: 0 for the first block in the file, 1 for the second, and so forth. If the file is extendible (type 3 and above), *irb* includes extent information and is specified as:

(blocks in main file * extent #) + (block # in current extent)

The range of *irb* is 0 through 16383.

ioff numbers the words within a block, beginning with word zero. The range of *ioff* is 0 through 127.

Status Information

jsec (file size) is the actual size of a file or its extent; it is always an even number of sectors with two 64-word sectors for each 128-word block in a disk file. It is not applicable to non-disk files.

jlu is the logical unit to which a file, disk or non-disk is allocated.

jty is the file type of the file; if forced to type 1 at open, then 1 is returned.

jrec is the record size of the file; it is meaningful for type 2 files only; it is specified at creation. For type 1 files, whether actual or forced to type 1 at open, *jrec* is set to 128 on the first read or write access.

For type 0 files, *jrec* specifies the read/write access code:

bit 15 = 1 indicates read access
bit 0 = 1 indicates write access

NAMF (Rename a File)

This routine renames an existing file. If the file 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,name,nname,isecu,icr)
```

where:

idcb	is the Data Control Block; an array of 144 + (n * 128) words, where n is positive or zero.
ierr	is for error return; 1-word variable in which negative error code is returned.
name	is the file name; 3-word array containing ASCII file name.
nname	is the new file name; 3-word array containing ASCII file name to replace *name* as the file name.
isecu	is the security code; optional 1-word variable in range 0 through +/− 32767; omitted only if file *name* was created without a security code or with a zero code; if specified, the code must match.
icr	is the cartridge reference; optional 1-word variable in the range 0 through 32767; if zero or omitted, the first file found with given *name* will be renamed if the security code matches; if specified, only a file on the specified cartridge is renamed.

A file will not be renamed if:

- the file is an active program file.
- the file is open to another program.

NAMR (Parse an Array)

The NAMR routine parses an array (buffer) of any length and returns up to seven subparameters.

```
CALL NAMR(ipbuf,ubuf,maxb,istart)
```

where:

ipbuf	is the parameter buffer; 10-word array in which up to seven subparameters are returned from the input string.
ubuf	is the user buffer; input array to be parsed.
maxb	is the *ubuf* length in characters; 1-word variable.
istart	is the starting character position; 1-word variable; will be updated with each call to NAMR.

NAMR was originally designed for parsing FMGR file namrs, but it can be used to parse other arrays as well.

NAMR parses the character string between the starting character position (*istart*) and the next comma (,). Up to seven subparameters, delimited by colons (:), are derived from the character string. The first subparameter may be up to three words long; the remaining subparameters may be only one word long. Each subparameter may be type 0 (null), type 1 (integer numeric), or type 3 (ASCII); these correspond to the FMGR global parameter types. *istart* will then be updated to point at the character following the comma, so that a subsequent NAMR call will parse the next substring.

The ten-word output array (parameter buffer) has the following structure:

Words 1 through 3 constitute the first subparameter

Word 1:	If type = 0, word 1 has a value of 0. If type = 1, word 1 is a 16-bit twos-complement number. If type = 3, word 1 contains characters 1 and 2.
Word 2:	If type = 0 or 1, word 2 has a value of 0. If type = 3, word 2 contains characters 3 and 4 or trailing blanks.
Word 3:	If type = 0 or 1, word 3 has a value of 0. If type = 3, word 3 contains characters 5 and 6 or trailing blanks.

Word 4 gives the parameter types of all seven subparameters, in two-bit pairs. Values of the two-bit pairs are:

0 = null
1 = integer numeric
2 = (no meaning assigned)
3 = ASCII

Bits 0 and 1 give the type of the first subparameter.
Bits 2 and 3 give the type of the second subparameter.
Bits 4 and 5 give the type of the third subparameter.
Bits 6 and 7 give the type of the fourth subparameter.
Bits 8 and 9 give the type of the fifth subparameter.
Bits 10 and 11 give the type of the sixth subparameter.
Bits 12 and 13 give the type of the seventh subparameter.

Words 5 through 10 have the format of word 1. They are (items in parentheses are the corresponding file namr subparameters):

Word 5 is the 2nd subparameter (security code).
Word 6 is the 3rd subparameter (cartridge reference).
Word 7 is the 4th subparameter (file type).
Word 8 is the 5th subparameter (file size).
Word 9 is the 6th subparameter (record size).
Word 10 is the 7th subparameter.

On return, a negative value in the A-Register indicates something was parsed.

OPEN (Open a File)

OPEN opens a file for read or write access. The file must have been created prior to the OPEN call. It may be a disk or non-disk (type 0) file. If the specified Data Control Block is already associated with an open file, that file is closed and the specified file is opened.

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. Non-disk (Type 0) files may be opened with a function code specified at creation or a function specified in the OPEN call.

When a file is opened, it is positioned at the first record in the file.

```
CALL OPEN(idcb,ierr,name,ioptn,isecu,icr,idcbs)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned if unsuccessful, file type if successful.

name is the file name; 3-word array containing ASCII file name.

ioptn are the open options; optional 1-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 has access to 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.
- Type 0 files use function code defined at creation.

To open a file with other options, set *ioptn* as described below under OPEN Options.

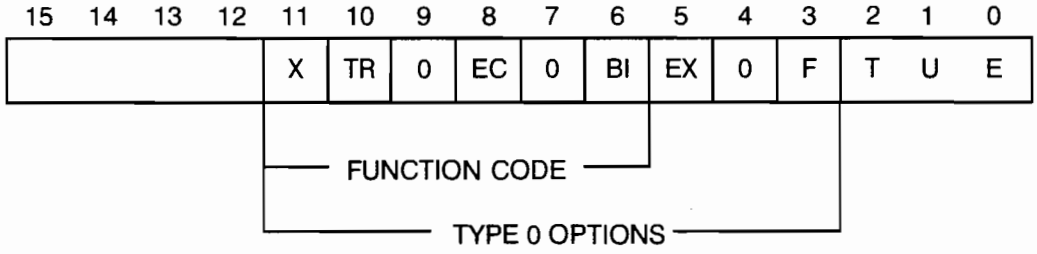
isecu is the security code; optional 1-word variable; must be specified to open a file that was created with a negative security code or to write on a file protected with a positive code; may be omitted if the file was not protected at creation.

icr is the cartridge reference; optional 1-word variable; if set, FMP searches only that cartridge for the file; if omitted it searches cartridges in the cartridge list order and opens the first file found with the specified name.

idcbs is the DCB buffer size; optional 1-word variable; number of words in the DCB buffer if larger than 128; if omitted, FMP assumes that DCB size (control words + buffer) is 144 words, regardless of *idcb* dimension. The DCB buffer is not used for access to type 0 and type 1 files.

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. If the file is a type 0 file, it is locked.
1 File may be shared by up to seven programs.

U (bit 1) = 0 File opened for standard (non-update) write.
1 File opened for update.

T (bit 2) = 0 Use file type defined at creation.
1 File type is forced to type 1.

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.
1 Use function code defined in bits 6-10 of *ioptn*.

EX (bit 5) = 1 Permits extents on type 1 and 2 files.
0 No extents on type 1 and 2 files.

Bits 6 through 10 correspond exactly to the function code used for the read or write EXEC calls (EXEC 1 or EXEC 2).

X (bit 11) is defined for the appropriate driver. Refer to the *RTE-A Driver Reference Manual*, part number 92077-90011, for details.

All other bits should be set to zero.

If bits are set in the *ioptn* parameter, their value must be expressed as an octal or decimal number. For example, if you want to open a file for non-exclusive use and force type 1 access to the file, you would set bits 0 and 2. This would result in a binary value of 101, which would have to be converted to an octal value (5B) or a decimal value (5) before it was passed as a parameter to OPEN.

Exclusive/Non-exclusive Open (E bit)

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, you must make the call 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. If the file opened is a type 0 file and the device is not interactive, the LU is locked.

If more than one program needs to access the file, it should be opened non-exclusively by setting the *ioptn* E bit. A file may be open to as many as seven programs at one time. A non-exclusive open will not be granted if the file is already opened exclusively.

An active program file cannot be opened exclusively.

Update Open (U bit)

In update mode (when the U bit is set) the block containing the record to be written is read into the DCB buffer before the record is written. This is sometimes useful when you are writing records into the middle of existing data, as we shall describe below. Update mode has no effect on reading records.

When you write a record, FMP puts it into the proper position in the DCB buffer, so that when the buffer is posted to the disk file the record is placed in the position that you specified in your WRITF call. That is, FMP is smart enough to make sure that the record you write ends up in the right place in the file; this is true whether the file was opened in update mode or in non-update mode. The difference between the two modes shows up in the records surrounding the ones that you write.

Figure C-1 shows how the record that you write from your user buffer goes into the DCB buffer and then into the disk file. The + marks represent the other records in the DCB buffer. When your new record gets posted to the file the entire buffer is posted, both the record you have just written and the other records that were in the block at the same time. In update mode, FMP fills the DCB buffer with the block that contains the record that you are about to over-write. If, for example, you were going to re-write record 6 in update mode, FMP would read into the buffer the block that contained, say, records 5, 6, 7, and 8. Then, after you re-wrote record 6, records 5, 7, and 8 would be posted to the disk file along with the new record 6.

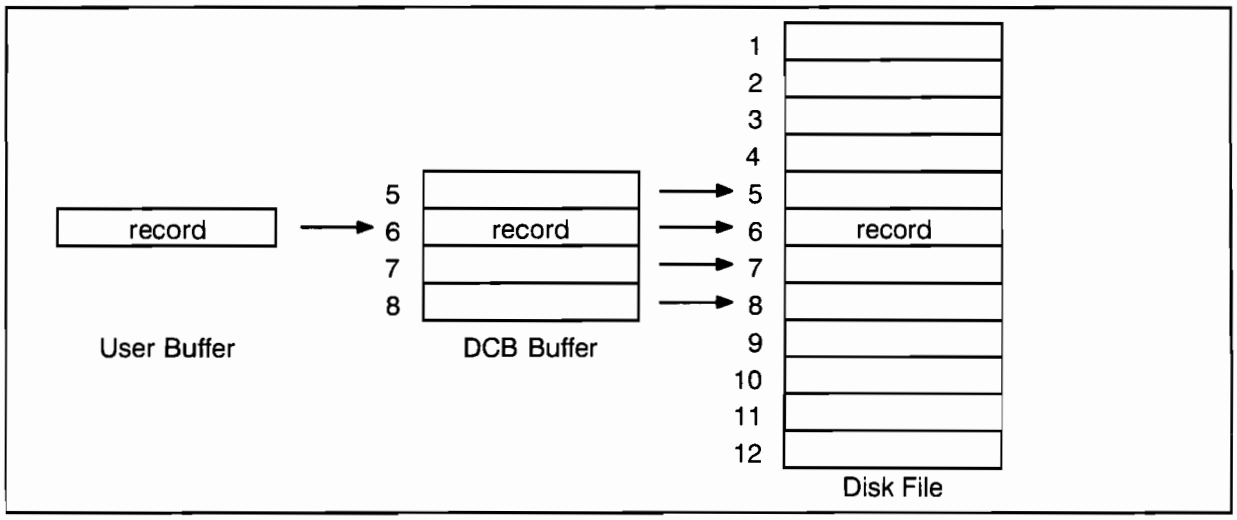


Figure C-1. Writing to a File

In non-update mode, FMP does not read anything into the DCB buffer before writing your record, so you might not get the results you were expecting. To continue the example, if the DCB buffer contained records 1, 2, 3, and 4 before you re-wrote record 6 in non-update mode, FMP would leave those records in place while it wrote the new record 6. The DCB buffer would then contain records 1, 6, 3, and 4, and these records would end up in the disk file when the buffer was posted.

The end result would be that the file contained records 1, 6, 3, and 4 where you expected to see records 5, 6, 7, and 8. Note that in non-update mode you still get the record written into the right position in the file; it's just that you might fill the rest of that block with garbage.

If you use update mode when you do not need it, you will spend extra time doing unneeded disk fetches. The recommendations below will help you to open files for efficient access.

Update mode should be used to write to type 2 files. A type 2 file should be opened in standard mode only when originally writing the file or adding new data at the end of the existing data and before the end of the file, and then only if the file is to be written sequentially.

Update mode is ignored for type 1 files. Although, like type 2, they 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.

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 you attempt to change it, a −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.

Access Function Override (F-bit)

Some devices require a specific set of options for access. You may override the access function defined at creation (or OPENF) by setting the *ioptn* F bit, and setting bits 6 through 11 to the desired access function. The function code, as used by EXEC 1 and 2, is described in Chapter 3.

Type 1 Access (T-bit)

Any disk file may be forced to type 1 access by setting the *ioptn* T bit. Type 1 access is faster because it bypasses the Data Control Block buffer and transfers data directly to the user buffer defined as *ibuf* in a READF or WRITF call. The file type defined at creation is not affected; the file is treated as type 1 only for the duration of this open. You are responsible for any packing or unpacking of records in files forced to type 1. That is, if the records are less than 128 words, you must determine the start and end of each record. Refer to READF and WRITF for particulars of type 1 access.

OPENF (Open a File or Device)

A call to OPENF opens a file or device for access; if you are opening a file, it must have been created prior to the OPENF call. The file opened may be a disk or non-disk (type 0) file. If a logical unit number is passed in the first word of the file namr, a DCB is created to allow type 0 access. In this case, no type 0 file is necessary and none is created by the call.

Generally, OPEN should be used when you are dealing exclusively with disk files, as it is smaller than OPENF. OPENF should be used when you are dealing with devices, as it allows you to open a device without first creating it as a type 0 file. If you might be dealing with either disk files or device, use OPENF.

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. Type 0 files may be opened with a function code specified at creation, or a function code specified in the OPENF call.

When a file is opened, it is positioned at the first record in the file.

```
CALL OPENF(idcb,ierr,name,ioptn,isecu,icr,idcbs)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned if unsuccessful, file type if successful.

name is the file name or logical unit number; either a 3-word array containing an ASCII file name or an 1-word variable containing a logical unit number.

ioptn are the open options; optional 1-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 has access to 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.
- Type 0 files use function code defined at creation.

To open a file with other options, set *ioptn* as described below under OPENF Options.

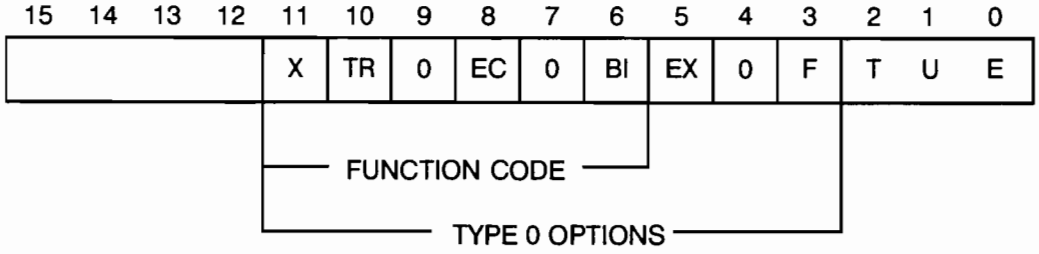
isecu is the security code; optional 1-word variable; must be specified to open a file that was created with a negative security code or to write on a file protected with a positive code; may be omitted if the file was not protected at creation.

icr is the cartridge reference; optional 1-word variable; if set, FMP searches that cartridge for the file; if omitted, it searches cartridges in the cartridge directory order and opens the first file found with the specified name.

idcbs is the DCB buffer size; optional 1-word variable; set to number of words in the DCB buffer if larger than 128; if omitted, FMP assumes that DCB size (control words + buffer) is 144 words, regardless of *idcb* dimension.

OPENF 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. If the file is type 0, then the device is locked.
1 File may be shared by up to seven programs

U (bit 1) = 0 File opened for standard (non-update) write
1 File opened for update

T (bit 2) = 0 Use file type defined at creation
1 File type is forced to type 1

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
1 Use function code defined in bits 6-10 of *ioptn*

EX (bit 5) = 1 Permits extents on type 1 and 2 files.
0 No extents on type 1 and 2 files.

Bits 6-10 correspond exactly to the function code used for the EXEC 1 or 2 call.

X (bit 11) is defined for the appropriate driver. Refer to the *RTE-A Driver Reference Manual*, part number 92077-90011, for details.

All other bits should be set to zero.

For further explanation of OPENF parameters, see the OPEN call.

Table C-2. OPENF Defaults

Device	Device Type	EOF Code	Spacing	Read/ Write	Comments
"Bit Bucket"	0	PA		WR	
Interactive Devices	00-07	PA		BO	If option parameter is 0, then echo bit is set.
Plotter Graphics Display	10-11	PA		BO	
Printer	12-13	PA		BO	
Card Reader	14	EO		BO	
Card Punch	15	EO		BO	
Mag Tape Cassette	20-23	EO	BO	BO	
Disks	30-37				Disks are accessible only via named files.
Various	40-77	EO	BO	BO	

EOF Code

EO = subfunction 100 (end-of-file mark)
PA = subfunction 1100 (page eject)

Spacing

FS forward space
BS backspace
BO both

Read/Write

RE read
WR write
BO both

POSNT (Position a File)

This routine positions a file relative to the current file position or to a specified record number. It can be used to position all file types.

```
CALL POSNT(idcb,ierr,nur,ir)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative FMP error code is returned.

nur is the number of records; 1-word variable specifying the number of records to position forward if positive, backward if negative; if *ir* is included as a non-zero value, *nur* specifies the record number to which the file is positioned. *nur* must be 32767 of less.

ir is an optional 1-word variable set to indicate that *nur* is interpreted as a record number; if *ir* is omitted or set to zero, *nur* is treated as the number of records to space forward or backward. Refer to Table C-3, below.

Table C-3. Relation Between Parameters *nur* and *ir* (POSNT Call)

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 Non-Disk Files (Type 0)

When the file is a non-disk 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 read. 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 other than as the first record backspaced over, an error (−12) is returned and the call terminates after forward spacing to position the file immediately after the EOF.

Positioning Random-Access Files (Types 1 and 2)

POSNT may be used to position these file types; however, file positioning for random access files can be specified in the read or write requests and POSNT may not be necessary.

Positioning Sequential-Access Files (Type 3 and Above)

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 disk or magnetic tape files.

POST (Post the DCB to a File)

This routine is called to post (write) the contents of the Data Control Block buffer to a disk 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 disk, assures that the next read is from disk, and can be used in a special case to save records in a file opened for non-exclusive use.

```
CALL POST(idcb,ierr)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for Error return; optional 1-word variable in which negative error code is returned; should be omitted only if A-Register is tested for errors.

This call is ignored for all files of type 0 or 1 since transfers to these files are always direct, bypassing the Data Control Block buffer.

PURGE (Remove a File)

A call to PURGE removes the named file from the system along with any extents associated with the file. When a file is purged, the file directory entry is no longer available. If the file was open, it is closed, freeing the Data Control Block. A file that is open to any program other than the calling program cannot be purged until it is closed.

```
CALL PURGE(idcb,ierr,name,isecu,icr)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

name is the file name; 3-word array containing ASCII file name.

isecu is the security code; optional 1-word variable; must be specified if file was created with a security code; otherwise, may be omitted.

icr is the cartridge reference; optional 1-word variable; if specified, FMP purges named file on specified cartridge. If omitted, FMP searches cartridges in cartridge directory order and purges the first file found with the specified name.

A file will not be purged if:

- the file is open to another program
- the file is an active program file (type 6)

Recovery of File Area Following PURGE

The area on disk 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 followed by other files, the cartridge must be packed in order to recover the file space. Packing is accomplished with the PK FMGR command (refer to the *RTE-A User's Manual*, part number 92077-90002, for a description of FMGR). If you create a new file that is exactly the same size as a purged file, the new file will replace the purged file.

READF (Read a File Record)

This routine transfers a record from an open file to the user buffer. Either one full record or a specified number of words is read. For sequential access files (type 0, 3 and above) the record read will be the record at which the file is currently positioned, or for random access files (type 1 and 2), it may be any specified record.

```
CALL READF(idcb,ierr,ibuf,il,len,num)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

ibuf is the user buffer; array into which the record is read; it should be large enough to contain the record; if *il* is specified, buffer should be of length *il*.

il is the length in words; optional 1-word variable specifying number of words to be read; should not be omitted for type 0 files; for other files, 1 record is read if *il* is omitted. Refer to Table C-4, below, for details of *il* use.

len is the number of words read; optional 1-word variable in which actual number of words read is returned; set to −1 if end-of-file is read; if omitted, information is not supplied.

num is the record number; optional 1-word variable set to record number to be read if positive, to number of records to backspace if negative; used only for type 1 and 2 files; if omitted, record at current position is read.

Relation of *il* to File Type

It is a good idea to specify *il* for non-disk file (type 0) and it doesn't hurt to specify it for other file types. If you do not know the length of a disk file record, *il* can be specified as the user buffer length to prevent reads beyond the user buffer. If the record is shorter than *il*, the exact record length is read for files with types 2, 3 and above. Table C-4 illustrates the effect of *il* depending on file type.

Figure C-2 illustrates a type 1 file read. The file is read directly into the user buffer when the number of words specified in *il* is greater than the 128 words expected for a type 1 file. Other file types may be forced to type 1 access at open in order to benefit from this type of transfer.

Table C-4. 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 record length defined for the file is less than *il*, one record is read.	Exactly *il* words are read; *il* may be more or less than a 128-word record.	Up to *il* words are read; if actual record length is less than *il*, one record is read.
il = 0 (not recommended)	Zero-length record is read; usually record is skipped and counted as read.	No action. (Zero-length record is 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.	128-word record is read.	Actual record length is read.
il < 0 (not recommended)	Up to -*il* characters are read.	No action.	Undefined.

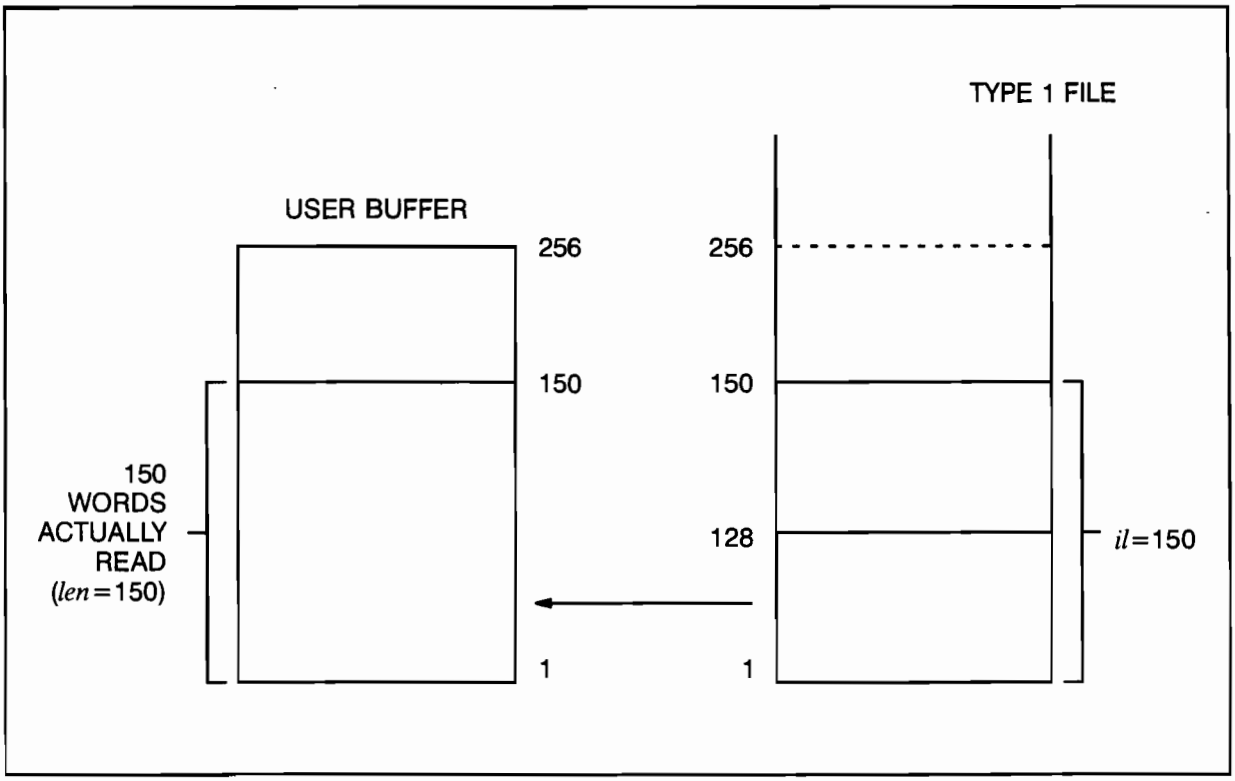


Figure C-2. Reading Type 1 Files with *il* Greater Than 128

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 disk record. This is because *len* is never set to a value greater than *il*.

To illustrate, suppose *il* is specified as 80 words. If 10 words are transferred, then *len* is set to 10. But whether 80 or more words are in the record, *len* is still set to 80, the value of *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 no 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 and 2:
ierr is set to −12 indicating an error. Access is not permitted beyond the end of file.

Type 3 and greater than 3:
len is set to −1 for the first EOF read; no error occurs but an attempt to read past the EOF causes an error (*ierr*= −12); you may not read past the end-of-file, but you may write beyond it. After EOF is read, the POSNT routine should be used to backspace one record before writing.

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*.

num is used only to position random access files (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 up to a maximum of 32767. If negative, *num* specifies the number of records to backspace from the current position in the file.

To illustrate, assume the type 1 or 2 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.

RWNDF (Rewind a File or Device)

This routine rewinds a non-disk file (type 0) or positions a disk file to the first record in the file.

```
CALL RWNDF(idcb,ierr)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned; may be omitted if A-Register is to be checked for errors.

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 positioned in reverse.

WRITF (Write a Record to a File or Device)

WRITF transfers a record from the user's buffer to an open file. For non-disk files (type 0) and sequential access files (type 3 and above), a specified number of words is written. Type 1 random access files are written in multiples of 128 words. Type 2 random access files are written in records whose length was specified at creation.

```
CALL WRITF(idcb,ierr,ibuf,il,num)
```

where:

idcb is the Data Control Block; an array of 144 + (*n* * 128) words, where *n* is positive or zero.

ierr is for error return; 1-word variable in which negative error code is returned.

ibuf is the user buffer; array containing the record to be written; should be large enough to contain the largest record to be written.

il is the length in words; optional 1-word variable specifying number of words to be written; if omitted, one record is written to type 1 and 2 files, zero-length record to other file types. Refer to Table C-5 for details of *il* use.

num is the record number; optional 1-word variable containing record number to be written if positive, number of records to backspace if negative; used only for type 1 and 2 files; if omitted or 0, record is written to current file position.

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 sequential access files (type 3 and above), it is essential to specify record length in *il*. Omitting *il* for these file types is the same as setting *il* to 0: a zero-length record is written. Refer to Table C-5 for other effects of *il*.

Computer Museum

Table C-5. Effect of *il* Parameter in WRITF

il Value	Type 0	Type 1	Type 2	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.

il can also be used to write an end-of-file on non-disk files (type 0) and sequential access files (type 3 and above). An attempt to write an end-of-file to a random access file (type 1 or 2) is ignored; no error is indicated.

When writing to a type 1 file, *il* is rounded up so that a multiple of 128 words is always transferred. When reading, the user buffer need be no larger than the length specified in *il*; only the requested number of word are transferred. Figure C-3 illustrates a write to a type 1 file with *il* = 150 words. In this case, 256 words (the shaded area) are actually transferred. Other file types may be forced to type 1 access at open in order to benefit from this type of transfer.

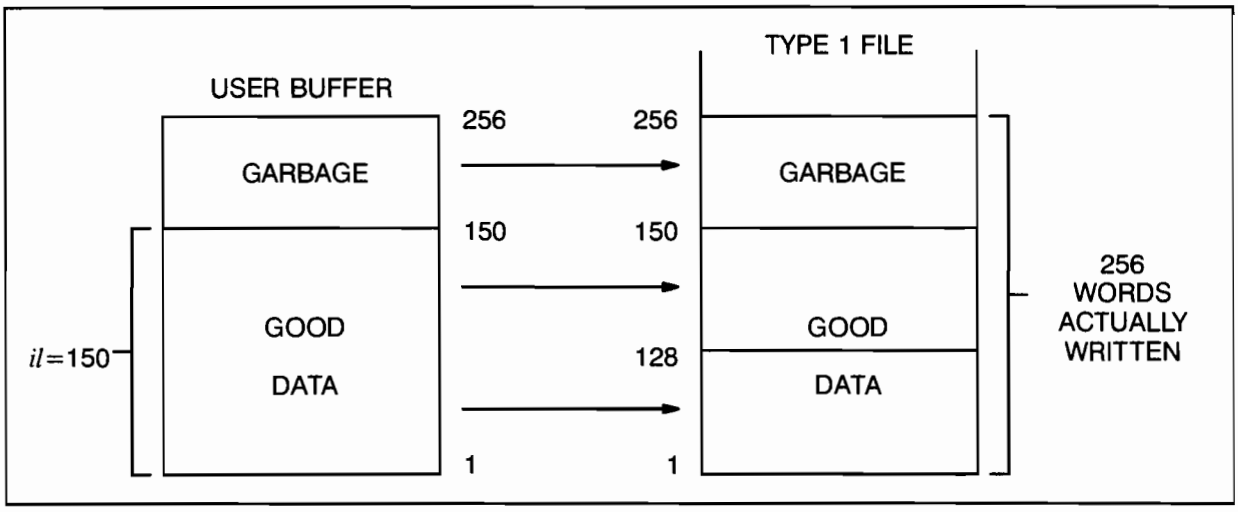


Figure C-3. Writing a Type 1 File with *il* Greater Than 128

Positioning with *num*

num is used only to position random-access file (types 1 and 2); if specified for other file types, it is ignored. If positive, *num* 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.

To illustrate, assume the file is positioned at the beginning of record 5.

1. If *num*=0 or is omitted, record 5 is written.
2. If *num*=6, record number 6 is written.
3. If *num*=−3, record number 2 is written.

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, the integrity of the file is destroyed.

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)
```

where:

idcb is the Data Control Block; 144-word array for use by XQPRG.

icode is the EXEC request code; 1-word variable used to schedule the program (9, 10, 23, or 24).

name is the program name; 3-word array; contains 5-character program name or 6-character program file name.

ifive are the scheduling parameters; optional 5-word array to be passed to program.

ibuf is the user buffer; optional array to be passed to program.

il is the user buffer length; optional 1-word variable; if positive, contains length in words; if negative, contains length in characters.

iprtn are the return parameters; optional 5-word array passed back from the program.

ierr is for error return; optional 1-word variable in which error code is returned; see below for details.

isecu is the security code; optional 1-word variable; must be specified to execute program in file created with a negative security code.

icr is 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.

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 (*iprtn*(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, 24, or 28).
11 = Not enough parameters

XQTIM (Time Schedule a Program)

The XQTIM routine allows you to schedule a program for timed execution. XQTIM checks to see whether the program has an ID segment, sets up an ID segment if necessary, and schedules the program for timed execution. If XQTIM sets up the ID segment, the ID segment is released when the program is removed from the time list.

```
CALL XQTIM(idcb,ierr,name,isecu,icr,ires,mult,itime)
```

where:

idcb	is the Data Control Block; 144-word array for use by XQTIM.
ierr	is for error return; 1-word variable in which the error code is returned.
name	is the program name; 3-word array; contains either a 5-character program name or 6-character program file name.
isecu	is the security code; optional 1-word variable; must be specified to execute a program contained in a file created with a negative security code; may be omitted if the file was not protected at creation time.
icr	is the cartridge reference; optional 1-word variable; if set, FMP searches that cartridge for the file; if omitted, it searches cartridge in the cartridge list order and executes the first file found with the specified name.
ires	is the resolution code; optional 1-word variable; if omitted, zero is used.
mult	is the resolution multiple; optional 1-word variable; program will be scheduled every *ires* * *mult* time intervals; if omitted, zero is used.
itime	are the time parameters; optional 4-word array; specifies either absolute or relative starting time for program execution.

Resolution code *ires*

This parameter specifies the time interval resolution to be used to schedule the program. The possible values are:

4 hours
3 minutes
2 seconds
1 tens of milliseconds
0 remove from time list

Time parameter array *itime*

This parameter specifies either the absolute time (in hours, minutes and seconds) or the relative time (in *ires* units) from now (the XQTIM call) the program is to start executing. The format is:

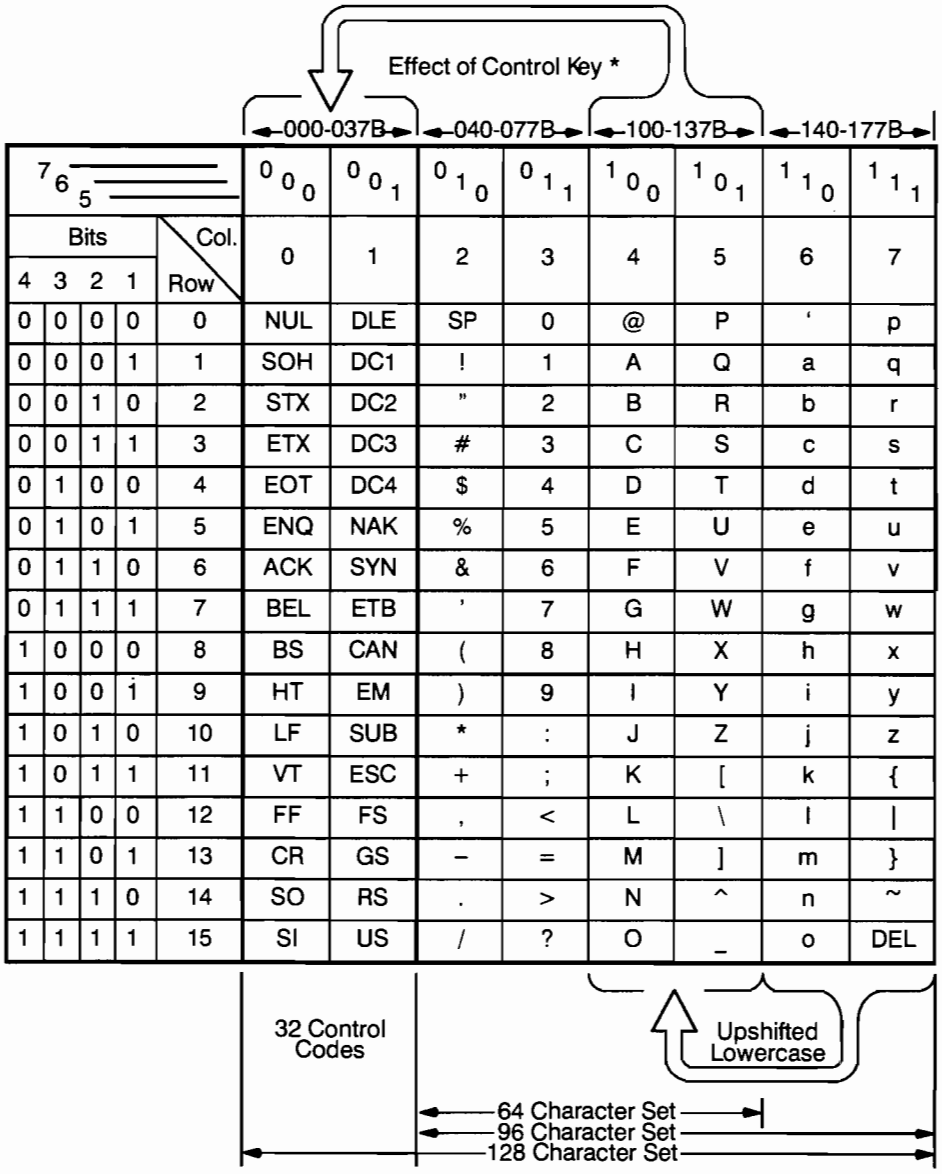
Absolute starting time:

Word 1 = hours
Word 2 = minutes
Word 3 = seconds
Word 4 = tens of milliseconds

Relative starting time:

Word 1 = −offset; must be negative; program begins execution *res*offset* from now.
Words 2 through 4 are ignored.

HP Character Set



7 6 5 / Bits 4 3 2 1 / Row	000	001	010	011	100	101	110	111
Col.	0	1	2	3	4	5	6	7
0 0 0 0 0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1 1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0 2	STX	DC2	"	2	B	R	b	r
0 0 1 1 3	ETX	DC3	#	3	C	S	c	s
0 1 0 0 4	EOT	DC4	$	4	D	T	d	t
0 1 0 1 5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0 6	ACK	SYN	&	6	F	V	f	v
0 1 1 1 7	BEL	ETB	'	7	G	W	g	w
1 0 0 0 8	BS	CAN	(	8	H	X	h	x
1 0 0 1 9	HT	EM	)	9	I	Y	i	y
1 0 1 0 10	LF	SUB	*	:	J	Z	j	z
1 0 1 1 11	VT	ESC	+	;	K	[	k	{
1 1 0 0 12	FF	FS	,	<	L	\	l	|
1 1 0 1 13	CR	GS	-	=	M	]	m	}
1 1 1 0 14	SO	RS	.	>	N	^	n	~
1 1 1 1 15	SI	US	/	?	O	_	o	DEL

Example: The representation for the character "K" (column 4, row 11) is

Bit	7	6	5	4	3	2	1
Binary	1	0	0	1	0	1	1
Octal	1		1			3	

Note: * Depressing the Control Key while typing an uppercase letter produces the corresponding control code on most terminals. For example, Control-H is a backspace.

Table D-1. Hewlett-Packard Character Set for Computer Systems

This table shows Hewlett-Packard's implementation of ANS X3.4-1968 (USASCII) and ANS X3.32-1973. Some devices may substitute alternate characters from those shown in this chart (for example, Line Drawing Set or Scandinavian font). Consult the manual for your device.

The left and right byte columns show the octal patterns in a 16-bit word when the character occupies bits 8 to 14 (left byte) or 0 to 6 (right byte) and the rest of the bits are zero. To find the pattern of two characters in the same word, add the two values. For example, "AB" produces the octal pattern 040502. (The parity bits are zero in this chart.)

The octal values 0 through 37 and 177 are control codes. The octal values 40 through 176 are character codes.

Decimal Value	Octal Values		Mnemonic	Graphic[1]	Meaning
	Left Byte	Right Byte			
0	000000	000000	NUL	N_U	Null
1	000400	000001	SOH	S_H	Start of Heading
2	001000	000002	STX	S_X	Start of Text
3	001400	000003	EXT	E_X	End of Text
4	002000	000004	EOT	E_T	End of Transmission
5	002400	000005	ENQ	E_Q	Enquiry
6	003000	000006	ACK	A_K	Acknowledge
7	003400	000007	BEL	🔔	Bell, Attention Signal
8	004000	000010	BS	B_S	Backspace
9	004400	000011	HT	H_T	Horizontal Tabulation
10	005000	000012	LF	L_F	Line Feed
11	005400	000013	VT	V_T	Vertical Tabulation
12	006000	000014	FF	F_F	Form Feed
13	006400	000015	CR	C_R	Carriage Return
14	007000	000016	SO	S_O	Shift Out } Alternate
15	007400	000017	SI	S_I	Shift In } Character Set
16	010000	000020	DLE	D_L	Data Link Escape
17	010400	000021	DC1	D_1	Device Control 1 (X-ON)
18	011000	000022	DC2	D_2	Device Control 2 (TAPE)
19	011400	000023	DC3	D_3	Device Control 3 (X-OFF)
20	012000	000024	DC4	D_4	Device Control 4 ($\overline{\text{TAPE}}$)
21	012400	000025	NAK	N_K	Negative Acknowledge
22	013000	000026	SYN	S_Y	Synchronous Idle
23	013400	000027	ETB	E_B	End of Transmission Block
24	014000	000030	CAN	C_N	Cancel
25	014400	000031	EM	E_M	End of Medium
26	015000	000032	SUB	S_B	Substitute
27	015400	000033	ESC	E_C	Escape[2]
28	016000	000034	FS	F_S	File Separator
29	016400	000035	GS	G_S	Group Separator
30	017000	000036	RS	R_S	Record Separator
31	017400	000037	US	U_S	Unit Separator
127	077400	000177	DEL	■	Delete. Rubout[3]

Table D-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
32	020000	000040		Space, Blank
33	020400	000041	!	Exclamation Point
34	021000	000042	"	Quotation Mark
35	021400	000043	#	Number Sign, Pound Sign
36	022000	000044	$	Dollar Sign
37	022400	000045	%	Percent
38	023000	000046	&	Ampersand, And Sign
39	023400	000047	'	Apostrophe, Acute Accent
40	024000	000050	(	Left (opening) Parenthesis
41	024400	000051	)	Right (closing) Parenthesis
42	025000	000052	*	Asterisk, Star
43	025400	000053	+	Plus
44	026000	000054	,	Comma, Cedilla
45	026400	000055	-	Hyphen, Minus, Dash
46	027000	000056	.	Period, Decimal Point
47	027400	000057	/	Slash, Slant
48	030000	000060	0	Digits, Numbers
49	030400	000061	1	
50	031000	000062	2	
51	031400	000063	3	
52	032000	000064	4	
53	032400	000065	5	
54	033000	000066	6	
55	033400	000067	7	
56	034000	000070	8	
57	034400	000071	9	
58	035000	000072	:	Colon
59	035400	000073	;	Semicolon
60	036000	000074	<	Less Than
61	036400	000075	=	Equals
62	037000	000076	>	Greater Than
63	037400	000077	?	Question Mark

Table D-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
64	040000	000100	@	Commercial At
65	040400	000101	A	
66	041000	000102	B	
67	041400	000103	C	
68	042000	000104	D	
69	042400	000105	E	
70	043000	000106	F	
71	043400	000107	G	
72	044000	000110	H	
73	044400	000111	I	
74	045000	000112	J	
75	045400	000113	K	
76	046000	000114	L	
77	046400	000115	M	
78	047000	000116	N	Uppercase Letters
79	047400	000117	O	
80	050000	000120	P	
81	050400	000121	Q	
82	051000	000122	R	
83	051400	000123	S	
84	052000	000124	T	
85	052400	000125	U	
86	053000	000126	V	
87	053400	000127	W	
88	054000	000130	X	
89	054400	000131	Y	
90	055000	000132	Z	
91	055400	000133	[	Left (opening) Bracket
92	056000	000134	\	Backslash. Reverse Slant
93	056400	000135	]	Right (closing) Bracket
94	057000	000136	^ ↑	Caret. Circumflex: Up Arrow[4]
95	057400	000137	_ ←	Underline: Back Arrow[4]

Table D-1. Hewlett-Packard Character Set for Computer Systems (continued)

Decimal Value	Octal Values		Character	Meaning
	Left Byte	Right Byte		
96	060000	000140	`	Grave Accent[5]
97	060400	000141	a	
98	061000	000142	b	
99	061400	000143	c	
100	062000	000144	d	
101	062400	000145	e	
102	063000	000146	f	
103	063400	000147	g	
104	064000	000150	h	
105	064400	000151	i	
106	065000	000152	j	
107	065400	000153	k	
108	066000	000154	l	
109	066400	000155	m	
110	067000	000156	n	Lowercase Letters[5]
111	067400	000157	o	
112	070000	000160	p	
113	070400	000161	q	
114	071000	000162	r	
115	071400	000163	s	
116	072000	000164	t	
117	072400	000165	u	
118	073000	000166	v	
119	073400	000167	w	
120	074000	000170	x	
121	074400	000171	y	
122	075000	000172	z	
123	075400	000173	{	Left (opening) Brace[5]
124	076000	000174	|	Vertical Line[5]
125	076400	000175	}	Right (closing) Brace[5]
126	077000	000176	~	Tilde, Overline[5]

Note 1: This is the standard display representation. The software and hardware in your system determine if the control code is displayed, executed, or ignored. Some devices display all control codes as "@" or space.
Note 2: Escape is the first character of a special control sequence. For example, ESC followed by "J" clears the display on an HP 2640 terminal.
Note 3: Delete may be displayed as "_", "@", or space.
Note 4: Normally, the caret and underline are displayed. Some devices substitute the up arrow and the back arrow.
Note 5: Some devices upshift lowercase letters and symbols (' through ~) to the corresponding uppercase character (@ through ^). For example, the left brace would be converted to a left bracket.

Table D-2. HP 7970B BCD-ASCII Conversion

Symbol	BCD (Octal Code)	ASCII Equivalent (Octal Code)	Symbol	BCD (Octal Code)	ASCII Equivalent (Octal Code)
(space)	20	040	@	14	100
!	52	041	A	61	101
"	37	042	B	62	102
#	13	043	C	63	103
$	53	044	D	64	104
%	57	045	E	65	105
&	†	046	F	66	106
'	35	047	G	67	107
(	34	050	H	70	110
)	74	051	I	71	111
*	54	052	J	41	112
+	60	053	K	42	113
,	33	054	L	43	114
–	40	055	M	44	115
.	73	056	N	45	116
/	21	057	O	46	117
0	12	060	P	47	120
1	01	061	Q	50	121
2	02	062	R	51	122
3	03	063	S	22	123
4	04	064	T	23	124
5	05	065	U	24	125
6	06	066	V	25	126
7	07	067	W	26	127
8	10	070	X	27	130
9	11	071	Y	30	131
:	15	072	Z	31	132
;	56	073	[	75	133
<	76	074	\	36	134
=	17	075	]	55	135
>	16	076	↑	77	136
?	72	077	←	32	137

† The ASCII code 046 is converted to the BCD code for a space (20) when writing data onto a 7-track tape.

E

Program Types for RTE-A

This appendix lists the program type codes defined for RTE-A. These codes appear in the "program type" field of XNAM relocatable records. Thus, each module in a relocatable file contains a program type code. Despite the name "program type", these codes appear in subroutine modules and all other relocatable modules. Most RTE-A compilers allow the program type code associated with a routine to be specified in the source code of the routine; refer to the appropriate compiler reference manual for more information.

Table E-1. RTE-A Program Types

Type	Description
0	RTE operating system routine.
2,3,4,6	Program main. The four type values are for compatibility with RTE-6/VM; no special processing based on these values is performed on RTE-A.
7	Subroutine.
8	Microcode definition module.
30	Module to be relocated into system common during system generation.
Note: All other types have no processing defined.	

F

Cleaning Up Open Files

If a program opens a file and terminates (or is aborted) without closing the file, the file is left open. The file system (specifically D.RTR) attempts to clean up open files automatically. How this is done depends on whether the file is a CI or FMGR file, and whether or not it is a temporary file.

Definition of Temporary Files

Temporary CI files and temporary FMGR files are implemented differently. A temporary file is defined as a file to be used for only a limited amount of time and then purged. Normally, a program closes and purges the file itself, but if the program terminates before it can do that, the system purges it automatically. A VMA backing store file is a good example of a temporary file.

A temporary file under FMGR is defined simply as a file whose name starts with a digit (0−9). Such files can only be created using the FMGR file system routine CRETS or the CI routine FmpOpenTemp.

A temporary file under CI is defined as a file that was created with the T option in the FmpOpen call and has not been closed since creation.

The major difference between the two is this: the FMGR temporary file is considered temporary even after it is closed, the CI temporary file is no longer considered temporary if it is closed. This means that the FMGR temporary file may be purged by D.RTR anytime the creating program is no longer using the file, whether the program closed the file or terminated leaving it open. D.RTR will only purge the CI temporary file if the file is left open by its creating program and the creating program is no longer running, if the creating program closes the file, the file is no longer considered a temporary file. This means that if another program opens the same file (with or without the T option in FmpOpen), and aborts without closing the file, the file is not be purged automatically because it lost its temporary status when it was closed by the creating program.

Two notes about CI temporary files:

1. The file is created with the T option which sets the T flag in the directory entry. Masking has a T qualifier, which makes it possible to purge all such files using the CI PU command like this: PU,@.@.T

2. If a file that has the T flag set is re-opened by a program that does not have the T option in the FmpOpen call, the T flag is removed from the directory entry. Conversely, if a file without the T flag set is re-opened by a program using the T option, the T flag is set in the directory. The rule is that the T flag in the directory is set or cleared according to the FmpOpen option string used by the last program to open the file.

How Clean-Up Is Done

The following is a description of how clean−up is done for files that are left open. The four possible cases are:

1. normal CI files
2. temporary CI files
3. normal FMGR files
4. temporary FMGR files

CI Files

Open flags for CI files are maintained in free space in D.RTR's memory. There is no limit to the number of open flags per file (except for the physical limit of D.RTR's memory). In the open flag is a pointer to the file, a pointer to the ID segment of the program that opened the file, and a set status bits which include a bit indicating if the T option was used in the FmpOpen call and a bit indicating if the file was created with the FmpOpen call.

Also associated with open files is the FS bit in a program's ID segment. This bit is maintained jointly by the system and D.RTR. When a program makes it first call to D.RTR, the FS bit is set. When the program terminates, the bit is cleared.

When a program makes an FmpOpen call, D.RTR sets up an open flag for that file in memory and, if this is the first D.RTR call the program has made, D.RTR sets the FS bit in the program's ID segment. When the program makes an FmpClose call, the open flag is removed from memory, closing the file. If the program terminates without closing the file, then the open flag points to an ID segment that has the FS bit cleared as a result of the program's termination. This open flag is now considered invalid.

Whenever any program on the system makes its first call to D.RTR, two things happen:

1. D.RTR
   scans all open flags in its memory to see if any are invalid. If an invalid open flag is found, the flag is removed, thus closing the file.
2. D.RTR
   sets the FS bit in the program's ID segment to indicate that this program has made a D.RTR call.

Note that this scan is done every time any program makes its *first* call to D.RTR, that is, its FS bit is clear. This means that invalid open flags are cleared sometime after they become invalid, but the timing depends on FMP activity on the system.

CI Temporary Files

CI temporary files are closed in the same way as normal files, with the following addition: if, when D.RTR finds an invalid open flag and determines that this open flag came from the FmpOpen that *created* the file, *and* that the T option was used in that FmpOpen call, then the file is purged.

Note the restriction that the file is purged only if the invalid open flag that is found is from the FmpOpen call that created the file. If FmpOpen is called just to open an already existing file, the file is not purged automatically, even if the T option is used.

FMGR Files

Open flags for FMGR files are maintained in the file's directory entry on disk. There is room for one to seven open flags per file. Included in the open flag is a pointer to the ID segment of the program that opened the file and a value called a sequence counter. This sequence counter is a number from 0 to 31 and is taken from the ID segment of the opening program. The sequence counter in the ID segment is managed by the operating system and is incremented whenever a program using the ID segment terminates or is aborted.

When a FMGR file is opened, the open flag is created using the current sequence counter value from the calling program's ID segment, and the flag is placed into the file's directory entry on disc. When the program makes an FmpClose (or FMGR CLOSE) call, the flag word is removed from the directory entry, thus closing the file.

If the program terminates without closing the file, the open flag remains in the directory entry on disc. At this point, however, the sequence counter in the ID segment has been incremented because the program terminated and it no longer matches the sequence counter in the open flag. The open flag is now considered invalid.

D.RTR closes an open flag whenever it finds one while it is scanning the directory. D.RTR scans directories for various reasons, such as opening, creating, and purging files, locking, mounting, and dismounting cartgridges, and so on. When D.RTR finds an open flag, it first checks to see if the program in the associated ID segment is dormant. If it is, it removes the open flag. If the program is not dormant, D.RTR compares the sequence counter in the open flag with the one in the ID segment. If they don't match, it removes the open flag.

To clean up an open flag left behind by a program, D.RTR can be forced to scan the directory in several different ways. A simple file opening action on the cartridge (such as listing a file) causes D.RTR to scan the directory. However, it scans only until it finds the file to open.

If the invalid flag is further down the directory, D.RTR won't find and clear it.

The following are some of the actions that cause a *complete* directory scan by D.RTR:

- Create/Rename file – scans for a duplicate file name.
- Purge file – scans the directory looking for extents.
- Pack/Lock/Dismount cartridge – scans for any open or RP'd files.
- FMGR DL command – this forces a scan because the FMGR DL command requests a cartridge lock followed immediately by an unlock, this is done with the explicit purpose of forcing D.RTR to clean up invalid open flags.

The FMGR DL command is the most common way of forcing D.RTR to clean up invalid open flags. Note that the CI DL command does not do this because it does not do the cartridge lock/unlock sequence.

Also, because the sequence counter has only 16 potential values, it is possible (though unlikely) that programs will have run in an ID segment and terminated 16 times before the open flag is checked. This would cause the sequence counter to roll over to the original value, and the open flag would look valid. This open flag cannot be cleared until the program residing in the ID segment terminates, thus incrementing the sequence counter and making the open flag invalid.

FMGR Temporary Files

FMGR temporary files are closed in the same manner as normal FMGR files except that once the file is closed, it is a candidate for automatic purging. The file is not purged right away. Instead, the sequence is like this:

Assume D.RTR is scanning a directory for some operation, for example, a file rename. In the process of scanning, it finds a temporary file with an invalid open flag. The open flag is cleared as a normal invalid open flag and D.RTR makes a note of where the temporary file entry is. After the file rename is completed, just before D.RTR returns to the user, it returns and purges the temporary file.

The one exception to this pattern is during a file creation: if D.RTR finds a temporary file during the scan for a duplicate name, D.RTR purges it before the file creation is actually done to insure the most space possible is available for the new file.

The important point is that D.RTR remembers only one temporary file per directory scan. That is, if D.RTR comes across a second temporary file later in the scan, it will ignore the earlier file it found and remember the new one. The result is that D.RTR will purge only one temporary file at a time per directory scan.

Index

F

G

H

I

P

Q

R

S

T

U

V

W

X

Z