HEWLETT PACKARD

RTE-6/VM

CI User’s Manual

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

Manual Part No. 92084-90036
E0693

Printed in U.S.A. June 1993
Fifth Edition

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.

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.)

First Edition	Dec 1983	
Second Edition	Jan 1985	
Update 1	Jan 1986	
Reprint	Jan 1986	Update 1 incorporated
Third Edition	Aug 1987	Software 5000 (Rev. 5.0)
Fourth Edition	Jan 1989	Software 5010 (Rev. 5.1)
Update 1	Jul 1990	Software 5020 (Rev. 5.2)
Fifth Edition	Jun 1993	Software 6000 (Rev. 6.0)

Preface

This manual tells you how to use the RTE-6/VM Command Interpreter (CI) and the CI file system. It also describes the FMP subroutines and certain utilities used in the CI file system environment. Descriptions of other utilities and subroutines that can be used in the CI environment are given in manuals listed below.

EDIT/1000 User's Manual	92074-90001
SYMBOLIC DEBUG/1000 User's Manual	92860-90001
Macro/1000 Reference Manual	92059-90001
RTE-6/VM LINK User's Manual	92084-90038
RTE-A • RTE-6/VM Relocatable Libraries Manual	92077-90037
RTE-6/VM Utility Programs Reference Manual (TF)	92084-90007

This manual is for first-time users of RTE-6/VM and the CI file system. If you are unfamiliar with RTE and CI, we recommend that you read the material in the order in which it is presented. If you are familiar with the FMGR file system and with the RTE-6/VM Operating System, skip to the chapter of interest.

How This Manual Is Organized

This manual is divided into four sections: An overview of the CI file system environment, information for the display terminal users, FMP calls for programmers using the CI files, and the CI file system utilities.

Chapter 1	Introduces the CI file system and CI.
Chapter 2	Describes how to use the CI system commands.
Chapter 3	Describes how to use the CI file manipulation commands.
Chapter 4	Describes how to use the CI program control commands.
Chapter 5	Contains the CI command descriptions.
Chapter 6	Describes the CI FMP subroutines.
Chapter 7	Discusses exception condition handling.
Appendix A	Provides error codes and error messages.
Appendix B	Describes the FMP call conversion from the FMGR to the CI file system.

Conventions Used in This Manual

The command syntax and other conventions used in this manual are described in the following paragraphs. Sample terminal displays include both user inputs and program prompts and messages. Comments are given in parentheses. For example,

```
CI> dl /derick/casey/@.@        (List all files in subdirectory CASEY under directory
                                  DERICK)
```

The command syntax conventions are as follows:

Convention	Meaning
Italicized versus Uppercase characters	Italicized characters in command syntax indicate variables. Capital letters indicate the exact characters required. (CI does, however, accept lowercase input.) For example, the command syntax for the AS command is: AS *prog partition_number* and the actual sample entry can be: CI> as testprogram 2
[ ]	Optional parameters are shown in square brackets. If you omit a parameter, use a comma as a placeholder before specifying additional parameters.
|	Alternate choices for a parameter are separated by a vertical bar.
, or blank	Use a comma or a space as a delimiter between commands and parameters. Blank spaces are used throughout this manual in all syntax strings.

There are certain terms used in all syntax strings that have standard meanings throughout this manual. The most common terms are described below.

prog Program name; up to five characters can be used. Examples of program names:

```
A
PROGA
ADVEN
TIMER
```

lu Logical unit number in the range of 0 to 255. It refers to a physical input/output (I/O) device. LU 1 is usually the user terminal. Exceptions are noted throughout this manual, for example, in the TO command description in Chapter 5. LU 0 is the bit bucket, a non-existent device to which unwanted data can be dumped.

file File descriptor, which includes parameters that describe various file properties such as search path, file type, size, and record length. It can be accepted in any of the following formats:

Standard: */dir/subdir/filename.typex.qual:::type:size:rlen*

Combined: *subdir/filename.typex.qual::dir:type:size:rlen*

FMGR: *filename:sc:crn:type:size:rlen*

filename A file descriptor parameter. In the CI file system, it can have up to 16 characters followed by the file type extension (typex) and the qualifier parameter (qual) with a period as the delimiter.

mask Mask field. May be the wildcard characters in the filename parameter (- and @) or a mask qualifier appended to the filename parameter. Refer to Chapter 3 for details.

file|lu Either a file descriptor or a logical unit number may be specified. A mask may be used in the file descriptor.

prog|file Either a program name or a file descriptor may be specified. Refer to the RU command description in Chapter 5 for more information.

pram One parameter is allowed.

pram*2
:
pram*n Two to *n* parameters are allowed. Unspecified parameters can default to zero or zero-length strings, depending upon the application.

Table of Contents

Chapter 1
CI File System Introduction

Chapter 2
System Commands

Chapter 3
Manipulating Files

Chapter 4
Controlling Programs

Chapter 5
CI Command Descriptions

Chapter 6
FMP Routines

Chapter 7
Exception Condition Handling

Appendix A
Error Messages and Codes

Appendix B
Converting FMGR File Calls

List of Illustrations

Tables

1

CI File System Introduction

This manual is the primary reference source for display terminal users and programmers in a file system environment managed by the Command Interpreter (CI) program. The CI file system allows efficient and more logical use of disk space and facilitates organization of files with a hierarchical directory structure.

This chapter provides a brief comparison between the CI and FMGR file systems. It describes the features of the CI program and introduces the basic characteristics of CI files.

RTE-6/VM File Systems

Normal interface to the operating system is through one of two programs, FMGR or CI. Each of these programs provides two basic capabilities:

- Interface to the operating system. This includes obtaining system status, modifying system parameters, running other programs, and controlling input/output devices.
- Interface to the file system. This includes creating, purging, and transferring files and obtaining status information.

In system interfacing, the FMGR and CI programs are similar, but with a few differences in command syntax or function. For example, the FMGR system break mode commands can be entered in CI without the SY prefix. File system interface is different between the two programs. An overview of the CI file system is given in this chapter. Refer to the *RTE-6/VM Terminal User's Reference Manual,* part number 92084-90004, for a full description of the FMGR file system.

The FMGR file system divides a disk into fixed-size cartridges that are identified with either negative LU numbers or positive cartridge reference numbers (CRN). The CRN can also be a two-character string. Each cartridge has a cartridge directory containing pertinent information on all files stored on that cartridge.

The CI file system divides the disk into large areas of free blocks. These areas are identified by LU numbers and are called disk volumes. Files in each disk volume are managed by directories and subdirectories, which maintain information on what files exist and where they are located on the disk. One directory in each disk volume contains the names of all unique directories in the disk volume. This directory is called the root directory, and the directories included in the root directory are called global directories. Directories that are included in other directories are called subdirectories.

Comparison of the characteristics of the two file systems is given in Table 1-1.

During user accounts installation, either FMGR or CI can be specified as the primary program to be used for accessing all relevant system functions. The primary program is defined as the program scheduled at log on. Each program can also be selected to run automatically at log on. The primary program setup procedure is described in the *RTE-6/VM System Manager's Reference Manual,* part number 92084-90009.

Table 1-1. RTE-6/VM File Systems Comparison

	FMGR File System	**CI File System**
File name	1-6 characters	1-16 characters
Cartridge/ Directory	1-2 characters or numeric cartridge names	1-16 characters in directory names
File Security	Security code used for file protection	Protection based on directory (and file) ownership
File Types	Defines the structure of the files	File type extensions describe the contents of the files
File Mask	None	Mask qualifier and special characters in file name
File Size	Extendable (except type 6)	Extendable (except type 6)
Time Stamps	None	Create, access, update times handled by the file system
Subdirectories	None	Subdirectories within directories and other subdirectories
File Recovery	None	Operator recoverable immediately after purge
Spooling	Can be done interactively and programmatically	Through FMGR
Incremental Backup	None	Done in conjunction with FST or TF utility

Command Interpreter Features

The Command Interpreter (CI) may be scheduled at logon or from FMGR or other user programs in the session environment. When CI is executed, the program prompt is displayed and CI is ready to accept a command. The commands and the required parameters can be entered in the command string in either uppercase or lowercase letters. Blank characters and commas can be used as the command string delimiters. Throughout this manual, the blank is used as the command string delimiter. However, if there are parameters omitted from the middle of a string, commas must be used as placeholders. Following are examples of CI command entries.

`CI> edit`	(Run the editor program to create a file)
`CI> wh`	(Display system status)
`CI> co report.txt data`	(Copy REPORT.TXT into new file DATA)
`CI> dl /jones/`	(Display all files in directory JONES)

Interactive operations available through CI include the following functions:

System Status Check
System Control
File Manipulation
Program Control

The Command Interpreter provides commands that start and stop programs and commands that change the way a program executes. Commands are available for the creation of directories and subdirectories that are used in file management. Files can be created, copied, stored in a directory or subdirectory, purged (destroyed), and renamed. You can control access to files and manipulate a group of files with a single command using the file mask feature. Time stamps are maintained for all files to keep track of date of creation, access, and updates. System information can be displayed on the terminal screen; for example, program status and input/output device status. System behavior can be controlled through the system commands. These features are described in Chapters 2 through 5.

Spooling is not available in CI. If you must use the spooling system, store your files on an FMGR disk cartridge (following the FMGR name convention) and then run FMGR to use the spooling system. For detailed information about spooling, refer to the *RTE-6/VM Batch and Spooling Reference Manual,* part number 92084-90006.

CI provides another level of commands for the System Manager. In addition to using all the interactive capabilities described in Chapters 2 through 5, the System Manager is allowed to do the following:

- Set up the session to run in the CI environment.
- Change the properties of any program. Applies to all program control commands.
- Override file protection. Applies to all file manipulating commands.
- Modify system attributes such as the system clock. Applies to all system control commands.
- Re-initialize disks.

Certain commands are subject to RTE-6/VM user capability level restrictions. If you do not have the proper capability level, entering any of these commands results in a capability error. To change your capability level, see your System Manager. For more information on command capability levels, refer to the *RTE-6/VM System Manager's Reference Manual* and the *RTE-6/VM Terminal User's Reference Manual.*

CI also provides special features such as command stack, command files, program execution control, multiple commands entry, and quoting (string passing).

The command stack lets you repeat commands without retyping or modify commands before repeating them. A command stack file is used to save the stack contents for subsequent sessions. This method enables a truly distinct working session environment where you can pick up just where you left off in the prior session.

The command file (also known as the transfer file) is used to minimize user interface; it allows a series of commands to be entered from a file instead of being entered one-by-one at a terminal. Associated with the transfer file command are positional variables and program/command execution control features. Positional variables are used in transfer files and CI command strings for passing values. Conditional branching commands (for example, IF-THEN-ELSE-FI and WHILE-DO-DONE) are also allowed in transfer files to control program or command execution. These features are particularly useful in program development.

CI allows multiple commands in a single entry and the use of quotes for value passing. These and the features mentioned above are further described in Chapter 2.

Introduction to CI Files

Files contain various types of information organized in a specific manner to facilitate storage, access, and data transfer. The information can be text (usually called ASCII data), data collected from some experiments or tests (binary data), information used by programs, or even programs themselves.

Files are identified by file names. Additional information is added to the file name to describe the file's associated properties. The storage location of the file is recorded in a directory. The file name includes a file type extension that further describes the type of information contained in the file. The file name and the associated attributes that identify a file are called a file descriptor. Colons, dots, and slashes are delimiters in file descriptors. File properties are described in Chapter 3. Following are the various forms of file specification.

`proga`	(File name with blank file type extension)
`prog.rel`	(File name)
`userManualchap2.txt`	(File name)
`progb.rel::directory`	(File descriptor)
`/directory/progb.rel`	(File descriptor)
`/directory/subdirectory/chapter6:::4:609`	(File descriptor)

In the CI file system, files can be grouped under unique directories or subdirectories. Subdirectories can be nested within other subdirectories. Thus a hierarchical file structure can be established.

File protection is based on directory ownership. The creator of directories or subdirectories becomes their owner. Ownership can be changed either by the owner or the System Manager. The owner of a directory or subdirectory can restrict file access on that directory (or subdirectory or individual files within a directory) by other general users.

Time stamping of all files is automatic in the CI environment. The CI file system maintains three time stamps: Time of creation, time of last access, and time of last update. Time stamps can be used to search, access, and purge files using the file mask feature.

CI files can be specified as a group using a file mask. The file mask includes a field appended to the file name parameter in the file descriptor. This field, the mask qualifier, determines the grouping of files. For example, you can use the mask qualifier in a file listing command (DL) to tell the file system to search everywhere in the system and not follow the hierarchical directory structure, or you can specify a search for files created on a certain date.

If you have the optional DS/1000 Distributed System, you can access files on other systems in the network using the CI DS transparency feature. Remote file access is described in Chapter 3.

When CI Is Not Available

CI may be busy when you want to enter another CI command. This usually happens if the program you are running takes a long time or gets in trouble, such as running out of printer paper when running PRINT.

A copy of CI called CM is kept waiting to handle such situations. CM is available to all users, except from the system console, when the System Manager has enabled it. Interrupting the system while CI is busy will run CM, if possible. You can interrupt the system by pressing any key on your session terminal except for terminals connected to multiplexer ports which have fifo buffering or type-ahead turned on. In all cases, pressing the break key on your session terminal will interrupt the system. Once you get the system's attention, the prompt

```
CM>
```

is displayed.

CM differs from CI in two ways: CM always runs programs without wait, and it exits after completing a single command. It lets you obtain system status quickly. In addition to getting a system status report, you can use CM to stop the program for which you are waiting. The TR, EX, IF-THEN-ELSE-FI, and WHILE-DO-DONE commands (explained in detail in Chapter 5) and the command stack cannot be used in CM. The RS command is available only in CM.

After executing one command, CM terminates and control is returned to CI. If the busy situation persists, you need to get the system's attention again to run CM and execute another command.

Be careful when running CM; it is designed as a backup user interface, not for tasks that take a long time, such as copying files. Doing such tasks keeps others from using CM, because there is only one copy. Use CM only for simple commands, such as OF, RS, UP, or WH.

At times CM may be busy, either with a normal command or because some operation has blocked. In these cases, the system will issue the break mode prompt:

```
S=xx Command?
```

where xx is the session identification number.

If your system does not have CM enabled and you interrupt the system, only the break mode prompt is issued. Any of the system commands (explained in detail in Chapter 5) can be issued at the break mode prompt.

2

System Commands

This chapter describes the CI system commands used in obtaining system status information and controlling certain system operations. File manipulation and program control commands are discussed in Chapters 3 and 4. If you are familiar with FMGR and the RTE-6/VM Operating System, you need only read about those commands specific to CI; for example, ECHO, RETURN, SET, and so on. You can then skip to Chapter 3 for the CI file system information.

CI provides system commands that you may use in the CI environment to display system status, get help information, and control certain system operations. A summary of system commands is provided in Table 2-1.

Commands that affect system processes and control system operations are reserved for the System Manager. These commands are shown in Table 2-2.

Table 2-1. System Commands

Command	Task
`?` (or `HELP`)	Display help summary
`?` *command* or `HELP` *command*	Display command description
`/`[*parameters*]	Access command stack
`BL`	Display buffer limits
`CN` *lu* [*function* [*pram*4*]]	Control I/O device
`ECHO` [*parameters*]	Display command parameters
`EQ`	Display EQT information
`EX`	Exit CI
`OF` *prog* `[ID]`	Terminate program
`PR` *prog* [*priority*]	Display/change program priority
`RETURN`[,*return1-5* [,*return_s*]]	Return value(s) and/or string

Table 2-1. System Commands (continued)

Command	Task
`SET` [*variable* = *string*]	Define/display variable
`SL`	Display LU information
`ST`	Display program status
`SZ`	Display/modify program size
`TI`	Display system time
`TM`	Display formatted system time
`TO`	Display device timeout
`TR` *file* [_parm*9_]	Transfer to command file
`UL` *label*	Unlock shareable EMA partition
`UNSET` *variable*	Delete a variable
`UP` *eqt*	Up a device
`UR`	Release reserved partition
`VS`	Display VMA size
`WH` $\begin{bmatrix}\texttt{AL}\\ \texttt{PA}\\ \texttt{SM}\\ \texttt{PR}\\ \texttt{PL}\end{bmatrix}$	System status report
`WHOSD` *directory* | *lu*	Display directory/volume status
`WS`	Display working set size

Table 2-2. System Manager Commands

Command	Task
`AG` [*number*|`OF`]	Modify partition aging
`BL` [*lower* [*upper*]]	Modify buffer limits
`TM` *year date hour min sec*	Modify system time
`TO` *eqt* [*interval*]	Set device timeout
`UL` *label*	Unlock any shareable EMA partition
`UP` *eqt*	Up any device
`UR` *partition*	Release any reserved partition
`VS` *prog* [*lastpg*]	Modify VMA size
`WS` *prog* [*wrksz*]	Modify any working set size

Getting Help

The CI program provides an online help summary and a quick reference guide. The help summary or a brief explanation of any command or item listed in the summary can be displayed with the HELP command.

The help command can be entered as ? or HELP. This gives a list of the commands by their command mnemonics and other useful items such as a description of file mask. For example, here is the response from a ? command:

```
CI> ?
Help available on:  (use ? <command> for help on <command>)
directory /HELP.DIR
??          AG          AS          ASK         BL
BR          CI          CL          CN          CO
CR          CRDIR       CU          DC          DL
DN          ECHO        EQ          ERROR       EX
FOWN        FPACK       FREES       FVERI       GO
HE          IF          IN          IS          IT
LI          LINDX       LINK        LU          MACRO
MASK        MC          MERGE       MO          MPACK
OF          ON          OWNER       PATH        PR
PRINT       PROT        PU          QU          RN
RP          RU          SCOM        SET         SL
SS          ST          STACK       SZ          TI
TM          TO          TR          UL          UNPU
UNSET       UP          UR          VS          WD
WH          WHILE       WHOSD       WS          XQ
```

To get information about a command, enter the command name after the help command. For example, to get information about the OF command, entering “? of” displays the following:

```
CI> ?
OF -- Terminate a program and optionally remove its ID segment or
remove a prototype ID segment

Usage: OF [prog[/session][opt]]

prog is either a program name with optional session qualifier or
the name of a prototype ID segment if the D option is given.  The
default is the last scheduled unprotected program on the primary
program (usually CI) chain.
    :
    :
More...[90%] a
```

In addition to commands and explanations, other information is available with the help command, including such items as file descriptor, file mask, and file type extension.

It is possible to add items to this list. The help command works by listing a file contained in a directory called HELP. You select the file it lists when you ask for help on a particular command. Entering the help command without any parameter displays the contents of a directory called HELP. By adding files to this directory, you can increase the number of items listed in the help summary.

Using the Command Stack

As command lines are entered at the terminal keyboard, they are saved in a stack for reference or reuse. The number of command entries in the stack varies depending on the length of the entries. A minimum of 20 entries will be saved; however, the average is approximately 300 commands.

If the stack is full, the oldest commands in the stack are removed to make room for new commands. Duplicate commands are not saved in the stack. Commands entered from a command file are not saved in the stack. Command lines in the stack can be edited and reentered or simply reentered without retyping.

Commands in the stack can be saved in a file. By default, a file called CI.STK on the working directory is used. (CI uses CI.STK on the home directory if one is defined; otherwise, it uses the working directory when you log on.) Another file can be created or selected to hold the command stack. Refer to the command stack and the WD command descriptions in Chapter 5 for details.

If you do not want to save your command stack in a file or do not want the file updated, set the predefined variable $SAVE_STACK to FALSE. Refer to the section on “Predefined Variables” for details.

To display the command stack, enter a slash:

```
CI>  /
--020/320--  Commands:  [/DEXTER/CI.STK]
wd  /mine/myprograms
dl
     .
     .
     .
li syslog
pu syslog
ru print report.txt
co 8 report1.sale
co 8 report2.sale
co 8 report3.sale
co 8 report4.sale
co 8 report5.sale
co report5.sale 4
prot report5.sale rw/
```

(A screenful of commands, defaulting to 20, is displayed. See SET command description in Chapter 5 for changing the default)

The command stack window is preceded by a banner that contains the starting line number of the window and the total number of lines selected for this display in inverse video, separated by a slash, and the name of the current stack file in square brackets, for example, [newfile].

Note that the cursor is at the bottom of the stack. Pressing the return key returns to CI, and a new command can then be entered. The cursor can be moved to any line using the terminal cursor control keys. The line can be edited using the local editing keys of the terminal. When the carriage return key is pressed, the line is entered as if it was typed from the terminal keyboard.

You can recall just the last command with the cursor positioned on the command line. This is done with two slashes. For example:

```
CI> //
--001/320--  Commands: [DEXTER/CI.STK]
prot report5.sale rw/
```

In this example, pressing the carriage return key repeats the last command. To display the stack with the cursor positioned on the second to the last line, enter three slashes. The number of lines backward from the last line can be specified with the corresponding number of slashes after the command stack command (the first slash). A maximum of 80 slashes is allowed.

You can enter a slash followed by a number (/n). In this case, a screenful of commands is displayed beginning with the line number specified (backward from the last line). The cursor is positioned on the line specified.

At this point, either enter the command or use the terminal editing keys to change the entry. Pressing the carriage return key enters the command line. If you do not wish to enter this line, you can move the cursor to a blank line and press the return key to return to the CI prompt, or you can enter a slash to repeat the whole command stack.

After you have displayed the command stack and before you return to the CI prompt, you can use stack mode commands in addition to terminal keys to manipulate the stack and mark groups of commands. Refer to the command stack descriptions in Chapter 5 for details. You can display all command lines containing a specific string by entering a slash followed by a period followed by the string (for example, /.string). If you insert a caret (^) between the period and the string, only the command lines starting with the string are displayed. Refer to Chapter 5 command stack descriptions for examples.

You can change the command stack display size by using the SET command. For example, the following command changes the command stack display size to 15 lines:

```
CI> set frame_size = 15
```

Obtaining System Status

You can display the following system information on your terminal screen with the appropriate CI commands. How to use these commands to get the desired information is explained in this section. Note that these are the most common commands. A full description of the status command WH is given in Chapter 5.

Status of your programs
Status of all programs
Status of system memory usage
Status of system devices

Display Program Status

To display the status of your programs, use the WH command as shown in the following example.

```
CI> wh

10:49:44:420

PRGRM   T   PRIOR  PT  SZ  DO.SC.IO.WT.ME.DS.OP.    .PRG CNTR. .NEXT TIME.
--------------------------------------------------------------------------
**PROG2 6   00090    5 18  * * * *  3,CI.65 * * * * * P:45031
  CI.65 6   00051   17 32  . . . .  3,WHZAT . . . . . P:21171
  WHZAT 1   00002    0   . .  1,  . . . . . . . . . . P:61647
--------------------------------------------------------------------------
ALL LU’S OK
ALL EQT’S OK
LOCKED LU’S (PROG NAME)  67(EDI67),  73(EDI73),  76(EDI76),
--------------------------------------------------------------------------
10:49:45: 60
```

In this example, there are three programs in memory. The display shows their status. Their names are listed in the leftmost column. Note that one of them is the WHZAT program which is run when the WH command is entered. This program is often abbreviated to WH. The other two are the command interpreter CI and a program called PROG2. CI is waiting for WH to finish. PROG2 is running at priority 90, but since WH is running at a higher priority, WH preempts PROG2 while it is printing its information. The status column shows what the programs are doing. The common conditions are shown in the following example. The information in the other columns is explained fully in the WH command description in the *RTE-6/VM Utility Programs Reference Manual,* part number 92084-90007.

To display the status of all programs, enter:

```
CI> wh al
```

A sample display is shown below.

```
10:51: 5:760
PRGRM   T   PRIOR   PT SZ  DO.SC.IO.WT.ME.DS.OP.    .PRG CNTR. .NEXT TIME.
--------------------------------------------------------------------------
**FMG65 3   00052    5 18  * * * *  3,CI.65 * * * * * P:45031
  CI.65 6   00051   17 32  . . . .  3,WHZAT . . . . . P:21171
  WHZAT 1   00002    0  .  .  1,  . . . . . . . . . . P:61647
.
**FMG54 3   00052    8 18  * * * *  3,EDI54 * * * * * P:45031
  EDI54 6   00051   10 32  . . . 2,EQ: 13,AV:2,ST:002 P:23532
**FMG76 3   00052   20 18  * * * *  3,EDI76 * * * * * P:45031
  EDI76 6   00051   16 32  . . . 2,EQ: 35,AV:2,ST:000 P:23532
**FMG73 3   00052   18 18  * * * *  3,RUN73 * * * * * P:45031
  RUN73 4   00075   19 18  . . . 2,EQ: 32,AV:2,ST:002 P:44573
**FMG67 3   00052   11 18  * * * *  3,EDI64 * * * * * P:45031
  EDI67 6   00051   12 32  . . . 2,EQ: 26,AV:2,ST:000 P:23532

.
  SPOUT 1   00011    0  .  . . . .  3,CL  046 . . . . P:41706
  UPLIN 1   00003    0  .  0, . . . . . . . . . . . . P:00000 10:51: 8: 80
  GRPM  1   00004    0  .  . . . .  3,CL  060 . . . . P:51050
  RTRY  1   00020    0  .  . . . .  3,CL  059 . . . . P:55325
  LUMAP 2   00030    4  6  . . . .  3,CL  045 . . . . P:36045
  LOGON 3   00049   23 12  . . . .  3,CL  062 . . . . P:40535
  LGOFF 3   00053   14 10  . . . .  3,CL  063 . . . . P:37227
  R$PN$ 3   00005   26  4  . . . .  3,CL  061 . . . . P:36032
  RSM   3   00020   22  4  . . . .  3,CL  057 . . . . P:42025
  RFAM  3   00030   25 10  . . . .  3,CL  049 . . . . P:55153SWP
  EXECM 3   00030    7  4  . . . .  3,CL  052 . . . . P:41525
  OPERM 3   00030    6  3  . . . .  3,CL  054 . . . . P:37464
  PTOPM 3   00030    9  8  . . . .  3,CL  050 . . . . P:37601
  EXECW 3   00030   15  9  . . . .  3,CL  051 . . . . P:37706
  DLIST 3   00030    6  4  . . . .  3,CL  053 . . . . P:42053SWP
  QCLM  3   00028    5  3  . . . .  3,CL  065 . . . . P:36026SWP
  INCNV 3   00020    3  3  . . . .  3,CL  056 . . . . P:37303SWP
  OTCNV 3   00020    4  3  . . . .  3,CL  055 . . . . P:37247SWP
  CI.71 6   00051   13 32  . . . 2,EQ: 30,AV:2,ST:002 P:42235
--------------------------------------------------------------------------
ALL LU’S OK
ALL EQT’S OK
LOCKED LU’S (PROG NAME)  67(EDI67),  76(EDI76),
--------------------------------------------------------------------------
10:51: 8:370
```

This display is similar to the previous example except that it includes other programs, either system programs or programs belonging to other users. Programs are grouped by owner; those at the bottom of the list are system programs.

Display Memory Usage

How the operating system uses memory is indicated by the partition status. Use the WH command to display the programs that are in the partitions. You can use this information to tell if you have enough memory in your system for all the programs you want to run. To display the partition status, enter the following:

```
CI> wh pa
10:51:48:550
 PTN#   SIZE       PAGES   BG/RT SHR/LBL #ACT L  PRGRM       PTN-PRIOR
------------------------------------------------------------------------
 1 R      32       65-   96 BG                    D.RTR              1
 2M      200       97-  296 BG                   <NONE>
 3S       32       97-  128 BG                   <NONE>
 4S       32      129-  160 BG                   LUMAP             294
 5S       32      161-  192 BG                   FMG65             170
 6S       32      193-  224 BG                   OPERM             294
 7S       32      225-  256 BG                   EXECM           26192
 8S       20      257-  276 BG                   FMG54            2854
 9S       20      277-  296 BG                   PTOPM            1578
10        32      297-  328 BG                   EDI54              51
11        32      329-  360 BG                   FMG67            1374
12        32      361-  392 BG                   FDI67              51
13        32      393-  424 BG                   CI.71              51
14        32      425-  456 BG                   LGOFF             137
15        32      457-  488 BG                   EXECW             292
16        32      489-  520 BG                   <NONE>
17        32      521-  552 BG                   CI.65              51
18        32      553-  654 BG                   FMG73              76
19        32      585-  616 BG                   RUN73              75
20        50      617-  666 BG                   FMG76              52
21M       50      667-  716 BG                   <NONE>
22S       25      667-  691 BG                   RSM               282
23S       25      692-  716 BG                   LOGON             129
24M       51      717-  767 BG                   <NONE>
25S       26      717-  742 BG                   LUQUE              25
26S       25      743-  767 BG                   R$PN$              85
27-50   <UNDEFINED>
------------------------------------------------------------------------
MAXIMUM PARTITION SIZE AVAILABLE
RT   50 PAGES, BG   50 PAGES, MOTHER   200 PAGES
MAX. PART. SIZE GUARANTEED AVAILABLE -DUE TO SHAREABLE EMA
RT   50 PAGES, BG   50 PAGES, MOTHER   200 PAGES
------------------------------------------------------------------------
10:51:50:810

CI>
```

Display I/O Configuration

Systems differ widely in the number and types of peripherals such as disks, printers and tape drives. The I/O configuration of a system is the way that these devices are connected to the system and identified. The operating system identifies each device by a logical unit (LU) number or an equipment table entry (EQT) number. The LU or EQT number for a particular device can be used to specify that device. The I/O configuration information is displayed with the LUPRN utility or the SL command. The SL command displays an abbreviated status of all session devices or selected devices. The LUPRN utility displays a more detailed status of the system devices.

The LUPRN utility displays, on your terminal screen, a listing of what devices are on your system. For each LU, LUPRN shows the type of device attached to that LU, a select code identifying what I/O card the device is attached to, the subchannels associated with that device, and the device status. Refer to the *RTE-6/VM Utility Programs Reference Manual* for more details about LUPRN.

Following are examples using the SL command to display device information.

To display information about LU 6:

```
CI> sl 6
SLU   #  6=LU   # 6=E  6
```

The display shows the session and system LU numbers, the EQT number, and device status if the device was down (inoperative or offline).

To display all devices:

```
CI> sl
SLU    1=LU    # 65 = E   17
SLU    2=LU    #  2 = E    1
SLU    3=LU    #  3 = E    2
SLU    4=LU    #145 = E   17 S 1
SLU    6=LU    #  6 = E    6
SLU    8=LU    #  8 = E    8
SLU   20=LU    # 20 = E    2 S 1
SLU   23=LU    # 23 = E    2 S 4
SLU   26=LU    # 26 = E    2 S 7
SLU   36=LU    # 36 = E    2 S17
SLU   47=LU    # 47 = E    2 S28

CI>
```

To display I/O information using the LUPRN utility:

```
CI> luprn
                        RTE-6 System Device Configuration
                   RTE-6 System rev = 6000  LUPRN's rev = 6000
               9:54 AM WED., 21 APR., 1993...Sorted by Session LU
 Time Base (11B)   Priv. Fence SC (none)   Partitions (35)   Memory size (1024K)

SLU  LU  EQT,sc SCD Flags AV T.out  Stats  Driver  DP Device Name         LU SLU
--------------------------------------------------------------------------------
  1   73  13    21B  BPS   2 327.67    2B  DVM05   47  8-CH MUX (DDV05)   73   1
  2    2   1    12B D                102B  DVR32    4  7905/6/20/25 DSK    2   2
  3    3   2,10 13B D                100B  DVP32   50  7905/6/20/25 DSK    3   3
  4  108  13,1  21B  BPS   2 327.67    2B  DVM05   47  Left CTU @ LU 80  108   4
  5  109  13,2  21B  BPS   2 327.67    2B  DVM05   47  Right CTU @ LU 80 109   5
  6    6   6    20B  B               131B  DVB12   44  2608A Printer       6   6
  7    7   7,1  30B   PS        .02        DVA65   53  DS1000 to 1000      7   7
  8    8   8    15B  B S       5.00    1B  DVR23   56  9TK Mag Tape #0     8   8
  9    9   9    25B  B S       5.00    1B  DVR23   56  9TK Mag Tape #0     9   9
 10   10   1,4  12B D                120B  DVR32    4  7905/6/20/25 DSK   10  10
 11   11   1,5  12B D                120B  DVR32    4  7905/6/20/25 DSK   11  11
 12   12   1,6  12B D                120B  DVR32    4  7905/6/20/25 DSK   12  12
 15   15   1,9  12B D                120B  DVR32    4  7905/6/20/25 DSK   15  15
 17   17   1,10 12B D                120B  DVR32    4  7905/6/20/25 DSK   17  17
 24   24  10    24B D S       60.00   40B  DVM33   44  CS-80 Tape Drive   24  24
 30   30   2,2  13B D                100B  DVP32   50  7905/6/20/25 DSK   30  30
 48   48   2,13 12B D                120B  DVR32    4  7905/6/20/25 DSK   48  48
 61    4   4    17B  PS                4B  DVA66   53  HDLC/BiSync card    4  61
 62    5   5    17B  PS                    DVA66   53  HDLC/BiSync card    5  62
 65    3   2,10 13B D                100B  DVP32   50  7905/6/20/25 DSK    3  65
--------------------------------------------------------------------------------
    DP=Driver Partition page ($=SDA), SLU=Session LU
    (T.out is in seconds)                EQT Flags:
    LU # with a      EQT availability:     D=DCPC, B=Buffered, T=Timed-out
    D means the       1=down, 2=busy,      P=Driver handles Powerfail
    LU is down.       3=waiting DCPC       S=Driver handles Timeout
```

Controlling Devices

At times, you may need to control the operations of an I/O device from the terminal, such as to rewind tape or eject paper. The system performs these operations in response to the device control requests. To control devices interactively, the CN command is used to send the control requests. This is done by entering the CN command with the proper command parameter.

The common functions and the command parameters required are listed below.

Function	Command Parameter
Reset device	0 (zero)
Top-of-Form (paper feed)	TO
Rewind tape	RW
Write End-of-File	EO
Forward one file	FF
Backward one file	BF
Forward one record	FR
Backward one record	BR

The following examples illustrate some typical device control requests. In these examples, the printer is LU 6 and the magnetic tape drive is LU 8. For some devices such as tape drives and printers, the parameters may be omitted for the most common operations. For example, if CI recognizes an LU as a tape drive or printer, it assumes that the command without any control parameter is rewinding tape or ejecting paper, respectively.

```
CI> cn 6                 (Eject printer paper on LU 6)

CI> cn 8                 (Rewind tape drive on LU 8)

CI> cn 8 ff              (Advance tape to the next file)

CI> cn 8 bf              (Rewind tape to the previous file)

CI> cn 8 fr              (Advance tape to the next record)

CI> cn 8 br              (Rewind tape to the previous record)
```

In each RTE computer system, there are many different types of devices that are controlled by a software interface module called a device driver. There may be additional parameters needed for controlling a peripheral device. Refer to the appropriate RTE-6/VM driver reference manual for details.

Various other I/O control requests can be issued with the CN command. For example, the following command sets up multiplexer ports on the REV.C MUX (IDM00):

```
CI> cn 1 30b 152331b
```

In this case, the numbers are octal parameters required for the multiplexer specified in the multiplexer documentation. This entry sets up LU 1 as a 9600 baud terminal on port 1 with the standard options. The equivalent for the Revision D MUX (IDM00) is:

```
CI> cn 1 30b 131b
```

The CN command treats LU 1 as the system LU 1, rather than as your terminal. Other control requests are described in the CN command description in the *RTE-6/VM Terminal User's Reference Manual*, part number 92084-90004.

Changing I/O Device Attributes

The CI program provides several commands that modify I/O operations. The operating system maintains a set of attributes for each device. The common attributes include the operational status of the device and the waiting period to complete an I/O request. Most of the attributes are set up when the system is created; details are contained in the *RTE-6/VM System Manager's Reference Manual*, part number 92084-90009. Some of the attributes can be modified with CI commands if necessary. Modifications made with CI commands remain in effect until the system is rebooted or another modification is made to the same device.

In addition to the attributes mentioned above, there are others that can be changed in special situations. These are the device HP-IB address, device priority (not to be confused with program priority), and the driver parameters specified at generation time. These can only be changed by the System Manager.

How to Bring Up a Device

One of the most important attributes of a device is whether it is working or not. The RTE system maintains the device status, whether a device is "up" (working) or "down" (not working). All devices are initially assumed to be working; if the operating system finds out that a device is not working, it suspends I/O operations to the device until the situation is corrected. The UP command notifies the system that a particular device has been fixed. For example, to notify the system that the magnetic tape unit (LU 8, EQT 8) is operational, enter:

```
CI> up 8
```

(The EQT number, not the LU number, is used)

This allows I/O requests to go to the device. If the original problem recurs, the device goes down again. This happens for various reasons. A device may be inadvertently taken offline, effectively disconnected from the system. Tape drives go offline because most tape save/restore utilities put the tape drive offline when they are finished to allow removal of the tape and to prevent another user from using the tape drive. Printers are taken offline for manual form feeds. Whenever a device is offline and you need to access it, you must first place it online and bring up the device. Otherwise, the device cannot complete the control request and the operating system marks the device as down.

When the operating system detects a downed device, a message is displayed:

```
CI> cn 8
IONR L* 4E 14 S***
```

This indicates that the device is unavailable until the problem is fixed. Note that only the first request to a downed device gets this message. Subsequently, all I/O control requests to that device are placed on hold. It can be mysterious to have programs waiting to access a downed device, because the programs seem to be waiting for no reason. Either the WH or LU command can be used to find out if there are any downed devices in the system. To bring up your terminal, use the EQT number for that terminal and not LU 1.

The LU command can be used to determine a device's EQT number. See the discussion of the LU command in Chapter 5 of this manual and the SYSTEM LU command in the *RTE-6/VM Terminal User's Reference Manual* for more information.

Changing Timeout Values

In most cases, something is wrong if an I/O operation takes too long. A disk or printer should always respond within 1 second and a disk I/O operation should complete in 5 seconds. The operating system detects a problem via a "timeout," a device attribute that tells the system how long to wait for a response from the device.

Each LU has an associated timeout value. When the system starts an I/O operation, it also starts a timeout timer. If the timer goes off before the operation completes, the system takes the appropriate action, determined by the driver, which varies from device to device. Usually the device is noted as downed, awaiting user intervention.

Timeout values are specified either during system generation or with the TO command, in units of 1/100th of a second. A timeout value of 100 means one second. This unit is chosen to match the resolution of the time base generator. The associated EQT number of an LU is required to specify the timeout value.

To set the timeout on LU 8 (EQT 8) to 10 seconds, enter the following:

```
CI> to 8 1000
```

This sets the device with an EQT number of 8 (by normal convention a magnetic tape unit) timeout value to 10 seconds.

There are two other useful forms of the TO command. Specifying a timeout of zero really requests an infinite timeout. This is useful for devices where there is no limit to how long it might take I/O to complete. The most common example is a terminal; there is no particular time limit for entering commands, so it is reasonable to set terminal timeout values to zero.

Entering the TO command with an EQT number but without any timeout parameter displays the timeout value currently in effect for that LU. To display the timeout value for your terminal, use the system EQT number for the terminal, not LU 1.

Displaying System Time

The current system time can be displayed with the TM command. Although this command is also used to reset the system time, it is typically used only by the System Manager or installer for this purpose. To display the current system time, enter:

```
CI> tm
Wed Mar 2, 1989 7:39:34 am
CI>
```

It is important to maintain the correct system time; otherwise, RTE features such as time scheduling programs and time stamping files cannot be used effectively. Read about the TM command in Chapter 5 to learn how to set and reset the system time.

Executing a Command File

You can create a command file, using the EDIT program, to execute a series of commands without user intervention. The file contains all the commands to be executed in the desired sequence.

To execute commands, the command file name is entered with or without the TR command. At the end of the command sequence, CI returns to the source of the TR command, either another command file or the interactive mode. We recommend that you use the file type extension .CMD on all your command files. Note that the .CMD file type extension is required if the file is to be found via UDSP#2. Refer to Chapter 5 for a detailed description of the PATH and TR commands.

The following is a sample command file REPORT.CMD on the working directory:

```
co report01::src datafile1::data
co report02::src datafile2::data
:
co report30::src datafile30::data
pu report01::src ok
pu report02::src ok
:
pu report30::src ok
pu /src ok
```

To execute REPORT.CMD, enter any of the following:

```
CI> tr report.cmd

CI> tr report

CI> report

CI> report.cmd
```

Positional Variables

Positional variables are defined in the CI command string or in the TR command. The variable names are $1 through $9, where the number following the dollar sign indicates the position of the variable in the CI or TR command string. For example, either of the following commands sets the positional variables $1 through $4:

```
   CI> ru ci myfile.cmd prog1 prog2 prog3 prog4


   CI> tr myfile.cmd prog1 prog2 prog3 prog4
```

Positional variables can be separated by blanks or commas, but commas must be used to specify non-consecutive positional variables. For example, to transfer to a command file and specify values for only $1 and $4, enter:

```
   CI> tr myfile.cmd prog1,,,prog4
```

Three commas are required to ensure the value of positional variable $4 is PROG4.

You can specify any string (for example, a number or a valid file descriptor) for the positional variables; unspecified positional variables are set to null. If more than 9 variables are specified, only the first 9 values are used and the extra values are ignored. The command string containing the positional variables can be a maximum of 256 characters, including delimiters. Positional variables cannot be deleted.

Once values are set for the positional variables, they are used until CI terminates, another TR command is executed, or you exit from a command file.

The values of positional variables are local. Before executing a TR command, CI saves the current values of $1 through $9. While executing the command file, the values specified in the TR command string are used in the variable substitutions. When the command file is exited, the original values of $1 through $9 are restored.

A command file must be specified to set the positional variables; however, LU 1 (your terminal) can be entered instead of a command file name. For example, you can specify values for the positional variables as follows:

```
   CI> tr 1 myprog1 myprog2 myprog3 myprog4 myprog5
```

You then can use the positional variables in other CI commands; for example, LI $1 would list file MYPROG1 at the terminal.

Concatenation of variables is allowed. For example:

```
   CI> tr 1 R T E                        (3 parameters set up)

   CI> co $1$2$3AnswerFile SpoolA        (Copy file RTEAnswerFile)
```

User-Defined Variables

User-defined variables are defined using the SET command and deleted using the UNSET command. See Chapter 5 for a detailed description of the SET and UNSET commands.

User-defined variables are global; a variable defined in response to a CI prompt can be used in a command file, and a variable defined in a command file can be used in response to a CI prompt. Note that a user-defined variable is referenced by preceding the name with a dollar sign ($). For example, if the value of variable name is set as shown below,

```
CI> SET name = ‘RTE-6/VMPrimarySys‘
```

then the user-defined variable $name can be used in other CI commands and in any command file to represent the value ’RTE-6/VMPrimarySys’.

When referencing a user-defined variable, CI determines the end of the variable name to be the first character that is not valid for a variable name (valid characters are letters, digits, and underscores). For example, in the following command, the period indicates the end of the user-defined variable name:

```
CI> echo $file.ftn
```

This allows you to define similar variables names, such as $FILE, $FILENAME, and $FILENAME1.

Concatenation of user-defined variables is allowed. For example:

```
CI> set file = program1        (Define file name, file type extension, and
                               directory)

CI> set ext = .ftn

CI> set dir = ::mydir

CI> ftn7x $file$ext$dir        (Compile PROGRAM1.FTN::MYDIR)
```

Another example of concatenation is as follows:

```
CI> set dir = /system          (Define a directory)

CI> li $dir/answers            (List file /SYSTEM/ANSWERS)
```

Note that the slash (/) entered after user-defined variable $DIR is necessary. If you omit the slash, CI thinks the variable name is $DIRANSWERS because the blank after ANSWERS is the first invalid character for a variable name.

To concatenate two words, use the backslash (\) or the single character quote described in the section on “Quoting”. In the following example, variable NAME is set to user and the backslash is used to concatenate the variable with other ASCII strings.

```
   CI> set name = user

   CI> li $name\2                (List file USER2)

   CI> li $name\3                (List file USER3)

   CI> li $name\prog             (List file USERpROG)
```

Note that the character after the backslash is not changed to uppercase by CI. So if you need to enter a string such as PROG, you must type P instead of p.

You should delete unneeded user-defined variables. CI uses its free space to save variable names and values. If too many variables are defined, CI runs out of space and returns an error. This may affect user-defined and predefined variables; for example, CI may not have enough space to return a value to a variable.

Predefined Variables

When you begin a CI session, there are predefined variables. These variables are initialized to default values by CI. However, you can use the SET command to modify the values of all the variables except $MY_NAME. You can also modify $WD, but CI updates its value after each WD command. The SET and ECHO commands can be used to display the values of predefined variables. You cannot use the UNSET command to delete these variables.

The predefined variables are as follows:

$AUTO_LOGOFF

Allows for automatic logoff if session is inactive. CI initializes $AUTO_LOGOFF to zero, which means automatic logoff is not in effect. If you set $AUTO_LOGOFF to a non-zero value, CI times out after that many terminal timeouts. If CI is the only active program, after four CI timeouts an EX,B command is executed to terminate the session.

$DATC

The datecode revision of the operating system; for example, 6000 for Revision 6.0. This variable is for user information and can be deleted by the UNSET command.

$FRAME_SIZE

The size of the command stack display. When you log on, the command stack display size is initialized to twenty lines. It can be set to any positive integer greater than zero.

$HOME

$HOME is set to the directory in which CI starts up. $HOME cannot be deleted by the UNSET command.

$LOG

A flag indicating whether commands executed in a command file are logged to the terminal. CI initializes this variable to OF, which means that commands are not displayed at the terminal. To display commands at the terminal, set the value to ON.

$LOGON

The user and group name the user is associated with during the session, in the form user.group.

$MY_NAME

The true or schedule name of CI. This variable can never be altered.

$OLDPWD

$OLDPWD is set to the previous working directory ($WD) whenever a WD or CD command is executed.

$OPSY

The ID number of your operating system.

$PROMPT

The prompt that is displayed when CI is waiting for input. CI initializes this variable based on the name of the program file containing CI.

$RETURN1 - $RETURN5

Five integer values (ASCII representation) returned from execution of the last command. CI updates the values as commands are executed. These variables can be set to values between −32768 and 32767, inclusive.

$RETURN_S

An 80-character string returned from execution of the last command. CI updates the value as commands are executed.

$RU_FIRST

Flag indicating whether RU or TR is to be assumed if you only enter a file name in response to a CI prompt or as a line in a command file. CI initializes this variable to TRUE, which means CI first attempts to execute an RU command for the specified name. Set this value to FALSE if you want CI to assume that the file name entered is the name of a command file. You should set the variable to FALSE if you will be executing more command files than program files.

$SAVE_STACK

Flag indicating if the command stack is saved when you exit CI or when the command stack file is changed with the WD command. CI initializes this variable to TRUE, which means the stack should be saved. Set the value to FALSE if you do not want the stack saved.

$SESSION

Number of your current session. CI initializes this variable to your session number and updates the value after execution of every CI command.

$WD

Name of the current working directory. CI updates this variable after execution of each WD command.

The following example changes the value of $PROMPT:

```
CI> set prompt = waiting:
WAITING:
```

The following example displays the value of $OPSY:

```
CI> echo $opsy
-17
```

Nesting Command Files

Command files can be nested by using the TR command, implicitly or explicitly, in a command file. Before CI transfers control to a new command file, the positional variables ($1 through $9) are saved. Upon returning from one level of command file nesting to the previous level, these values are restored.

String Processing and Parameter Parsing

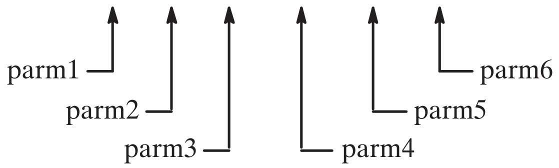
If you do not quote a character or a string, CI takes the following actions:

1. Shifts it to uppercase.
2. Strips any contiguous groups of blanks which precedes or trails a comma.
3. Replaces each remaining contiguous group of blanks with a comma.
4. Delimits command at semicolon.
5. Performs variable substitution before executing the command.

Thus, the string is shifted to uppercase and parameters are delimited by spaces and/or commas. For example:

```
           String Entered                                Parsed Value

CI>  now  is the,  time  ,for,all             NOW,IS,THE,TIME,FOR,ALL
CI>  now  ,   is the,time   for all           NOW,IS,THE,TIME,FOR,ALL
```



Substituted variable strings are processed in a slightly different manner. The case of the string is not altered and spaces are preserved in the substituted string. When parsing the parameters of the substituted string, each contiguous group of blanks which precedes or trails a comma is stripped but all other blanks in the string are preserved. This has the effect that substituted string parameters are delimited by commas but not blanks.

For example, if FIRST.CMD consists of:

```
set log = on
set params = ‘now  is the,     time     ,    for all,good‘
tr ech $params
```

and ECH.CMD consists of:

```
set log = off
echo >$1<
echo >$2<
echo >$3<
echo >$4<
```

executing FIRST.CMD results in the following:

```
CI> first
SET,PARAMS,=,now  is the,    time    ,  for all,good
TR,ECH,now  is the,    time    ,  for all,good
SET,LOG,=,OFF
>now  is the<
>time<
>for all<
>good<
```

If FIRST.CMD consists of:

```
set params = ‘now, is,the,    time    ,  for all,good‘
```

```
set log = on
set params = ‘now, is,the,    time    ,  for all,good‘
tr ech $params
```

executing FIRST.CMD results in the following:

```
CI> first
SET,PARAMS,=,now,  is,the,    time    ,   for all,good
TR,ECH,now,  is,the,    time    ,   for all,good
SET,LOG,=,OFF
>now<
>is<
>the<
>time<
```

Quoting

There are two methods of quoting available to allow characters to pass unaltered to the destination program, command file, or CI command. A single character is quoted by preceding it with a backslash (\). A string is quoted by enclosing it in backquotes (‘).

To include a backquote in a quoted string, enter a second backquote with the backquote you want passed as part of the string.

Some examples of quoting are as follows:

```
CI> echo ‘Hello.  How are you?‘
Hello.  How are you?
```

(Display string unaltered by CI)

```
CI> echo ru,savename,‘Jane Doe‘
RU,SAVENAME,Jane Doe
```

(Pass a blank in command string)

```
CI> echo ‘This is a backquote (“).‘
This is a backquote (‘).
```

(Pass a back quote as part of the quoted string)

```
CI> echo peek\npoke
PEEKnPOKE
```

(Display string with the “n” unaltered by CI)

```
CI> li $file\x
```

(Use the backslash (\) to tell CI that $file is the undefined variable (not $filex). If $file is Foo, this command will list FOOX.)

Multiple Commands per Line

You can enter more than one CI command per input line by separating the commands with semicolons (;). Blanks immediately before or after a semicolon are ignored. A semicolon used to separate commands must not be enclosed in quotes.

Two examples are as follows:

```
CI> wh;dl
```

This executes the WH program followed immediately by DL.

```
CI> ftn7x test.ftn 0 - ; link test.rel ; test
```

This compiles, links, and runs program TEST.

Return Status

Most commands, programs, and command files can return status to CI to indicate success or failure of execution. CI interprets the internal status returned by commands.

Programs and command files can return five integer values and a string to CI. The first of these integers is used for status. The rest of the values are additional information for the user. A status of zero indicates success; anything else indicates failure. The five integers are then made available to you in the string variables $RETURN1 through $RETURN5. The returned string is saved in variable $RETURN_S.

Programs pass CI the five integer values through the system routine PRTN and the string via an EXEC 14 call. Command files return these values using the CI RETURN command. See the RETURN command description for further details.

The return status is used by CI's execution control structures discussed later in this chapter. Note that the control commands IF-THEN-ELSE-FI, WHILE-DO-DONE, and the SET and ECHO commands do not alter the return variables. This is to assure that you can access these values before they are modified.

The AG, BL, CU, DN, EQ, IT, LU, OF, ON, PR, QU, ST, SZ, TI, TO, UL, UR, VW, and WS commands do not return status to CI; therefore, $RETURN1 always equals zero after any of these commands are executed.

Execution Control Structures

A powerful feature available in command files is the IF-THEN-ELSE-FI and WHILE-DO-DONE control structures, which enable decision making during execution of the command file. (See the discussion of each control structure Chapter 5 for more detailed information.)

The following statements compile TEST. If no errors or warnings occur during the compile, TEST is linked; otherwise, EDIT is run on TEST.FTN so you can fix the errors.

```
IF ftn7x test.ftn 0 -
THEN link test.lod
ELSE edit test.ftn
FI
```

In the next example, the file SOME_FILE is printed five times. The IS command compares the value of $COUNT and zero; as long as $COUNT is greater than 0, the WHILE loop continues executing. CALC is a simple user-written program that accepts two ASCII representations of integers, converts them to integers and performs the specified operation. The result, in ASCII, is returned to $RETURN_S.

```
set count = 5
WHILE is $count gt 0
  DO calc $count - 1
  set count = $RETURN_S
  print some_file
DONE
```

Timeout/Logoff Function

To eliminate inactive sessions on a system, CI can log off a user. The variable $AUTO_LOGOFF can be defined to tell CI how many device timeouts can occur at your terminal before CI times out. Each time CI times out, a warning message is displayed on the terminal. After the fourth timeout, CI executes an EX command.

The next example begins the CI timeout process after CI waits 15 minutes for input. Set the terminal timeout to 30000 (see the TO command) and the $AUTO_LOGOFF variable to 3. If CI receives no input for 60 minutes, the session is terminated.

```
CI> to 113 30000
CI> set auto_logoff = 3
CI>
Waiting for input...
Going...
Going...
Gone!
Finished
```

Note The $AUTO_LOGOFF feature does not work on RTE-6/VM systems when using a BACI Card.

3

Manipulating Files

This chapter describes how to use CI commands to manipulate files, directories, and subdirectories. Table 3-1 provides a summary of the commands. Included in this chapter are descriptions of file properties; this information will help you take full advantage of the file system.

File Properties

Each file has certain properties associated with it. Some properties describe the way information is organized within the file, and others contain information about the file, such as its location, ownership, protection, and time stamps. The file properties are listed below and described in the following paragraphs:

- file name
- file type extension
- directory
- subdirectory
- file type
- file size
- record length
- owner
- protection
- time stamps

File Names

Each file in a directory has a unique name, consisting of up to 16 characters, which distinguishes it from other files in the directory. (Duplicate file names may be used for files that reside in different directories.) The first character of each file name must be a letter, which is used to distinguish between file names and LU numbers that represent I/O devices.

A file type extension, consisting of a period followed by up to four characters, can be appended to a file name. Thus, a full file name, including a file type extension, can contain up to 21 characters. The following file name includes a file type extension that indicates it is a text file:

```
   CurrentManualCh1.txt
```

File type extensions are further discussed in the next section.

File names can be entered in upper or lowercase letters, and capitalization is optional. CI always shifts the input to uppercase. We recommend that you avoid using characters other than letters and numbers in a file name. “Reserved” characters are those that have special meaning to CI and/or cannot be used. For example, the slash (/), “at” sign (@), minus sign (−), left bracket ([), greater than sign (>), period (.), and comma (,) are reserved characters. The use of other punctuation characters should also be avoided. Although FMGR file naming does not have the same restrictions, note that problems could occur if you try to move FMGR files whose names contained reserved or other punctuation characters to a CI file volume.

It is possible to accidentally create a file name that has the high order bit in a character set. Such a name will print as a normal file name, but cannot be manipulated as a normal name, nor purged from CI. In this case, you must use FMP routines with the appropriate ASCII and non-ASCII characters to manipulate or purge the file name. FMP routines are described in Chapter 6 of this manual.

Temporary Files

A temporary file is one that is intended to be used while a program runs and is typically purged after the program finishes executing. The file system has a built-in mechanism for automatically purging temporary files. Temporary CI and FMGR files can be created using FMP routines. A temporary CI file is identified by a bit in the file’s directory entry. A temporary FMGR file is identified by having a number as the first character in its name. See Chapter 7 for additional information about temporary files.

I/O Devices Referenced as Files

In addition to identifying a file, the file name can be a number that identifies an I/O device. This number is a logical unit (LU) number assigned at generation time to all devices in the RTE system. The LU numbers for devices such as terminals and printers can be used in most cases where a file name appears. To try using LU numbers to indicate I/O devices, use the CO command to copy a file. Since your terminal is always LU 1, you can display a file to your terminal as follows:

```
   CI> co welcome.txt::system 1
```

Use of LUs is further described in the section “Data Transfer To and From Devices” later in this chapter.

Table 3-1. File Manipulating Commands

Command	Task
`CD [–` | *directory*`]` `CD` *old new*	Change working directory
`CL`	List mounted disk volumes
`CO` *src_file dest_file* `[`*pram*`]`	Copy file
`CR` *file*	Create file
`CRDIR` *directory_name* `[`*lu*`]`	Create directory on specified LU
`DC` *lu*	Dismount disk volume
`DL [`*mask* `[`*option* `[`*file* | *lu* `[`*msc*`]]]]`	Display directory contents
`IN` *lu* `[`*blocks* `[OK]]`	Initialize disk volume
`LI [`*flags*`]` *file* | *lu*	List file or LU
`MC` *lu*	Mount disk volume
`MO` *src_file* | *dir dest_file* | *dir*	Move file/directory
`OWNER` *directory* `[`*newOwner*`]`	Display/reassign directory owner
`PATH [–E]`	Display current UDSP information
`PATH [–E] [–N:`*n*`]` *udspnum* `[`*dirname1* `[`*dirnam2* `[...]]]`	Display/define specified UDSP or UDSP entry
`PATH [–E] –F,`*file* | *lu*	Display/define UDSP using commands from specified file or LU
`PROT` *file* `[`*newprot*`]`	Display/modify file protection
`PU` *file* | *dir* `[OK]`	Purge file/directory
`PWD`	Display working directory
`RN` *file* | *dir newname*	Rename file/directory
`UNPU` *file*	Unpurge file
`WD [`*directory_name* `[`*file* | `+S]]`	Display/set up working directory with option for posting or changing command stack file
`WHOSD` *file* | *directory* | *lu*	Display session using directory or disk LU as part of a UDSP

File Type Extensions

A file name may include a file type extension that indicates the type of information in the file, for example, text, binary data, or listing. The file type extension consists of a period and up to four characters appended to the file name. For example, in the file name parameter EDIT.RUN, the file type extension is .RUN. A blank file type extension is allowed, and is the default for some programs and commands. If you do not use a file type extension, you need not include a period after the file name.

Standard file type extensions should be used when files contain standard information. For example, all executable program files should have file type extension .RUN, relocatable files should have file type extension .REL, and all CI transfer files should have file type extension .CMD. The standard file type extensions are as follows:

Table 3-2. Standard File Type Extensions

File Type Extension	File
.cmd	CI command file
.dat	Data file
.dbg	Symbolic Debug/1000 file
.dir	Directory or subdirectory entry
.doc	Document file
.err	Error message file
.ftn	FORTRAN source file
.ftni	FORTRAN source include file
.hlp or .help	Help file
.lib	Library of relocatables
.lod	LINK command file
.lst	Listing
.mac	Macro source file
.maci	Macro source include file
.map	Load map list
.merg	Merge file for relocatables without headers
.mlb	Macro library file
.mnf	Manual numbering file
.mrg	Merge file for relocatable libraries with headers
.pas	Pascal source file
.pasi	Pascal source include file
.rel	Relocatable (binary) file
.run	Program file
.snp	System snapshot file
.spl	Spool system file
.stk	Command stack file
.sys	System file
.txt	Text file

When you specify a file, you must include the file type extension if there is one. If you specify only REPORT for a file named REPORT.TXT, you are implying a blank default file type extension, which does not match REPORT.TXT.

Some programs and program commands assume different default type extensions. For example, the CI program RU command uses the default file type extension of .RUN for programs scheduled without file type extension. Refer to the manuals appropriate for the programs you are using for any default file type extensions (for example, the manuals for EDIT and DEBUG/1000).

File Descriptors

A file descriptor is a term used to specify a file by means of its attributes, including file size, type, record length, subdirectories, and directory. Colons are used to separate the parameters, and slashes are used to separate subdirectories, directory, and file name. The following two examples show file descriptor formats:

filename : : *directory* : *type* : *size* : *record_length*

or

/ *directory* / *subdirectory* / *filename* : : : *type* : *size* : *record_length*

Note that the file name parameter includes the file type extension, if there is one. You must use a colon as a placeholder for each default parameter that is followed by another parameter.

The maximum length of a file descriptor is 63 characters, including delimiters, and we recommend that you keep file descriptors in the range of 40 characters, because FMP routines expand them to include unspecified attributes, which may cause them to exceed the limit. It is possible, however, to create files with file descriptors longer than 63 characters by using working directories or by renaming directories.

The file name parameter in the file descriptor can contain a mask qualifier that you can use to access multiple files. In addition, two wildcard characters, @ and −, can be used in the file name parameter. Refer to the “File Masks” section later in this chapter for details.

In the following examples of file descriptors, the file names in the user entries are shown in uppercase letters for clarity only. Directories and subdirectories in the comments are shown in uppercase letters as they are throughout this manual.

`MANUAL.TXT::op:4`	(Type 4, text file in directory OP)
`/op/output/OUTLINE.TXT:::4`	(Type 4, text file in subdirectory OUTPUT in directory OP)
`EDIT.RUN::programs`	(File in directory PROGRAMS)
`PROGRAMMERS:::3:356`	(Type 3, text file in working directory with a size of 356 blocks)
`/new/pascal.dir`	(Subdirectory PASCAL in directory NEW)

`/new`	(Directory NEW)
`/jones/@.@`	(All files in directory JONES)

Directories

Directories contain files and other directories, called subdirectories. Directories maintain information about files including their names, file type extensions, all the optional properties defined for the files, and their locations. Many directories can be on one system, and each directory can have multiple subdirectories.

Each directory has a unique name of up to 16 characters, subject to the same rules as file names except that a global directory name can be a single integer. The directory name can be specified along with the name of a file you want to access, but its use is optional.

Directories can be specified as follows:

::*directory_name*

or

/*directory_name*

In the first format above, the directory name is preceded by two colons, which separate the file name from the directory name. This form is generally used with FMGR files, and the CI file system may display this form for compatibility with FMGR files. The field between the colons is used by FMGR files to define an optional file security code; for example, file:sc:crn.

The second format is typically used if the files are organized in a hierarchical structure. Such a file structure may contain directories that have nested subdirectories. This form of specifying files is used to indicate the search path for the files in the CI, or hierarchical, file structure. Figure 3-1 illustrates the hierarchical file organization.

If the directory name is omitted in a file descriptor, a default directory called the working directory (WD) is used. The working directory can be defined or changed with a WD command. Once defined, the working directory remains in effect until changed by another WD command. You may display the name of the working directory by using the WD command without any parameter.

Certain programs contain a special feature that lets you schedule other programs without specifying the directory name. If you omit the directory in the program runstring, standard directories set up by the system are searched. For example, in executing the RU (Run) command, CI searches a directory named PROGRAMS for programs specified without a directory. The standard default directory search sequence used by CI is described later in this chapter.

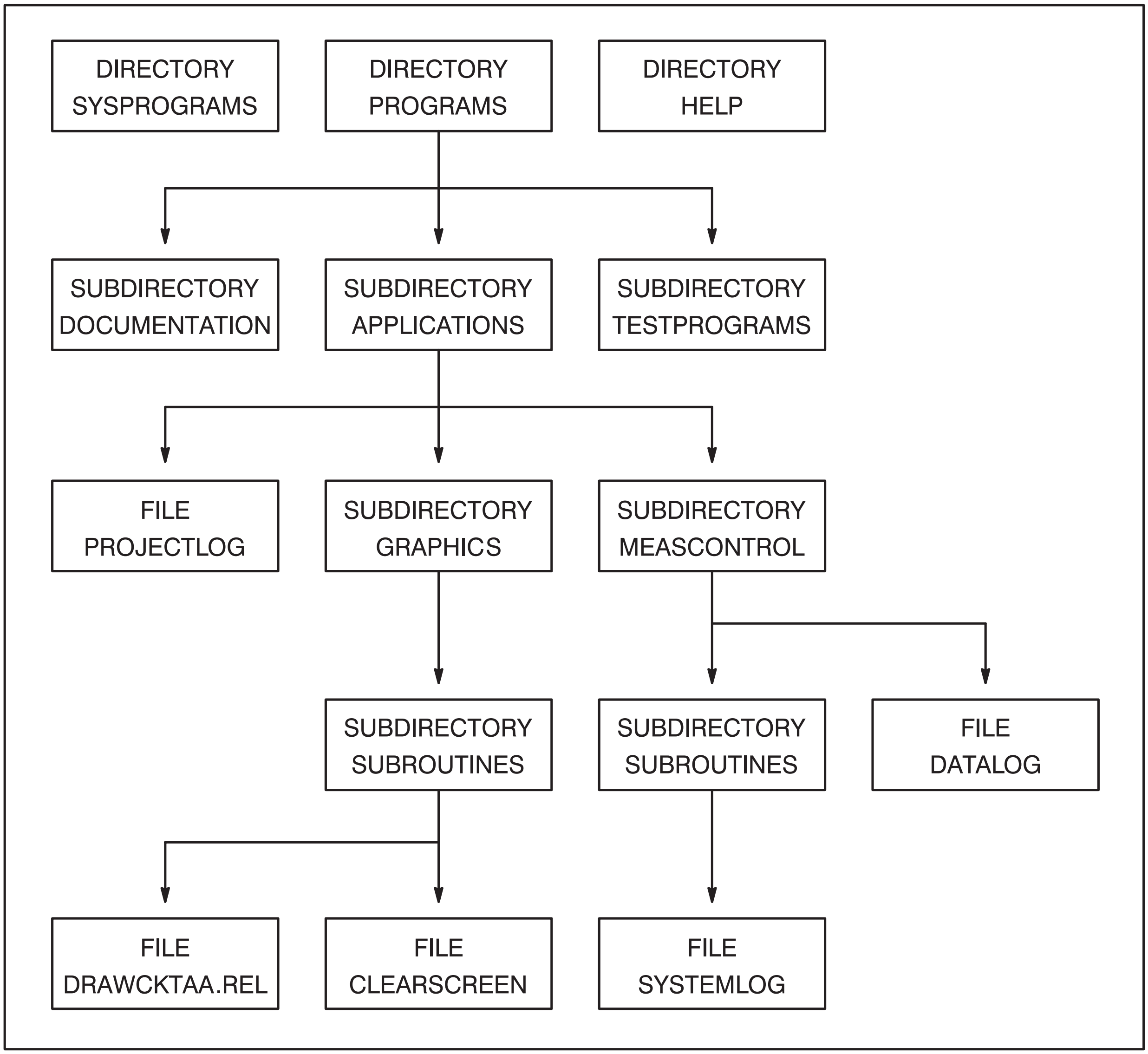


Figure 3-1. Sample Hierarchical File Organization

Subdirectories

Subdirectories are directories that are contained or embedded within other directories. Subdirectories in turn can contain other subdirectories, and there can be many levels of subdirectories. Unlike directories, subdirectories can have duplicate names as long as all names within a single directory are unique.

Subdirectories have the same properties as directories, and in this manual, references to directories also apply to subdirectories, unless otherwise noted.

When you want to specify a file that is in a subdirectory, use the hierarchical format, preceding the file name with the subdirectory name and separating the two names by slashes. For example, if a file named MANUAL.TXT is in directory DIR, it can be specified as follows:

```
/dir/manual.txt
```

If this file is moved to the subdirectory SUBDIR, it is specified as follows:

```
/dir/subdir/manual.txt
```

The first form above is used when there are no subdirectories. The second form is used to specify a search path in a hierarchical file structure in which there may be many levels of subdirectories. There is no limit to the number of levels of subdirectories that can be nested inside other directories; however, there is a limit to the length of the file descriptor (a maximum of 63 characters, including delimiters).

In a sample hierarchical file structure, shown in Figure 3-1, enter the following to specify a file named DRAWCKTAA.REL located in subdirectory SUBROUTINES:

```
/programs/applications/graphics/subroutines/drawcktaa.rel
```

There may also be a file with the same name in subdirectory APPLICATIONS. To specify this file, the following form is used:

```
/programs/applications/drawcktaa.rel
```

The hierarchical file structure provides a search path to minimize the search time and allow duplicate file names for files that reside in different directories or subdirectories.

In a hierarchical file specification, a directory name is always preceded by a leading slash; without the slash, the name is assumed to be a subdirectory. For example,

```
/system/archive/file.txt:::3
```

specifies that FILE.TXT is located in subdirectory ARCHIVE in directory SYSTEM, while

```
system/archive/file.txt:::3
```

specifies that FILE.TXT is in subdirectory ARCHIVE of subdirectory SYSTEM. (Since no slash precedes the name SYSTEM, it is assumed to be a subdirectory.) In this case, since a directory is not specified, the working directory is assumed, which means SYSTEM is a subdirectory of the working directory. In other words, the entire search path is different.

Directory Specifiers "." and ".."

Two other specifiers can be used to indicate the directory path to a file and are useful for moving around the directory tree without explicitly specifying subdirectory names. They are:

. – identical to "the current working directory"
.. – identical to "the parent of the current working directory"

These characters are used in place of specific directory names in a file descriptor.

For example, assume the directory structure is as shown in Figure 3-1 and the current working directory is /PROGRAMS/APPLICATIONS/GRAPHICS. To access the file PROJECTLOG, you can use the file descriptor

```
/PROGRAMS/APPLICATIONS/PROJECTLOG
```

or you can enter

```
../PROJECTLOG
```

Here the leading ".." is identical to "/PROGRAMS/APPLICATIONS" (the parent directory of GRAPHICS). Similarly, the file DATALOG can be accessed using

```
../MEASCONTROL/DATALOG
```

Multiple sets of ".." can be used to continue up the directory tree. For example, if the working directory is

```
/PROGRAMS/APPLICATIONS/GRAPHICS/SUBROUTINES
```

the file SYSTEMLOG can be reached with the following descriptor:

```
../../MEASCONTROL/SUBROUTINES/SYSTEMLOG
```

Here the first ".." refers to the parent of SUBROUTINES (GRAPHICS) and the second refers to the parent of GRAPHICS (APPLICATIONS). The remainder of the descriptor continues down from that point. An equivalent descriptor is

```
/PROGRAMS/APPLICATIONS/MEASCONTROL/SUBROUTINES/SYSTEMLOG
```

You can string the ".." characters together until the top level of the directory tree is reached. At that point, additional ".." sequences have no effect. For example, from /PROGRAMS/APPLICATIONS, the descriptor

```
../../../../../DOCUMENTATION
```

is identical to /PROGRAMS/DOCUMENTATION.

The “..” character refers to the working directory, and the descriptor “./xxx” is basically equivalent to the descriptor “xxx”. The reason for using “.” instead of just specifying the file alone is that some commands and applications treat the two descriptors differently. For example, the command

```
   ru prog.run
```

causes PROG.RUN to be picked up from the working directory or some other directory in the programs search path (refer to the discussion of the RU command for details on the search path). But the command

```
   ru ./prog.run
```

forces PROG.RUN to be picked up only from the working directory; that is, it disables the search algorithm.

Directory Specifier “#n”

You can use another leading character sequence,

#*n*

where *n* is a number from 0 to 8, in place of the directory name in a file descriptor. This instructs FMP to search for the specified file using UDSP number *n*. For example, the file descriptor

```
   #1/prog.run
```

tells FMP to search through the directories defined in UDSP number 1 to find the file PROG.RUN. (UDSPs are further described later in this chapter and in the discussion of the PATH command in Chapter 5.)

Note that some commands and applications use a default UDSP when searching for files; for example, the RU command uses UDSP #1. This sequence overrides the default UDSP. For example, the command

```
   ru #4/prog.run
```

causes a search for the file PROG.RUN within UDSP #4 instead of the default UDSP #1, which is normally used for the RU command.

File Type

Each file descriptor has a file type parameter that indicates how the information in the file is organized. The file type is a number and is not to be confused with a file type extension. There are standard RTE file types defined with the following characteristics:

Type 0 An I/O device. Type 0 is used in accessing devices with file calls. There is no disk file or directory entry for type 0 files, and they do not have the other properties listed in this section.

Type 1 Random access files. These do not have any structure information in them. These files contain fixed record lengths (128 words). They can be read and written very quickly.

Type 2 Fixed-length record, random access files. The record length is defined when the files are created. They are usually user-created, large data files.

Type 3 Type 3 and higher files are variable-length and higher record, sequential files suitable for use as text files. There is no difference in the handling of file types 3, 4, and 7. Type 3 is for general purpose files and can be used for text. This is the default file type when files are created with the CR command. Type 4 is recommended for text files. By convention, type 5 is used for Compiler or Assembler relocatable output files, type 6 is for program files that are memory-images of executable programs, and type 7 is for Compiler or Assembler absolute binary output files. Type 6 files are treated the same as type 1 files. Type 7 and higher files are user-defined.

File type is important when files are accessed programmatically. Substituting a random access file for a sequential file or vice versa will cause problems.

File type is specified after the directory name, separated by a colon. For example, you can create a type 1 file with the following entry:

```
CI> cr file.dat::system:1
```

If the subdirectory and directory are specified before the file name, the file type is preceded by three colons. For example:

```
CI> cr /system/subdir/file.dat:::1
```

The directory name was moved to the front of this case, but colons are required as placeholders. When creating a file in the working directory, placeholders are also required if the file type is specified. For example:

```
CI> cr subdir/file.dat:::1
```

File Size

The file size parameter in the file descriptor specifies how many blocks of disk space the file needs. One block is 128 words (256 bytes or characters). One printed page takes about 10 blocks of disk space. You can specify how big a file should be when you create it. If you do not specify the size and there is no information about the file, the file system chooses a size of 24 blocks. If the contents of the file are known, for example, when you create a file with the CO command, the exact size of the file is used.

The size of a file is specified in the file descriptor after the file type parameter, separated by a colon. For example, to create a file of 100 blocks, enter the following:

```
   CI> cr bigger.dat::system::100
```

To specify both size and type, enter the following:

```
   CI> cr macro.err::system:1:100
```

(Create a type 1 file, 100 blocks in length called MACRO.ERR in directory /SYSTEM)

Except for type 6 and some type 1 files, the file system automatically increases the file size to accommodate more data as needed through a process called “extending the file”. “Extents” are always at least as big as the original file size, because there are performance advantages to having fewer, larger extents. Type 6 and some type 1 files cannot be extended because they are memory images of programs of the RTE system.

Files that are larger than 16383 blocks are rounded off to multiples of 128 blocks. Files can be as large as any disk available in your system. Files larger than 16383 blocks must be created by specifying the size as a negative number of 128 block “chunks” of the file. For example, a 50000 block file is specified by a size of $-50000/128 = -391$, rounding to the nearest larger number in absolute value. Large files are usually created by a program and rarely by a user.

Be aware that a file size parameter larger than 32767 blocks will be accepted, but the desired file size will not be created. For example, the following command creates a file with 13110 blocks:

```
   CI> cr file:::36214
```

Record Length

Record length is the last parameter in the file descriptor, specified in units of words and used mostly for fixed-length, type 2 files. For example, the following entry creates a type 2 file of 100 blocks with 64-word records:

```
   CI> cr file.dat::system:2:100:64
```

This field is also used in type 3 and higher files. The file system uses the value of the longest record in the record length field. This value appears in messages displayed by the file system utilities to indicate the longest record. Any other value specified by you in type 3 and higher files is ignored.

File Ownership and Associated Group

A directory's owner is the user who created it. All files in a directory are considered owned by the directory owner. The associated group of a directory is the associated group of the directory owner. The same is true for subdirectories. However, the owner and the associated group of a subdirectory can be different from the owner and associated group of the directory that contains the subdirectory.

The directory owner can change the protection status of files in that directory. The protection status defines the read/write access allowed for the owner, members of the associated group, and general users. See the section on "Protection" below for more information. The directory owner (and no other user) can also reassign the directory ownership and the associated group.

An entire CI volume can have an owner and associated group. The initial owner of a volume is the user who initialized the volume, and the associated group is that user's associated group. A volume's ownership and associated group may be reassigned using the owner command.

See the section on "Manipulating Directories" in this chapter for more information about ownership, associated groups, directories, and subdirectories.

Protection

File protection is a security measure in the CI file system that governs read and write access. It is defined when a file is created or copied into a directory, and it can be specified differently for the owner, members of the associated group, and general users. The default file protection provides both read and write access for the owner and read access only for members of the associated group and other users.

When a file is created, it assumes the protection status defined for the directory on which it resides. A copied file assumes the protection status defined for that file if one exists; otherwise, it assumes the protection of the directory into which it is copied.

You can specify protection on a file-by-file basis, in any combination of read and write access for the owner, members of the associated group, and general users.

Directories also contain protection information that governs the protection status of any file created in that directory. For example, if a directory allows read and write access for the owner and read access only for members of the associated group and general users, all files created in that directory have that same protection status unless it is changed by the PROT command.

Read and write protection for a directory differs slightly from that for a file. Directories that are write-protected (read access only) prevent general users from changing information in the directory through CI commands; they cannot create, purge, or rename files in the directory. Directories that are read-protected prevent general users from finding out the contents of those directories.

A CI volume also has a protection status that regulates the reading, creating, purging, or renaming of the global directories on that volume.

Time Stamps

Time stamps are maintained for all files in the CI file system except those created with FMGR. The time stamps include the time of creation, time of last access, and time of last change. Times reflect both time of day and date, with a one-second resolution. Time stamps are not maintained for directories.

Time stamps are changed automatically by the system; users can only examine them with the DL command. The examples that follow illustrate the use of time stamps.

```
   CI> dl file_descriptor a          (Examine time last accessed)

   CI> dl file_descriptor c          (Examine time created)

   CI> dl file_descriptor u          (Examine time last updated)

   CI> dl file_descriptor uac        (Display all three time stamps)
```

Creation time is set when a file is created. Update time is set whenever a file is closed after being changed. Because the update time is not set until a file is closed, the update time of an open file is not accurate. Access time is set whenever a file is opened. Examining the directory information for a file does not affect the data in a file, so it does not count as an access.

When a file is created by copying an existing file, the create and last access times are changed to the time of copying. The last update time, however, remains the same as that of the existing file; this preserves the revision history of the file.

File Masks

Access to multiple files is simplified when you use a file mask, which lets you specify several fills with a single entry, using one or more of the fields in the file descriptor. For example, the daily entries of a system log can be accessed by masking the date code in the file names. Alternatively, all those files can be accessed by specifying "syslog−−−−−−".

```
SYSLOG010181
SYSLOG010281
:
:
SYSLOG123081
SYSLOG123181
```

In this case, the dash (−) is used to mask one character position (except a blank character).

The @ character is used to mask all characters. Thus, the files shown above can also be accessed with another single entry:

```
SYSLOG@
```

Some file related commands can refer to a number of files using one file descriptor with the aid of a file mask. The file mask feature uses all the fields in the file descriptor plus a special mask qualifier field. The fields used in this manner can be any or all of the following:

file name (including file type extension)
mask qualifier appended to file name
directory
subdirectory
file security code (FMGR files only)
file type
file type extension
file size
file record length
time stamps
user.group
backup status

The "." and ".." alternate directory specifiers may be used in file masks just as in regular file descriptors. However, the #n specifier may not be used.

The mask characters "−" and "@" can be used only in the file name and file type extension fields; they have no special meaning in any other fields, including directory and subdirectory. The dash masks a single character position, and the @ character masks zero or more characters.

The mask qualifier field is a string of characters appended to the file name after the file type extension. It is separated from the file name by a period. Special characters are used in the qualifier field to facilitate finding the desired files. These characters are:

Table 3-3. Mask Characters

Characters	Description
(user.group)	Mask by specified user. Return the files belonging to a given user, files in a specified group, or files belonging to a given user regardless of group.
a	Access time stamp mask (see the section on “Time Stamp Masks” in this chapter).
b	Match only those files that need to be backed up. (Refer to the discussion of the TF utility in the *RTE-6/VM Utility Programs Reference Manual*.)
c	Create time stamp mask (see the section on “Time Stamp Masks” in this chapter).
d	A search directive. If any directory matches a mask, then all files in that directory match, regardless of other characteristics.
e	A search directive. Search all the mounted disk volumes in the system, including the FMGR file system disk cartridges. (This can take a long time, depending on the size, contents, and the number of volumes.)
k	A search directive. Search from the specified directory down through any subdirectories in that directory, applying the mask to all files within the search path. K is similar to d in that it preserves the directory structure (for example, in a copy) and similar to s in that it applies the mask to each file in the path. K overrides or cancels a d qualifier.
m	Return extent entries on FMGR directories.
n	Do not match directories. Useful mostly for copying. Overrides the d qualifier.
o	Match only open files.
p	Match only purged files.
s	A search directive. Search from the specified directory down through any subdirectories in that directory, applying the mask to those files throughout the search path.
t	Match only temporary files.
u	Update the time stamp mask (see the section on “Time Stamp Masks” in this chapter).

Table 3-3. Mask Characters (continued)

Characters	Description
w	Walk through FMGR directories. FMGR directories are written on the disk in a staggered fashion. They must be accessed in the same staggered fashion to find files in the order that they appear in the directory. This is known as walking. An application in which the order of file access is not important can gain performance by accessing the directory in a faster, non-staggered manner known as running. The masking routines use the fast way unless the w qualifier is set or the buffer area supplied is 8K words, in which case no speed is gained by running.
x	Match only files with extents (not applied to FMGR files).
y	Return correct extent information on directories (requires an additional disk access for each directory).

Any of the time stamps can be used as the mask qualifier. Time stamps can be specified as an option in commands that use the file mask feature, or as a range. The format is as follows:

```
[c|a|u][xxxxxx.xxxxxx][–][xxxxxx.xxxxxx]
```

where the xxxxxx.xxxxxx is a date and time in the format YYMMDD.HHMMSS; thus, 890529.120000 is noon May 29, 1989. Only one choice among c/a/u (create, access, or update) is allowed . The default is the last update time. The dash character "–" is not a mask character when used in the qualifier field, but is used to specify a range of dates. If "–" is not used, the specification is for files that match that date/time. For example:

```
c780416.121108–810611.141411
```

This entry specifies files created between April 16, 1978 12:11:08 and June 11, 1981 14:14:11. The time can be specified with as few digits as desired. Thus "a81–83" specifies files last accessed during or after 1981 and before or during 1983.

The time stamps in the file system begin on Jan 1, 1970. Dates specified in years between 00 and 37 (inclusive) are interpreted as being the year 2000 through 2037. Time stamp values do not go beyond the year 2037.

Appropriate default values for each field are defined. If the name is not specified, all names match. The same is true for type, size, and record length. If the qualifier is not specified, all files qualify except purged files. (Note that if p is specified, only purged files qualify.) If the file type extension is not specified, only files with blank type extensions match. If the directory and subdirectory are not specified, only the working directory is used. The directory and subdirectory specification has precedence over the e option (if both are specified, only the directory is searched).

There are several special cases in specifying directories using file masks. If the mask ends with a slash (/), as in /FOO/JOE/, it is equivalent to /FOO/JOE/@.@ (default name and file type extension). This mask directs the file system to search for all files with any name and file type extension in subdirectory JOE of global directory FOO.

The trailing / is a way of referring to the contents of a directory rather than to the directory itself. To refer to the files in directory FOO, the proper mask is /foo/. Thus, to list the files in directory FOO, the command is:

```
   CI> dl /foo/
```

Note that /FOO will not list the files in directory FOO, but the following entries will:

```
   CI> dl ::foo
   CI> dl @.@::foo
   CI> dl /foo/@.@
```

If the file name in a mask ends with the wild card character (@) and the file type extension is not specified, a wild card file type extension is assumed. For example:

```
   file1@
```

is the same as file1@.@.

Files with a null file type extension can be specified with a trailing dot as follows:

```
   file2@.
```

If the mask ends with .DIR, as in /foo/joe.dir, only subdirectory JOE in global directory FOO is matched. The .DIR file type extension is needed whenever either a file or directory can be given, but can be omitted when only directories are allowed.

Examples of mask qualifiers:

`u82-`	(Updated during or since 1982)
`u82-83`	(Updated during 1982 or 1983)

Examples using the whole mask:

`@`	(Equivalent to “@.@”, specifies all files in the working directory)
`/foo/`	(Equivalent to /foo/@.@, specifies all files in directory FOO)
`/foo/@.`	(Specifies all files in directory FOO with a blank file type extension)
`/@.ftn.su82`	(Search all CI volumes for all FORTRAN source files last updated during 1982)
`/games/backgammon/source/@.@`	(All files in subdirectory SOURCE of subdirectory BACKGAMMON on directory /GAMES)
`@.dir`	(All subdirectories in the working directory; equivalent to @.dir)
`@.@.x:::4`	(Type 4 files with extents in the working directory)

Examples using the (user.group) mask qualifier:

```
   CI> dl /@.@.s(user)
or
   CI> dl /@.@.s(user.@)
```

(Return only those files belonging to user.GENERAL in the specified directory down through any subdirectories; both forms imply the GENERAL group)

```
   CI> dl @.@.(user)
```

(Return the files belonging to user.GENERAL in the working directory because this form implies the GENERAL group (will be all or none as the same user owns all files in a directory))

```
   CI> dl /@.@.s(user.)
```

(Return all files for user.GENERAL that have no associated group specified in all CI directories and subdirectories (these files were created and ownership was assigned using software prior to Rev. 5.0))

```
   CI> dl /@.@.s(@.)
```

(Return all files for which no associated group has been specified and search all CI directories and subdirectories (these files were created and ownership was assigned using software prior to Rev. 5.0))

```
   CI> dl /@.@.s(user.group)
```

(Return the files belonging to the user in specified group and search all CI directories)

Examples using search directive qualifiers:

```
   CI> co /A/B@.@.k /SCRATCH/
```

(Copy all subdirectories and files matching B@.@ within directory A to directory SCRATCH, preserving the directory structure)

```
   CI> co /A/@.@.k @
```

(Copy all subdirectories and files within directory A to the working directory, preserving the directory structure)

```
   CI> co /A/@ @
```

(Copy all subdirectories and files within directory A to the working directory, preserving the directory structure because the CO command implies use of the ‘d’ search structure)

Other examples:

```
   CI> dl @.txt
```

(Display files in the working directory with file type extension TXT)

```
   CI> dl a@.@.c83
```

(Display files in the working directory that start with a, created during 1983)

```
   CI> dl /joe/@.@.sc80-83
```

(Display files in directory JOE created during the period between 1980 and 1983. Also, the ‘s’ qualifier directs a search of any subdirectories of directory JOE for similar files)

Masking and FMGR Files

For FMGR files, the dot ".", is both a legal character and a delimiter in the mask. If necessary, the masking routines modify masks as follows for FMGR files with dots in their names before applying the masks to the FMGR files.

- The name and type extension are examined as if they referred to a CI file.
- If there is no dot in the mask, no change is made.
- If the type extension is null (that is, there is none), the dot is replaced by a "−" (minus) character. For example, "xyz." becomes "xyz−".
- If the type extension is "@", the dot is changed to "@". For example, "xyz.@" becomes "xyz@@". A CI file name represented simply as "@" (which is equivalent to "@.@") is modified to "@@@".
- Otherwise, the type extension is put back with the name. For example, "xyz.w" remains "xyz.w".

All FMGR files with a single dot in their names will be found, but additional files may also be returned. When defining masks for FMGR files with names containing multiple dots, replace all but one dot with a minus (−) character.

Destination Masks

The CO, MO, and RN commands can use a destination file mask in addition to a source file mask to give the command a framework for the destination file name. For example, the command "RN @.SRC @.FTN" renames all the files on the current directory that have file type extension .SRC to have file type extension .FTN. In general, if a name or a file type extension is specified in the destination mask, it is used for the destination file name or file type extension. If either is defaulted using the @ character, the name or file type extension of the source file is used.

The @ character must mask a complete name or file type extension. Thus the command "RN %@.REL @.REL" will NOT remove the "%" from the front of the files with type extension .REL. The .DIR file type extension cannot be changed in the destination file descriptor. If the source file type extension is .DIR, the destination type extension will be .DIR, regardless of the destination mask type extension.

The destination mask has the same rules as the source mask for implicit "@". Thus, /SOURCES/ is equivalent to /SOURCES/@.@. This results in the default name and file type extension.

For the type and record length fields, the values from the source file are always used, even if a value was specified in the destination mask. For the security code and file size fields, any value used in the destination mask overrides that of the source. The next paragraph describes how a destination directory path is generated.

The destination directory path consists of both the destination mask and the source file directory path, starting with the destination mask, to which the source directory path, less the directory path in the source mask, is appended.

The following examples illustrate destination masks used with the CO command.

Copy all files in subdirectory /PROGRAM/DOCUMENTS into subdirectory /MANUAL/DOCUMENTS.

```
   CI> co /program/documents/@ /manual/documents/@
```

In this example, the destination subdirectory must exist prior to executing the copy command. This also could have been accomplished using the following command:

```
   CI> co /program/documents.dir.d /manual/@
```

In this example, the d qualifier in the source mask specifies all files in the directory /DOCUMENTS. The subdirectory will be created if it does not exist.

Copy these files to a subdirectory called /MANUAL/CHAPTERS, changing the subdirectory name at the destination.

```
   CI> co /program/documents/@ /manual/chapters/
```

Subdirectory /MANUAL/CHAPTERS must be an existing subdirectory for the command to work. An alternate form is:

```
   CI> co /program/documents.dir /manual/chapters.dir
```

The destination subdirectory in this example will be created if it does not exist.

More examples are as follows:

```
   CI> mo main.txt subroutine.ftn   (Move MAIN.TXT to SUBROUTINE.FTN)

   CI> co main.lst @.temp           (Copy MAIN.LST to MAIN.TMP)

   CI> rn /program /pgm             (Rename directory PROGRAM to PGM)

   CI> co /pgm.dir /new/@           (Create subdirectory PGM in directory
                                    NEW with all files and subdirectories
                                    that are in directory PGM)
```

File Operations

The following sections describe the file operations you can perform using CI commands.

Directory Listings

One of the most useful file operations is listing the files in a directory, which shows the files available to you. The directory list command is DL. Entering DL without any parameters returns a list of file names in your working directory, sorted in alphabetical order. For example, to list all files in the working directory:

```
CI> dl
directory ::SMITH
A.B          D.E          TEMP.FTN          TWENTY.FTN
```

Here the working directory is SMITH, which contains the four files listed. Note that files A and D have uninformative names and non-standard file type extensions. Such names are not recommended for important data.

DL can also be used to get a list of the files contained in another directory simply by specifying the name of the directory. There are several ways to list the contents of a directory, as shown below.

To list all files in directory named JONES:

```
CI> dl ::jones            (Recommended for FMGR files)

CI> dl @.@::jones

CI> dl /jones/            (Recommended for hierarchical files)

CI> dl /jones/@.@
```

To list all files in subdirectory SUBDIR, which is in global directory JONES:

```
CI> dl /jones/subdir/

CI> dl /jones/subdir/@.@
```

This gives the names of the files contained in those directories. The trailing slash must follow the directory or subdirectory to get the desired effect.

Listing Files

The LI command lists the contents of a file to your terminal for examination. You can use a file mask to list a group of files.

To list file /SYSTEM/WELCOME.CMD:

```
   CI> li /system/welcome.cmd
```

In this example, the file is displayed on the terminal screen in pages of lines separated by the following prompt:

```
   More...
```

After the first use of the LI command your prompt may look like this:

```
   More [x%] ...
```

where x% represents the percentage of the file already listed.

You can respond to this prompt by entering a single character, optionally preceded by a number from 1 to 32767 called *n* below, to select from the following standard options:

Character	Action
<space>	List another page, or another *n* lines if given
<return>	List the remaining lines without page breaking
A or Q	Abort the listing
+	List one more line or skip *n* lines and list one more line
P	Set page size to *n* lines and list another page
Z	Suspend the LI program (restart with the system GO command)

The abort character "a" can be either upper or lowercase. After it is entered, the listing stops and the CI prompt is displayed.

Several other commands use this method of display, pausing after each screenful to let you read what has been displayed with the same choices for abort or continuation. The LI command also provides many more paging functions than the standard choices shown here. If you are using a printing terminal, you can send the command output to a file, then use the CO command to copy the file to the printing terminal without stopping. Refer to the LI command description in Chapter 5 for details of all the LI options.

If you enter a file mask, you are prompted as follows before each file is listed:

file: *filedescriptor*, `list? [Y]`

Respond to this prompt by entering a single character:

Character	Action
Y or <space>	Yes, list this file
N	Skip this file, move to the next
A or Q	Abort the LI command

LI has many options for display formatting and file filtering. See Chapter 5 for details.

Copying Files

The CO command can be used to make a copy of any type of file and to copy files to or from I/O devices.

To copy FILE1.TXT to NEWFILE1.TXT, enter:

```
   CI> co file1.txt newfile1.txt
```

The source file is given first, followed by the destination file. The source file descriptor can be masked to include a number of files. The destination file must not exist in this case; CO will not overwrite files unless directed by a replace duplicate (D) option.

The CO command creates the destination file with the same attributes as those associated with the source file. Some attributes in the destination file can be specified in the file descriptor (security code in FMGR files and file size). There is a set of optional command parameters to control the copying process. These are options provided to control the way data will be transferred and are most useful when transferring data to or from an I/O device. For more information on the CO command options, refer to the CO command description in Chapter 5 of this manual. Following are more examples of file copying entries.

Copy /SYS/REPORT1 to the working directory:

```
   CI> co /sys/report1 report1
```

Note that the destination file uses the default working directory and is not defaulted to the source file directory.

Copy a file to an existing file on the working directory:

```
   CI> co file1 masterfile d
```

D is the replace duplicate option; the current masterfile is to be purged if it exists.

Copy a file to magnetic tape (LU 8):

```
   CI> co file 8
```

Copy a file to the terminal screen (LU 1):

```
   CI> co file 1
```

Renaming Files

The RN command is used to change the name of a file (or files with the use of a file mask). It can also change the file type extension. You must have write access to the directory containing files to be renamed. To change the name FILE1.TXT to NEWFILE1.TXT, enter the following:

```
   CI> rn file1.txt newfile1.txt
```

In this example, the file FILE1.TXT will no longer exist after this operation. The new file name cannot be an existing file in this case. Refer to the RN command description in Chapter 5 for details.

Moving Files

The MO command is used to move files from one directory to another. For example, to move file FILE1.TXT::SMITH to FILE1.TXT::JONES, you could copy the file to the new destination and then purge the original file. However, if FILE1 is an enormous file, this takes a long time, and there must be enough disk space for both copies.

You can use the MO command to move the file to a different directory if both directories are on the same LU. If the directories are on different LUs, an error is returned. You must use the CO command to copy the file to the new directory and the PU command to purge the original file. Or you can use CO with the P option to purge the source after the copy. Before moving a file between directories, you can use the DL command with the L option to display the LUs on which the directories reside.

The following example moves FILE1.TXT from directory /SMITH to directory /JONES:

```
CI> mo /smith/file1.txt /jones/file1.txt
```

Purging Files

The PU command is used to purge a file, removing it from the directory. A group of files can be purged by using a file mask. You must have write access to the directory containing the files to be purged.

To remove FILE1.TXT, enter the following:

```
CI> pu file1.txt
Purging FILE1.TXT ...[ok]
```

This removes FILE1.TXT from the working directory. The disk space that FILE1.TXT occupied is now available for use by another file, but the data is still unaffected. The PU command does not destroy the contents of a file it removes. It leaves enough information so that as long as the disk area occupied by the purged file has not been overwritten, the file can be recovered with the UNPU command. This is very useful if you inadvertently purge the wrong file.

You can purge a number of files using the file mask feature. If the optional OK parameter is not specified, a prompt is displayed before each file is purged, and a Yes response is required to purge the file.

For example:

```
CI> pu file-.txt
Purging FILE2.TXT:::4:24 (Yes, No, Abort, Stop Asking) [Y] ? Y
Purging FILE3.TXT:::4:24 (Yes, No, Abort, Stop Asking) [Y] ? Y
```

If you choose the Stop Asking option, the prompt is not displayed again; only a message indicating that the file is being purged is displayed.

If the OK parameter is specified, the prompt is suppressed and only the message indicating the file is being purged is displayed. For example:

```
CI> pu fil-.txt ok
Purging FILA.TXT:::4:24 ...[ok]
Purging FILB.TXT:::4:24 ...[ok]
Purging FILC.TXT:::4:24 ...[ok]
```

Be sure that the directories containing important files are write-protected. The PU command only checks the directory protection, NOT the file protection, when purging files.

Unpurging Files

The UNPU command is used to restore a purged file (or files), usually immediately after the error occurs. It is effective as long as the purged file has not been overwritten.

To restore file FILE1.TXT that was purged earlier, enter:

```
CI> unpu file1.txt
```

There is no particular limit to the length of time that a purged file remains recoverable. It depends on such random factors as the number of files being created and the position of the file on the disk and in the directory. Unpurging should be used immediately after an erroneous purge command. If the command returns an error message indicating that the file cannot be recovered, the file has been overwritten.

In program development, there may be several purged files with the same name. This can happen through sequences of create and purge operations, but it is relatively uncommon. You can unpurge all of the files of the same name successively by unpurging and renaming them one at a time.

Creating Empty Files

Empty files can be created with the CR command. The file space specified for these files is reserved as soon as each file is created. The CR command cannot be used to overwrite an existing file.

To create a file called FILE1.TXT:

```
CI> cr file1.txt
```

You can specify various file attributes: type, size, and record length. The following examples illustrate creating empty files with these attributes.

```
CI> cr file.dat::system:1            (Create a type 1 file)

CI> cr /system/subdir/file.txt:::1   (Create a type 1 file in a subdirectory)

CI> cr file.mnl:::1                  (Create a type 1 file in working directory)

CI> cr /system/bigger.dat::100       (Create a file of 100 blocks)
```

Changing File Protection

The protection status of files can be displayed with the PROT command. Protection status of a file can only be changed by the owner of the directory containing the file or by the System Manager.

To display the protection status of a file in the working directory:

```
   CI> prot file.txt
   directory ::DOUG
         name     prot
                                         (File is write protected from members of
   FILE.TXT       rw/r/r                 the associated group and general users)
```

The protection status is given in abbreviations, W for write access and R for read access. The owner status is given first, followed by a slash, then the status for members of the associated group, followed by a slash, then the general user status.

Most files are usually assigned read access for general users and members of the associated group, and read and write access for owners. To reassign the protection status, refer to the following examples.

```
   CI> prot report rw/                   (Read and write allowed for owner only)

   CI> prot receipts rw/r                (Read/write for owner; read for others)

   CI> prot testdata.txt rw/rw/r         (Read/write for owner and group; read
                                         for others)

   CI> prot memo rw/r/                   (Read/write for owner; read for group;
                                         no access for others)
```

To change the protection for all files in a directory, follow the example shown below.

```
   CI> prot /data/ rw/rw/rw           (Read/write access to everyone for
or                                    all existing files in directory DATA.)
   CI> prot ::data rw/rw/rw
```

In this example, all existing files in directory DATA are allowed both read and write access. Note that the protection for directory DATA has not changed. All files to be created in that directory still follow the directory protection status.

Manipulating Directories

Directories can be thought of as system files with which only the operating system is concerned. Each directory contains information about the files that are in the directory, although the data in the file itself is not in the directory. File data is kept elsewhere on the disk volume. (Volumes are described separately in this chapter.) Directories have an initial size, and they are automatically extended to hold more files as necessary. When files are purged, the directories are not truncated; the space previously occupied by the purged files is reused when new files are added.

Subdirectories can appear in other directories in much the same way that any other file does. Directory and subdirectory names always have file type extension .DIR to distinguish them as directories; no other file can have a type extension of .DIR. There is usually no need to specify the .DIR file type extension when dealing with directories because it is implied by the way the name is used. For example, .DIR is not needed in the name /MAIN/SUBDIR/FILE, nor is it needed in the WD (working directory) command. (The entry /MAIN.DIR/SUBDIR.DIR is not valid, as the file type extension .DIR cannot be used in front of a slash.)

Operations involving directories include creating directories, changing the working directory, listing directories, renaming directories, purging directories, and examining and changing directory ownership and protection. These are all discussed in the following sections.

Creating a Directory

Directories are created with the CRDIR command. To create a global directory named SYSTEM, enter the following:

```
   CI> crdir /system
```

This entry creates a global directory SYSTEM on the same disk volume as the working directory. If there is no working directory or if you want to place SYSTEM on a different disk volume, enter the following:

```
   CI> crdir /system 12
```

This creates directory SYSTEM on disk volume LU 12. To find out what disk volumes are available, use the CL command. In this example, since you entered the command, you become the owner of directory SYSTEM. Other users are not allowed to create another directory of the same name. This directory is a global directory with the initial default protection status. Global directories have the following default protection status:

`RW/R/R`	(Read and write allowed for owner and read only for other users)

All subsequent files managed in directory SYSTEM have the same protection status unless changed by the PROT command, either for the directory or individual files.

Note that the following command does not create a global directory; rather, it creates a subdirectory called SYSTEM in the current working directory.

```
   CI> crdir system
```

Creating a Subdirectory

Creating a subdirectory is similar to creating a directory. To create a subdirectory of directory SYSTEM called SUBDIR, enter the following:

```
CI> crdir /system/subdir
```

This creates the subdirectory SUBDIR in global directory SYSTEM. Note that the .DIR file type extension is not necessary. The subdirectory protection is set to that of the directory in which it is being created. If this is a global directory, protection is set to RW/R/R. The user who creates the subdirectory becomes its owner, even if it is a subdirectory of a directory that the user does not own (but has write access to).

The difference between specifying subdirectories and directories is that a leading slash is used for a global directory, while none is used for a subdirectory.

Display/Setup Working Directory

The working directory is searched first by the file system when it searches for files. It is the directory used if a file is specified without any directory name. The working directory can be a subdirectory.

To examine the name of the working directory, use the WD command without any parameter. For example

```
CI> wd
Working directory is ::DOUG
```

To set up a working directory or to reassign another directory as the working directory, enter the WD command with the name of the directory (or subdirectory). For example:

```
CI> wd games
```

Here GAMES, a subdirectory in the current working directory, becomes the working directory.

```
CI> wd /games/rules
```

In this case, subdirectory RULES is a working directory.

If there is no need for any working directory, specify 0 as follows:

```
CI> wd 0
```

The effect of this command is that the first FMGR disk on the cartridge list is used if the directory or CRN (in FMGR files) is omitted.

The WD command can also be used to post the contents of the command stack to the associated file or to change this file. The default command file is CI.STK on the working directory. To post the contents of the stack to the default command stack file, enter:

```
CI> wd,,+s
```

CI.STK is opened, and the command stack is posted there.

This can also be done as you change the working directory. For example:

```
   CI> wd /debbie +s
```

The above command causes the file associated with the command stack to be posted with the contents of the stack. Then the command stack is overwritten with the contents of /DEBBIE/CI.STK or cleared if the file does not exist. You can specify any file to be associated with the command stack. For example:

```
   CI>  wd,,Cmdstackfile.stk
```

In this case, the command stack is posted to the current command stack file if one exists; if it does not exist, it is created. Then the contents of Cmdstackfile.stk are placed into the command stack. If this file does not exist, the command stack is cleared. At log off, this file is posted with the contents of the command stack and closed.

Moving Directories

The directory path of a file or subdirectory can be changed by the MO command, which is especially powerful in manipulating directories. MO can be used to move all files in one directory to another. For example, to change subdirectory /SYSTEM/SUBDIR into a new global directory NEWDIR, enter the following:

```
   CI> mo /system/subdir.dir /newdir
```

(Move SUBDIR into the global directory table and rename it to NEWDIR.)

This changes the way you refer to all of the files in the directory as well; they must be preceded by /NEWDIR instead of /SYSTEM/SUBDIR. Directories do not have to be empty to be moved.

Displaying Directory Owner

The owner of a directory can be displayed with the OWNER command. This can be done by all users of the system. To display the owner of a directory named SYSTEM, enter the following:

```
   CI> owner /system
   Owner of /SYSTEM is DOUG
```

Changing Directory Owner and Associated Group

The owner and associated group of a directory can be changed with the OWNER command. This can only be done by the current owner or by the System Manager. Assuming that you created directory SYSTEM, to change its owner to JONES and associated group to LAB, enter the following:

```
   CI> owner /system jones.lab
```

Use this command with caution. Once the ownership is changed, you are no longer the owner and may not have the same protection status. You may not be able to write (or read/write) into the directory, and you cannot revert the ownership. From this point on, only JONES.LAB can change the ownership and associated group. The subdirectories within directory SYSTEM are not affected.

If you do not specify an associated group in the OWNER command, the associated group becomes the default logon group of the user who is the owner. For example, assume the group LAB is the default logon for JONES. The following command

```
CI> owner /system jones
```

changes the owner and associated group of the directory SYSTEM to JONES and LAB, respectively.

If you do not want a directory to have an associated group, you can specify NOGROUP. This also turns off group protection in the directory, as shown in the example that follows.

```
CI> owner /temp
Owner of /TEMP is JONES.LAB        (An associated group is defined)

CI> prot /temp directory /         (Group protection is on)
name          prot
TEMP.DIR      rw/r/r

CI> owner /temp jones.nogroup      (Making no associated group)

CI> owner /temp
Owner of /TEMP is JONES            (No associated group is defined)

CI> prot /temp directory /
name          prot
TEMP./DIR     rw/r

CI> owner /temp jones.accounting   (Defining an associated group)

CI> owner /temp
Owner of /TEMP is JONES.ACCOUNTING

CI> prot /temp directory /         (Group protection is on again)
name          prot
TEMP.DIR      rw/r/r
```

Purging Directories

Directories and subdirectories can only be purged by the owner when they are empty. All files must be purged or moved to another directory before purging the directory. Directories cannot be unpurged.

To purge a directory named GAMES:

```
   CI> pu /games
```

Note that the form ::GAMES cannot be used because this is interpreted by PU as all files in directory GAMES. PU will purge them all if there are files in directory GAMES. If not, the message “Directory is empty ::GAMES” is displayed. You must precede the directory specification with a slash.

To purge a subdirectory called SUB.DIR under directory SYSTEM:

```
   CI> pu /system/sub.dir
```

Displaying/Changing Directory Protection

The protection status of a directory or subdirectory can be displayed with the PROT command. Only the owner can change the protection status of a directory or subdirectory.

The following examples show displaying protection status:

```
   CI> prot /system                 (For directory SYSTEM)

   CI> prot /system/                (For all files in directory SYSTEM)

   CI> prot /system/data.dir        (For subdirectory DATA)

   CI> prot /system/data/           (For all files in subdirectory DATA)
```

To change the protection for a directory (SYSTEM):

```
   CI> prot /system rw/rw/rw        (Read/write access for everyone)
```

To change the protection for a subdirectory (DATA):

```
   CI> prot /system/data.dir rw//   (Read/write access for owner only;
                                    read/write protected from others)
```

Searching for Files

When you enter a file-referencing CI command, CI checks to see if you specified a directory; if so, CI searches that directory for the file and returns an error if the file is not found.

If you do not supply directory information, CI attempts to locate the file. For all file-referencing commands except RU and TR, CI searches your current working directory or all mounted FMGR cartridges if you do not have a working directory. An error is returned if the file is not found.

When searching for files specified in the RU and TR commands, CI follows special default search sequences. By defining UDSPs #1 and #2, you can change the default search sequences for the RU and TR commands, respectively.

Default Search Sequence

If you do not include directory information with a RU or TR command (implied or explicit), the following search sequence is used to locate the file:

- The current working directory is searched. If the file is not found, a default type extension of .RUN or .CMD is assumed and the working directory is searched again.
- If you do not have a working directory, all mounted FMGR cartridges are searched.
- If the file is still not found, global directory PROGRAMS or CMDFILES is searched using the .RUN or .CMD default file type extension, respectively.

Defining UDSPs

User-Definable Directory Search Paths (UDSPs) allow you to change the default search sequence used to find command and program files. The RU command uses UDSP #1 and the TR command uses UDSP #2.

For example, suppose you want CI to search the following directories when searching for a command file:

- Current working directory
- /JONES/UTILITIES/CMDS
- /CMDFILES

The following PATH command defines UDSP #2 to use this search sequence:

```
CI> path 2 . /jones/utilities/cmds /cmdfiles
```

The period (.) indicates that your working directory (at the time the TR command is entered) is to be searched for the file.

To display the contents of UDSP #2, enter the following:

```
CI> path 2
UDSP #2:    /JONES/STUFF [current WD]
            /JONES/UTILITIES/CMDS
            /CMDFILES
```

The first directory displayed, /JONES/STUFF, is the name of the working directory at the time you entered the PATH command to display UDSP #2.

UDSP #1, which is used by the RU command, can be set to use a different search pattern. Assume you want the RU command to use the following search sequence:

- /MINE/PROGRAMS
- Current working directory
- /MINE/MORE/PROGS
- /PROGRAMS

The following PATH command sets UDSP #1 to this sequence:

```
CI> path 1 /MINE/PROGRAMS . /MINE/MORE/PROGS /PROGRAMS
```

Refer to the description of the PATH command in Chapter 5, “CI Command Description,” for more details.

Specifying UDSPs in File Descriptors

Just as UDSPs #1 and #2 are used by the TR and RU by default, it is possible to explicitly request a UDSP search sequence for a particular file. For example:

```
CI> li #2/comp.cmd
```

requests the LI command to search through UDSP #2 to find COMP.CMD. This lists the same file that the RU command finds.

The sequence #N/*file* may be used with the FMP Open routines and with any command that opens the file, except where a file mask is required (such as with DL).

Manipulating Volumes

Each physical drive consists of one or more volumes. A volume is a self-contained section of a disk, independent of any other volume. A volume is always identified by its disk LU number, from 1 to 63. Volumes never cross physical drives, and files and directories never cross volumes.

Each volume contains a unique set of information about the files on it, including the names of all the global directories on the disk, as well as a table that tells which disk blocks have been allocated to files. This table is called a bit map, because the table is composed of bits rather than addresses or values.

Common operations performed are: mounting a volume, dismounting a volume, changing ownership and protection, and listing contents of a volume. An operation that is infrequently performed is initializing a volume, making it ready for system use.

Mount/Dismount Volumes

Mounting a volume makes that volume and all the files on it available to the file system. Dismounting a volume removes that volume and makes the files on it inaccessible to the system. These operations are not performed frequently except with removable media such as floppy disks, that must be mounted after they are installed and dismounted before they are removed.

For example, to mount a volume with disk LU number 12:

```
   CI> mc 12
```

If the disk volume has a valid FMP or FMGR directory, the volume is mounted. If the disk volume does not have a valid FMP or FMGR directory, you are prompted to confirm that the volume should be initialized. This is done to avoid the accidental corruption of volumes that are not FMP or FMGR types (for example, backup utility volumes).

Initializing a volume sets up information needed by the operating system, including the list of directories and the bit map for keeping track of disk space usage.

When you mount a volume there is a chance that directory names on the volume just mounted will conflict with directory names on already mounted volumes. If duplicates occur, the names of the duplicate directories are displayed. If you need the new ones, you can rename the duplicate directories already mounted, then dismount and remount the volume.

For example, to dismount volume LU 12:

```
   CI> dc 12
```

Before you dismount a volume, make sure there are no open files, working directories, restored programs, or directories that are part of a current user's UDSPs on that disk volume. Otherwise, when you try to dismount the volume you will get an error message each time an error is encountered and the dismount command will be aborted.

The DC command shows only one error at a time, which means that you must repeat the command until all the errors are found. You must identify and correct all the errors separately before the dismount command can be completed. For example, the following commands can be used to check for conditions that can prevent dismounting disk LU 12:

```
CI> wh,al               (Check all RP'd programs)

CI> dl 12 o             (Check for opened files. This lists all files on LU 12,
                        which can take a long time if there is a large number of
                        directories and files.)

CI> wd 0                (Remove the working directory. This command must be
                        used for each user who has a working directory on that
                        LU in a multiuser environment.)

CI> whosd dir/12        (Check for any session accessing a specified directory or
                        a directory on LU 12 as a working directory or as part of
                        a UDSP)
```

Volume Ownership and Protection

Volumes have owners and protection as do directories. The original owner of a volume is the user who initialized it. The protection of a volume governs the accessibility of global directories on that volume.

Ownership is displayed and changed with the OWNER command. For example:

```
CI> owner 12v           (Display the owner of volume 12)

CI> owner 15v fred.lab  (Change the owner of volume 15 to FRED.LAB
                        and the associated group to LAB)
```

Protection is displayed and changed with the PROT command. Only the owner of a volume can change the protection. For example:

```
CI> prot 10v            (Display protection of volume 10)

CI> prot 13v RW/R/R     (Only the owner may create global
                        directories on volume 13)
```

Listing Volumes

The CL command is used to list the volumes that are currently mounted. The CL command has no parameters. It provides a list of two types of volumes: those mounted as described above and those mounted as FMGR cartridges (as discussed in the section on “FMGR Files” in this chapter). Unmounted volumes are not listed.

```
CI> cl
File System Disk LUs: 19 17
FMGR Disk LUs (CRN): 16(16) 20(A2)
```

Initializing Volumes

Initializing a volume prepares it for first-time system use. This function is done automatically by the MC command, but the IN command can also be used on a mounted volume. The IN command can be used to remove all the data on a volume without having to purge all the files first. Initializing a disk volume permanently destroys any existing files, so be certain that the files on that LU are no longer needed. In a VC+ environment, this command can be used only by the superuser.

For example, to remove all data on volume 12 without dismounting it, enter the following:

```
CI> in 12
Re-initialize valid directory [N]? y
Initializing Disk
CI>
```

Transferring Data to and from Devices

You can send data to and from an I/O device instead of a file by replacing the file descriptor with the LU number of the I/O device. Devices that can be used include printers, terminals, magnetic tape units, and HP-IB devices. This method of data transfer should never be used with disks, CTDs, and Distributed System (DS) network links.

The CI commands that transfer data to or from files are CO and LI. Other CI file-referencing commands that do not deal with the data in files (for example, RN and PU) cannot be used for this purpose.

The CO command can include an LU as either the source or destination LU, or both. When an I/O device is specified as a source, the CO command moves data until the device sends an end-of-file mark. On a magnetic tape, there is an end-of-file mark; on a terminal, data is terminated with a control-D character (control key CTRL and D typed at the same time). For example:

```
   CI> co 1 newfile.txt
```

The CO command puts everything you type into the file until a CNTL-D is entered.

The CO command is also used to send data to an I/O device. File masking can be used to send several source files with each CO command. These are sent to the device one file at a time, so they are written sequentially. Note that both the source and destination parameters in the CO command can be LU numbers representing different I/O devices. You can even copy from the terminal keyboard to the display by entering:

```
   CI> co 1 1
```

The LI command can also be used to list information from an I/O device. The same rules apply as when you use the CO command with I/O devices.

The file system does some special processing depending on what type of device you are using. Some devices must be used by one user at a time to get good results; for example, you cannot have line printers or magnetic tapes with output from several different programs being interleaved. The system locks the LU to the program using it to prevent access by other programs. If another program already has the LU locked, the second program waits until the LU becomes available. Terminals are not locked so that messages can still get through to them.

Other special processing ensures that the data is transferred to or from the LU in the proper format. The system recognizes printers, magnetic tapes, and terminals, and it does special processing required for them. For most devices, data transfer is not a problem. If you have special devices, a special program must be written for the computer-device interface.

In addition to the CI commands, most programs that use files accept an I/O device LU number as a file. For example, EDIT can list part of a file to an I/O device. However, there are times when a program expects a disk file and in this case an LU number will not be accepted. This may occur because the program must read the data twice or because it must refer to the directory information for the file. I/O devices do not have file directory information.

FMGR Files

The following paragraphs present an overview of how files are handled by the FMGR program. FMGR should be used only for existing files created with FMGR. When using FMGR files in the CI file system, note the differences between the two types of files.

The main characteristics of FMGR files and the difference between FMGR and normal files are:

- File names are limited to six characters.
- Directory names are limited to two characters or can be a number.
- Subdirectories are not allowed.
- There can only be one directory per volume; the volume and directory are known collectively as a cartridge.
- There are no file type extensions, time stamps, or owners.
- Protection is included as part of the file descriptor in the form of a security code parameter.
- There is no facility available to unpurge a file.
- The following characters are not allowed in FMGR file names:

```
    + − : ,
```

CI file referencing commands can be used to a limited extent for FMGR files. For example, with the proper parameters, DL can be used to list a FMGR disk directory, and CO will copy files to and from a FMGR disk directory. Other commands that can be used with FMGR are MC, DC, CL, LI, PU, and RN. It is not possible to set your working directory as a FMGR directory, but you can set it to zero:

```
CI> wd 0
```

This indicates that you have no working directory. When you have no working directory, the file system searches for a file specified with no directory name by searching all of the FMGR cartridges in the order they are mounted (as reported by CL).

Although CI can handle FMGR files, note the following cases:

- Names with slashes cannot be used.
- Names with dots or ending with dots are not acceptable, except for a single dot in character position 2, 3, 4, or 5.
- The “at” sign (@) is interpreted as a wildcard character in CI commands, although a FMGR file name containing @ will eventually be selected.

We recommend that you rename such files. Otherwise, only FMGR can be used to access them. If CI commands are used for FMGR files, they must observe the FMGR restrictions given here.

Some CI commands do not work with FMGR files because of the differences between FMGR and CI directory information. For example, you cannot change the protection status of an FMGR file with the CI PROT command. In addition, FMGR is the only program that can initialize or pack an FMGR directory.

DS File Access (DS Only)

Systems that use the DS/1000-IV Distributed System Network can access files located on other RTE systems within the DS network. This includes FMGR files located on other RTE systems connected to your system. The same operations used to access files on any local system can be used to access files in the DS network. Local system (or local node) means your system, and remote system (or remote node) means any other system connected to your system via the DS network. If your system does not use the DS/1000-IV Distributed System Network, skip the following paragraphs.

Specifying Remote Files

DS transparency software is used to access files at remote systems. Files and directories in a remote system can be listed and copied to and from your system. Wildcard characters can be used in the file name parameter, and file masks can be used in the file descriptor. You can specify a remote file as an input to programs such as LINK, EDIT, or other utility programs.

To specify a file located in a remote system, the node number or name of the remote system is included in the file name. Each system has a node number; these numbers are explained in the DS manuals. Each system can also be assigned a node name; these are kept in a file called NODENAMES in the SYSTEM directory. This file is used to associate node names with node numbers. The DS software uses it to build a table of names for node numbers. The NODENAMES file contains entries of the form:

`*` *comment*

or

node# *nodename* [*comment*]

As an example:

```
* Test System 1  (comment line)
1 SYS1
* Test System 2
2 SYS2
* Central Systems
3 Central1
4 Central2
```

Specify the node number (or name) by appending it to the file descriptor, separated by a ">" sign; for example:

```
/Directory/File>Nodename
```

or

```
File::Directory>Nodename
```

This specifies a file located at the node named Nodename. The > sign must follow all other file information, including type, size, and record length. Note that the nodename delimiter is the > sign, and it is a valid FMGR file name character. Any FMGR file name with the > sign anywhere except the first character cannot be accessed. For example, the name >FILE can be used in a file specification, but A>FILE is interpreted as file A in the remote system named FILE.

Remote File Access

If the remote system operates in the session environment, the appropriate account logon entry can be included in the remote file specification. The account name and password, if one is required, are specified within square brackets; for example, [USER]. The trailing bracket is optional but is recommended for clarity. The account delimiter ([) cannot be used in a FMGR file name except as the first character. To specify a file at node 27 in the session environment:

```
   /directory/file[user]>27
```

or

```
   /directory/file>27[user]
```

If the USER account has a password, you must enter it; use a slash as a delimiter:

```
   /directory/file[User/Password]>27
```

Note that the password will be displayed on your terminal screen. If you enter the wrong password or log on without it, an error message will be displayed:

```
   Incorrect password
```

Upon successful logon, you can access all files available in that account and under the same restrictions applicable to that account. You will remain logged on during the time that the file is open, and you will be logged off at the remote node when the file is closed.

Files within the DS network can be transferred to and from any two nodes, local-remote or remote-remote. When transferring files from one remote system to another, two logon entries and two nodes are required for the source and destination system. The node specification for a local system can be omitted. File masks can be used.

```
   CI> co /mydir/@.ftn>systemA[UserA] /dir/@>systemB[UserB]
```

This example copies all FORTRAN source files from a directory in SYSTEMA to a directory in SYSTEMB. This sample entry is valid as long as the systems specified (and your system) are actively connected in the DS network and the file system access rules are observed. If you are at either SYSTEMA or SYSTEMB, the local node name can be omitted:

```
   CI> co /mydir/@.ftn /dir/@>systemB[UserB]
```

or

```
   CI> co /mydir/@.ftn>systemA[UserA] /dir/
```

DS File Access Considerations

In accessing remote files through the DS network, keep in mind the following considerations.

FMGR files are accessible unless the file name contains one or more characters that have special meaning, such as > or [. The DS transparency software operates from CI and other programs that use the CI file system. If your system operates strictly with FMGR, refer to the DS manuals for all DS operations.

It is legal and useful to specify the local system in the node specification. For example, this allows you to move a file from another account on your local system. If an account name is specified without a node, the local system is assumed.

Some file names may begin with a greater-than sign (>). For example, the entry “dl /dir/>27” does not specify a remote file. To specify a remote file, use:

```
   CI> dl /dir/@>27
```

If a system failure such as power failure occurs while remote files are being accessed, note the following:

- If the remote system is down, requests to it will time out causing an error return from the FMP call making the request.
- If the remote system goes down and comes back up immediately, files that were open on that system will no longer be open, though it may appear that they are at the local end. Accesses to such files may get errors. Use the CLOSE utility described below to close these files.
- If the local system goes down, its files will be left open at the remote system. DS transparency software Rev. 5000 and later will attempt to close them the first time the local system sends a request to the remote system after it is rebooted. Versions earlier than Rev. 5000 will not close files that were left open. The recommended way to close these files is to use the CLOSE utility described below.

To close open files while accessing remote files:

```
   CI> close /directory/file                (At local node)
```

or

```
   CI> close /directory/file>node           (At local or remote node)
```

This sample entry closes a file if it is open to the DS transparency monitor TRFAS. You must specify a logon name for the local file if one was supplied when it was opened, even if the file is in your local node. CLOSE must be given the full file descriptor of the file to be closed, including the path and DS information.

Remote File Access Limitations

In some cases, CI file manipulation commands cannot be used on a remote system. The most common cases are:

- The default working directory cannot be used at a remote system.
- A program contained in a remote file cannot be run. (However, you can copy the file to the local system and then run the program.)
- Volumes at remote systems cannot be mounted or dismounted.
- Ownership of directories cannot be examined at a remote system.
- I/O devices (such as terminals or printers) at a remote system cannot be accessed.

4

Controlling Programs

This chapter explains how to use the CI commands for controlling programs. You can manipulate programs in several ways: restore them into system tables, remove them from system tables, stop programs momentarily or completely, resume execution of a suspended program, and modify the memory requirements.

A brief summary of the program control commands is shown in Table 4-1. Refer to Chapter 5 of this manual or to the *RTE-6/VM Terminal User's Reference Manual*, part number 92084-90004, for a full explanation of these commands.

Table 4-1. Program Control Commands

Command	Task
`AS` *prog part#*	Assign partition
`BR [`*prog*`]`	Break program execution
`GO [`*prog* `[`*pram*5*`]]`	Resume suspended program
`IT` *prog*`[`*res* `[`*mpt* `[`*hr m sec ms*`]]]`	Set execution time
`OF [`*prog* `[ID]]`	Remove program
`PR` *prog* `[`*priority*`]`	Display/modify program priority
`RP` *file* `[`*prog*`]`	Restore program
`[RU]` *prog* `[`*pram*5*`]`	Run program with wait
`SS [`*prog*`]`	Suspend program
`ST [`*prog*`|`*part#*`|0]`	Display program status
`SZ` *prog* `[`*size* `[`*mseg_size*`]]`	Display/specify program size
`VS` *prog* `[`*vsSize*`]`	Display/modify virtual EMA size
`WS` *prog* `[`*wsSize*`]`	Display/modify VMA working set size
`XQ` *prog* `[`*pram*5*`]`	Run program without wait
`? [`*command*`]`	Online help

Program Identification

RTE-6/VM provides many different programs to support a variety of tasks. These programs can be run from CI. Programs are scheduled by name, along with a program runstring that may include program parameters. The program name consists of up to five characters and must begin with a letter. If a program file with a file name of more than five characters is specified in the run command, only the first five characters are used as the program name.

RTE manages program execution by identification (ID) segments. Before a program can be executed, it must be assigned an ID segment, which identifies the program and the location of its associated program file and maintains information such as program size, status and priority. The ID segment may be released at the end of program execution, or it can be established permanently with the RP command and removed with the OF command.

Program Priorities

Each program has an assigned priority, an attribute that indicates the program's importance. When you schedule a program for execution, the system may not execute your program immediately, depending on its priority in relation to that of other scheduled programs.

Program priority is in the range of 1 to 32767, lower numbers indicating higher priorities. If two programs are scheduled to run at the same time, the higher priority program will be run first. In addition, programs with equal priorities may be timesliced to appear to run concurrently. Program priorities can be changed interactively, as explained later in this chapter.

Running a Program

A program may be run from CI by using the RU command. For example, to run the editor program (EDIT), enter:

```
   CI> ru edit
```

In CI, RU is an implied command, which means it is not necessary in the command runstring. Therefore, the editor may also be run by entering:

```
   CI> edit
```

Any time a non-CI command is entered, CI first checks the $RU_FIRST predefined variable. If the variable is set to TRUE, CI assumes that the RU command was intended and attempts to execute the file. If you will be executing more programs than command files, you should set the $RU_FIRST variable to TRUE.

As you run the editor program, you may want to specify a file to be edited. The editor was written to accept a file name parameter in the runstring. For example:

```
   CI> edit prog.ftn
```

Program EDIT will also accept, as a second parameter, a command to be entered after opening the file. The entry

```
   CI> edit prog.ftn s
```

runs the editor, opens file PROG.FTN, and executes the editor S command (enter screen mode).

Parameters are accepted by other programs such as LINK, FTN7X, and MACRO. These are described in their respective manuals. User programs may be written to accept up to five numeric parameters from the runstring and a character string. This facility is described in the *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005.

Program Execution

Upon receipt of an RU command, the system searches for an existing ID segment for the program specified or creates one for that program. Then the program is scheduled to run by having its ID segment placed in a list of programs ready to execute. The system dispatches programs from this list in order of their priority.

Program CI is suspended to allow interaction between the program and your terminal. When the program terminates, CI again issues its prompt and accepts commands. This cycle is known as “run with wait”.

Sometimes it is desirable to let a program run while continuing CI interaction. This may be true for lengthy programs that require no user interaction. The XQ command will schedule a program to run and return control to CI. Its use is described in the following section.

Running Programs with Wait

To start a program with wait, enter the program name after the CI prompt.

```
   CI> edit
```

Program CI first checks that this is not a CI command. If the program name is the same as a CI command, precede the program name with the CI run command, RU. For example, to run a program called OWNER, enter:

```
   CI> ru owner
```

In this example, if the program was restored (that is, was assigned an ID segment), CI executes it. After the program terminates, it remains restored. If the program was not restored, CI restores and executes the program, and releases its program ID segment at completion. Note that the program’s ID segment will be duplicated if it can be cloned. The duplicate version will be released at termination. Programs that terminate saving resources should not be duplicated. To avoid duplication, link the program with the DC (don’t clone) option or run the program with the :IH option. See EXEC 6 in the *RTE-6/VM Programmer’s Reference Manual* for details on saving resources termination.

Special processing occurs when a program file needs to be restored. When CI looks for a program file, it uses the name and directory specified. If only the program name is specified, CI first searches for a restored program, then for a file in the working directory, and finally for a file in a system global directory called PROGRAMS.

The following sequence illustrates how CI searches for programs.

1. When the command EDIT is entered, it is examined by CI and interpreted as a program since it is not a CI command. CI searches for an ID segment restored for EDIT. If one is found, CI runs EDIT, using that ID segment.

2. If there is no restored EDIT, CI scans the working directory for a file named EDIT or EDIT.RUN. If one is found, CI allocates an ID segment for that program file and executes it. If EDIT is still not found, CI searches for EDIT.RUN in directory PROGRAMS.

3. If there is no working directory (such as after a “wd 0” command), CI scans all FMGR cartridges in the same way FMGR searches for files. If unsuccessful, CI then searches for /PROGRAMS/EDIT.RUN.

The above program search sequences apply to the RU, XQ, IT, and RP commands as well as to scheduling operations done by other system programs such as EDIT and LOGON. The program search sequence does not apply to other CI commands. For example, entering “li edit.run” will not find EDIT.RUN unless it is in the working directory. You must specify the directory (or FMGR cartridge) where EDIT.RUN resides.

Specifying the directory/subdirectories allows CI to skip the search sequence and proceed directly to the file. Entering /DIRNAME/EDIT allows CI to find EDIT quickly in directory DIRNAME.

One way to make use of the default program search sequence is in program development. Because your working directory is searched first, you can have your own version of any program in the working directory, leaving the unmodified version in directory PROGRAMS where it is accessible to other users.

The system will handle cases where there are two or more programs scheduled with the same name. This can happen in two situations: several copies of a program may be running at the same time (for example, EDIT may be run by several users); or shortening two different file names may lead to the same 5-character program names (for example, DATALATCH.RUN and DATALOGGER.RUN). The system handles these situations by changing the names of the duplicate programs, replacing characters four and five of the program name with the session number. For example, the second copy of EDIT becomes EDI77. If the name is still a duplicate, then characters three and four are replaced with the session number and character five is A, B, C, and so on. For example, the third copy of EDIT becomes ED77A, the fourth copy ED77B, and so on.

Running Programs without Wait

To run programs without wait, the XQ command is used. XQ starts the program specified and returns control to you, indicated by the CI program prompt. For example:

```
CI> xq prog/file       (Scheduling PROG/file without wait)
CI>                    (PROG/file executing; CI back in interpretive mode)
```

The XQ command is not recommended for use with interactive programs. It is best used for programs that take a long time to run and do not require any user intervention.

You can run several programs at the same time using XQ. This command works the same way as the RU command, including restoring the program and changing the name if necessary. If you try to start a program that is already running with XQ, a message is displayed to report that the program is busy. CI returns to the interactive state with the CI> prompt. To run a program without wait:

```
CI> xq /testdata/subharmonics.run
CI>
```

Any errors reported during program execution are displayed at the terminal along with any completion message. The WH command can be used to check the status of the program scheduled with the XQ command. Refer to Chapter 2 in this manual or to the *RTE-6/VM Utility Programs Reference Manual*, part number 92084-90007, for a full description of the WH command.

Time Scheduling Programs

To schedule a program to start at a later time (up to 24 hours), the IT (Interval Timer) command is used. IT sets up a program to run at a particular time based on the processor time-of-day clock after being scheduled by the ON command. ON operates in the same way as the XQ command, except for the time delay. For example, to run program CLOCK with parameters A, B, and C at noon:

```
CI> it clock,1,,12
CI> on clock a,b,c
```

CI returns control to you after this command. At 12:00, program CLOCK runs once. The IT command handles ID segments and program naming in the same way as the RU and XQ commands.

The time must be specified in 24-hour format; 1:30 pm is 13:30. Minutes and seconds are optional. The maximum time delay is 24 hours. If at 4:05 pm you specify the program start time as 4 pm, the program runs at 4:00 pm tomorrow, rather than immediately.

You can also use IT to start a program and subsequently run that program repeatedly at some time interval. To run program CLOCK at one-hour intervals in the above example:

```
CI> it clock 4 1 12 30
CI> on clock a b c
```

Note that the a, b, and c parameters are passed to CLOCK on the first run only.

The time interval for repeated execution can be specified as hours, minutes or seconds, or tens of milliseconds; the first parameter is 4, 3, 2, and 1, respectively. The next parameter is a multiplier, for example, one hour would be specified as “4 1” and 30 minutes, “3 30”.

To remove program CLOCK from the time list, enter the following:

```
CI> it clock
```

The IT command syntax is shown in Chapter 5. Refer to the *RTE-6/VM Terminal User’s Manual* for a complete description of the IT and ON commands.

Restoring Programs

Typically, program restoring is a process that assigns to the program file an ID segment in a system table that keeps track of programs to be executed. The ID segment contains information necessary to run the program: the 5-character program name, its location on disk, scheduled run time, priority, partition assignments, and other information required by the operating system. CI commands that affect these program attributes cannot be used until the program is restored.

A program can be restored in one of two ways. The most common method is to implicitly restore the program through the use of an RU, XQ, or IT command where no ID segment has been allocated for that program. The ID segment is released upon program termination. A second method is to explicitly restore the program with the RP command. The program is allocated an ID segment, but is not scheduled for execution. The ID segment is permanently assigned until removed with the OF command.

For example, to restore program TEST.RUN, enter:

```
   CI> rp test.run
```

The program can now be run using the RU, XQ, or IT command, and the ID segment will remain allocated upon program termination.

If a second user tries to run a program that was previously linked using the DC (do not clone) option and restored with the RP command, the system issues an error message indicating that the program is busy. The second user can either wait for the program to finish or use a second parameter in the RP command to create another ID segment with a new program name:

```
   CI> rp test.run test2
```

The second user can now use the RU, XQ, or IT command with program TEST2. Note that the second program name must be five characters or less. This method is not required for programs that were previously restored implicitly, because the system will automatically create a new name in this case (described in the “Running Programs with Wait” section). RP’d programs that are not running will be OF’d when the user logs off.

Removing Programs

The OF command is used to remove a program. To remove a program restored by the RU, XQ, or IT command, enter:

```
   CI> of <program name>
```

If the program was not restored with RP, its ID segment is released. If the program was restored with the RP command, only the execution is terminated, and the program ID segment remains intact. To remove the program's ID segment, the second parameter, ID, is needed. For example, to remove the ID segment of program TOWER that was restored with the RP command:

```
   CI> of tower id
```

The OF command (with or without the ID parameter) stops an executing program abruptly. Stopping a program in this way terminates the program execution without performing the normal clean-up operations. This command is normally used to stop a program in trouble. Any I/O operation in progress is terminated (any system resources used are returned). Data being written to a file is not posted, which may leave the file in an abnormal state.

Breaking Program Execution

You can use the BR command to stop a program in an orderly manner, rather than abruptly as with the OF command. BR can be entered when you do not want to wait for a program to finish. If the program was scheduled with wait (RU command), you must first interrupt the system and obtain the CM or break mode prompt. If the program was scheduled without wait (XQ or IT command), the BR command can be issued from CI. BR can be entered with or without a program name. For example:

```
   CI> test2
      <press any key>
   CM> br test2
   CI>
```

The program name must be the same name as the name reported by the WH command because CI may have made up the name to avoid having duplicate names. The BR command can be used without the program name to break the program most recently run without wait. Refer to Chapter 4 of the *RTE-6/VM Terminal User's Reference Manual* for details.

For this command to work, a program must acknowledge the break bit in the ID segment using the system call IFBRK (refer to the *RTE-6/VM Programmer's Reference Manual*). This is implemented in all system programs but not necessarily in user programs. If BR does not halt the program, you must wait until the program finishes or use the OF command.

Suspending a Program

Another method of stopping an executing program is to suspend it with the SS command. This command does not adversely affect the program or open file status; it simply suspends execution. SS is used the same way as the OF or BR command. However, the suspended program can be resumed with the GO command or terminated with the OF command.

The SS command does not interrupt any I/O operation in progress. It waits until the I/O operation is finished. Note that this may take a long time, and there is no message while CI is waiting.

Resuming Program Execution

Suspended programs can be resumed with a GO command. GO is used the same way as the OF, BR, and SS commands. The program is resumed at the point of suspension. For example:

```
CI> xq test2
CI> ss test2
CI> go test2
```

Normally, the GO command can be entered without any program name to resume the currently suspended program. Your System Manager can resume programs of other system users by specifying the name of the suspended program.

Restarting a Program

The RS command restarts a system program that is not executing properly; for example, a program that is hung on a downed device. The program is aborted and rescheduled for execution. The RS command can be used from CM or the system breakmode prompt.

The following example restarts CI:

```
CM> rs
```

Displaying Program Status

The ST command can be used to return the status of a specific program. ST returns some of the same information as the WH command, but in a shorter format. For example,

```
CI> st ftn7x
   90 0 0    0   0   0   0   0
```

Changing Program Priorities

All programs running under RTE-6/VM have a priority number that is recorded in the respective program ID segments. The priority number can be assigned when the program is written or when it is linked. It can also be changed dynamically with the PR command, as shown below.

The priority number can be in the range of 1 to 32767, with smaller numbers representing higher priorities. Typical values for user application software are in the range of 50 to 200. Higher priority real-time and system programs range from 1 to 40.

A primary task of the operating system is to run the highest priority executable user program followed by the next highest, and so on. When there are programs of the same priority, a technique called timeslicing is used. Programs of the same priority share the processor by having small intervals (or slices) of time allocated to them by the operating system in a round robin fashion. Timeslicing need not be implemented for all programs. A value called the timeslice fence is established at system generation time to set the priority below which timeslicing will be implemented.

If a user program has a very long elapsed running time in a busy system, or if it does not run at all, its priority may be too low. On the other hand, if it runs to the exclusion of other user programs, then its priority might be too high.

To change the priority of a program, use the PR command. For example:

```
CI> pr test2 50
```

(Changes priority of program Test 2 to 50)

Program priorities should be handled with caution. If you have a program with a very high priority, it might run continuously and prevent other programs from executing indefinitely.

Changing Memory Requirements

Some programs may require dynamic memory allocation; for example, reentrant routines or Pascal recursive procedures and dynamic data structures. Memory requirements may vary depending on input parameters or data given to the program. The operating system will not be aware of these factors and might not allocate enough memory to the program unless explicitly instructed to do so.

You can change the amount of memory allocated to a program in two ways. You can use LINK to make sure the program will get extra memory every time it runs (this is described in the *LINK User's Manual*.) Or you can use the SZ command after the program has been restored but before running it. For example, to change the memory allocation of DATALOGGER to 20 pages:

```
CI> rp datalogger
CI> sz datal 20        (Note the 5-character program name)
```

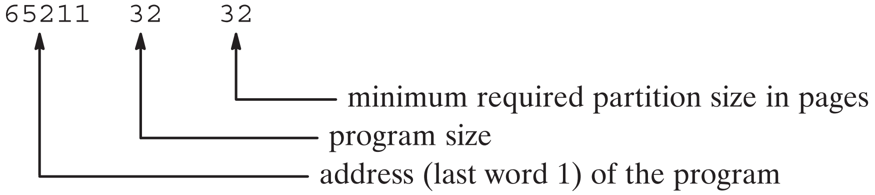
Now DATALOGGER will have 20 pages. The new memory partition allocation remains in effect as long as DATALOGGER is restored. If it is removed, it will revert to that defined by LINK. The SZ command cannot be used for a program that is executing.

If a program uses EMA, the SZ command modifies the EMA data space only (in the range of 2 to 1022 pages). For example:

```
CI> rp emapr
CI> sz emapr 300       (Changes the EMA space of EMAPR to 300 pages)
```

The size of a program can be displayed by entering the SZ command without parameters. For example:

```
CI> sz proge
```



```
65211  32   32
```

minimum required partition size in pages
program size
address (last word 1) of the program

Assigning Partitions

The system memory is divided at bootup time into dynamic and reserved partitions. Normally, when a program is run it is assigned memory as required from the dynamic memory. Reserved partitions are partitions of fixed sizes that can be reserved for specific programs. You can assign a reserved memory partition to a program with the AS command. The reserved partitions available can be checked with the WH,PA command.

For example, to assign PROGA to partition 1, which was previously created in the system, enter:

```
   CI> as proga 1
```

Program PROGA must be restored and must not be running.

When it is no longer necessary for a program to run in a reserved partition, you can remove the designation by using the AS command again, assigning the program to partition 0. There is never a partition zero; this number is used to remove the assignment. For example, to reassign PROGA to run in dynamic memory, enter:

```
   CI> as proga 0
```

Changing Virtual Memory Area

VMA programs are those that utilize an RTE feature that enables execution of programs requiring a very large amount of data storage. The data for a VMA program is contained in an area on disk called the Virtual Memory Area (VMA). The portion of data being processed is moved from disk to an area in memory called the Working Set (WS) so data is transferred between VMA and WS as necessary during program execution.

The WS size and the VMA space (VS) are defined using LINK, from 2 to 1022 pages of WS, and up to 65536 pages of VS. These can be changed with the WS or VS commands, respectively.

You may want to change the size of WS and VS with LINK because this changes the size permanently. Alternatively, you can use CI commands after restoring the program. The WS and VS commands can also be used to find out the area defined. For example:

```
CI> rp datalogger
CI> ws datal                          (Display VMA information for DATA1)
2
CI> vs datal                          (Display VMA information for DATA1)
3
    70135       31   36     5    1    8191
```

In both examples,

70135	=	logical address
31	=	program size in pages
36	=	minimum partition size
5	=	working set size
1	=	program's MSEG size
8191	=	virtual memory size

To change the WS and VS areas of a program:

```
CI> ws datal 45                       (Change working set size to 45 pages)

CI> vs datal 2500                     (Change VS size to 2500 pages)
```

The change made with the WS or VS command is effective as long as the program ID segment is in memory; when the program ID segment is released, the size reverts to that defined at program link time. Refer to Chapter 5 for more information on the WS and VS commands.

5

CI Command Descriptions

This chapter contains descriptions of all CI commands. The commands are described in alphabetical order. A tutorial of most of these commands and a command summary were provided in previous chapters of this book. Base set commands, which are available from the command prompt, are indicated by an asterisk (*).

? (Help)

Purpose: Displays a summary of CI commands or a brief description of a command or item on the summary display.

Syntax: `? [`*command*`]`

Description:

This command provides a quick reference of CI commands and utility programs. The form "?" without any parameters lists the HELP directory, showing a summary of available files. Entering "? <*command*>" lists a file called /HELP/<*command*>; for example, "? owner" lists file /HELP/OWNER. If there is no file by the name specified, a message is displayed. You can add files to the HELP directory to provide a quick reference of selected topics.

/ (Command Stack Editor)

Purpose: Allows previously entered command lines to be displayed, edited, and re-entered as new command lines.

Syntax: /[:][////|*n*][.*pattern*]

: denotes auto-execute mode.

//// represents the number of slashes that corresponds to the line number at which to start the frame; that is, /// equals line number 3, //// equals line number 4, and so on.

n is the line number at which to start the frame.

.pattern selects only lines that contain the specified pattern. The pattern syntax includes an optional anchor character (^) followed by one or more of the characters described below:

[^]{–|@|<*char*>|\<*char*>}...

^ (optional) anchors the pattern to the start of the command line

– matches any single character

@ matches zero or more characters

<*char*> matches the specified character

\<*char*> matches the specified character and allows "–" or "@" to be entered

\ quoting character

Description:

Note that although any number of slashes can be used to specify the line number at which to start the frame, entering one to four slashes (instead of /*n*) provides optimum speed. For line numbers higher than five, it is probably easier to enter a single slash followed by the desired line number.

Each command line entered is remembered in the "stack". The most recently entered commands are pushed onto the top of the stack, and the oldest commands fall off the bottom. Duplicate command lines are removed from the stack. When you enter command stack commands, these command lines can be displayed and modified with the terminal editing keys and sent again to CI as new command lines.

The command stack commands are entered from the CI> prompt, displaying a frame of previous command lines and entering command stack editing mode. Examples of commands that switch to stack mode are shown in Table 5-1.

Table 5-1. Stack Mode Commands

Command	Description
/	Display the last frame of lines
//	Display the last line
///	Display the last two lines (and so on)
/32	Display a frame starting at line 32
/.*pattern*	Display the last frame of lines containing *pattern*
/.^ *pattern*	Display the last frame of lines starting with *pattern*
//.*pattern*	Display the last line containing *pattern*
/5.*pattern*	Display the last 5 lines containing *pattern*

"Frame" above refers to the maximum number of lines to be displayed in a command stack window; in CI, this is specified in the FRAME_SIZE variable.

Additionally, you can include a colon (:) as the second character in any of the above command forms; this causes the first selected line to be automatically marked for execution and does not go into screen mode. For example, "/:/" re-executes the last line in the stack; "/:7" executes the seventh most recent command; "/:/.edit" executes the last line containing "edit".

The commands that select lines by patterns allow you to use the "−" and "@" characters in the pattern as single and multiple character wildcards, as in FMP masking. To specify either as a literal character, precede it with a backward slash (\). For example, "/.edit \−b@.ftn" finds all lines where EDIT was run in batch mode on any FORTRAN source; "/.a@b−c" matches lines "axxbbc" and "gab!c".

The command stack window is preceded by a banner that contains the starting line number of the window and the total number of lines selected for this display in inverse video, separated by a slash. For example, from CI:

```
CI> /3
--003/320--  Commands:  [/DEXTER/CI.STK]
edit glorp.ftn
ftn7x glorp 0 -,,s
link glorp.rel +de
```

This shows that the window starts at line 3 out of a total of 320 lines in the stack (CI also shows the name of the current stack file). If only those lines that contain a pattern are matched, the total line count reflects only those matching lines. For example:

```
CI> /3.edit
--003/024-- Commands:  [/DEXTER/CI.STK]
edit kaspritz.mac
? li;edit glink.mac
edit glorp.ftn
```

This shows that the window starts at selected line 3 out of a total of 24 lines selected by the pattern. To display the previous 20 lines that matched, use the CTRL-P editing command, as described below.

Once command stack mode has been entered, you may position the cursor to the desired line, modify it with the local terminal editing keys, and press <return> to execute the command. The editing commands shown in Table 5-2 are also recognized.

Table 5-2. Editing Commands

Command	Description
CTRL-A	Go to the start of the line where the cursor is positioned.
CTRL-D	Delete the current line from the stack.
CTRL-F	Display the following frame of selected lines.
CTRL-K	Mark current line for grouped execution in order of marking.
CTRL-P	Display the previous frame of selected lines.
CTRL-Q	Quit stack mode, start executing marked lines.
CTRL-U	May be entered instead of ctrl-Q on terminals using Xon/Xoff handshake protocol.
CTRL-Q CTRL-Q CTRL-U CTRL-U	Abandon stack mode, forget marked lines
CTRL-Z	Go to the end of the line where the cursor is positioned

In the table above, "ctrl-A" denotes pressing the <ctrl> key and the letter "A" at the same time. The editing mode commands are recognized only when they are entered immediately before a carriage return. In addition, any of the stack display commands may be entered to display a new window.

The stack lines are displayed with display functions when needed. Lines that are longer than the screen width are continued by dots in the last two columns. To enter a new line in editing mode that is longer than the screen width, you must use the cursor control keys to put the dots in the last two columns. The cursor must be placed on the first line of a continued command before selecting that line for execution or with one of the above commands.

A time out in the stack mode read causes CI to exit stack mode and return to the normal CI prompt.

Command stacks can be saved in files. At logon, a file called CI.STK is searched for on the home directory first. If it is not found there, it is searched for on the working directory or in the FMGR disk cartridges, if there is no working directory. If the command stack file is found, it is opened and its contents used to initialize the command stack. If it does not exist, the command stack remains cleared. In this case, the default file CI.STK is created on the working directory and the contents of the command stack posted there at logoff.

The file associated with the command stack can be changed to any file. The name of this file is designated with the WD command, and it can be in any directory accessible to the session user. Refer to the WD command description for details on changing and posting command stack files.

At logoff, if there is an open command stack file, the contents of the stack are posted to the file and the file is closed. If the file does not exist, it is created on the working directory or the top cartridge on the FMGR cartridge list. The contents of the stack is posted to this file, which is then closed. If a command stack file had not been specified, file CI.STK is used. If you do not want to either save your command stack in a file or have the current file updated, set the predefined variable $SAVE_STACK to FALSE.

Note that the stack subroutines are available for inclusion in user programs.

Examples:

Assume that the command stack contains the following and that the variable FRAME_SIZE is equal to 20:

```
wh us
who
dl ::users
? dl
dl ::system
dl
hello
?
? ex
dl
ru dl
ss spot.run
go spot.run
br spot.run
tm
wh
io 6
up 6
io 8
wh al
? wh
? pu
pu spot.lst
pu spot.dbg
wh pa
edit testprog.ftn
co testprog.ftn backup.ftn
li testprog.ftn
```

To display the last 20 commands:

```
 CI> /
--020/320-- Commands:  [/DEXTER/CI.STK]
? ex
dl
ru dl
ss spot.run
go spot.run
br spot.run
tm
wh
io 6
up 6
io 8
wh al
? wh
? pu
pu spot.1st
pu spot.dbg
wh pa
edit testprog.ftn
co testprog.ftn backup.ftn
li testprog.ftn
```

The cursor is at the bottom of the stack at a blank line. Now you can press the return key to return to CI without any further action or move the cursor up to select any command line. The terminal editing keys can be used to make changes and the command can be entered by pressing the return key, or you can use any of the stack mode commands described in the following section.

To display 20 lines beginning at command line 25 from the last command:

```
 CI> /25
--025/320-- Commands:  [/DEXTER/CI.STK]
? dl
dl  ::system
dl
hello
?
? ex
dl
ru dl
ss spot.run
go spot.run
br spot.run
tm
wh
io 6
up 6
io 8
wh al
? wh
? pu
pu spot.1st
```

Note that the cursor is positioned at the top of the stack. If n is less than 20, the number of lines specified will be displayed. For example:

```
CI> /9
--009/320-- Commands:  [/DEXTER/CI.STK]
wh al
? wh
? pu
pu spot.1st
pu spot.dbg
wh pa
edit  testprog.ftn
co testprog.ftn backup.ftn
li testprog.ftn
```

You can achieve the same result by entering the following:

```
CI> //////////
```

The following command sequence displays the last command line:

```
CI> //
--001/320-- Commands:  [/DEXTER/CI.STK]
li testprog.ftn
```

Posting and changing command stack files are done with the WD command. Following are examples of manipulating command stack contents and associated files:

Initial assumptions: working directory = /DEBBIE
associated command stack file = /DEBBIE/CI.STK

`CI> wd,,+s` (Command stack contents posted to /DEBBIE/CI.STK)

`CI> wd /tsmas ts.stk` (New working directory is /TSMAS. Command stack contents posted to /DEBBIE/CI.STK. Contents of /TSMAS/TS.STK are written into command stack. If TS.STK does not exist, the command stack is cleared and TS.STK will be created at logoff or when the next WD command with the command stack option is executed)

`CI> wd /debbie +s` (Working directory is changed to DEBBIE. Command stack contents are posted to /TSMAS/TS.STK; if it does not exist, it is created. Contents of CI.STK are written into the command stack)

Changing the command stack display frame size is done with the SET command (described earlier in this chapter) and predefined variable frame_size. An example follows:

```
CI> SET frame_size = 15        (Sets command stack display/frame size to 15)
```

Example:

Assume that the command stack contains the following command lines:

```
tr,transferfile.cmd
wh us
who
dl ::users
? dl
dl ::system
dl
hello
?
? ex
dl
ru dl
ss spot.run
go spot.run
br spot.run
tm
wh
io 6
up 6
io 8
```

To change the display size to 7 lines and then display the last 7 lines, enter the following:

```
CI> set frame_size = 7
CI> /
--007/320--  Commands: [/DEXTER/CI.STK]
go spot.run
br spot.run
tm
wh
io 6
up 6
io 8
```

AG (Modify Partition Priority Aging)*

Purpose: Modifies the rate a partition's priority number is increased and turns on or off partition priority aging.

Syntax: `AG` *numb*|`of`

numb Number of 10-millisecond intervals to be used as the aging rate. This value must be in the range of 10 and 32767.

of Turns off partition priority aging.

Description:

Partition aging is a feature that allows high-priority suspended (state 3) programs to be swapped out, replaced by lower priority programs. Details of the AG command are given in the *RTE-6/VM Terminal User's Reference Manual.* Increasing the priority number of a program lowers the program priority.

Examples:

```
CI> ag 100
```

(Increase the partition priority number by two every second)

```
CI> ag of
```

(Turn off all partition priority aging)

AS (Assign Partition)*

Purpose: Assigns a program to a reserved partition.

Syntax: `AS` *prog part_#*

prog	The program name, up to five characters.
part_#	A number that identifies the partition to which the named program will be assigned. Partition number = 0 removes the current assignment.

Description:

The AS command is identical to the SYSTEM AS command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

Examples:

```
CI> as test2 2
```

(Assign program TEST2 to reserved partition 2)

```
CI> as test 0
```

(Program test to run in any partition)

ASK (Display a Prompt and Read a Response)

Purpose: Displays a question or prompt, reads the response from the terminal and passes it back to the scheduling program in Return_S.

Syntax: `ASK` ‘*character string*‘

character string Any question or prompt you want to use. The string must be enclosed in backquotes (“).

Description:

The ASK command displays a question or prompt, reads a response from the terminal, and passes back information about the response to the scheduling program in return variables. It passes the following information back: an indication if the command was successful (Return1), the character length of the response string (Return2), the index of the first response character in the option string (Return3), and the response string (Return_S).

If the character string parameter of ASK contains a ‘?’, the question string stops at the first ‘?’ found and the rest of the string is taken as an option string. The zero relative index of the first response character in the option string is returned to the scheduling program. If you want it to be 1 relative, put a space between the ‘?’ and the first character in the option string. If the first character in the response is not in the option string, a −1 is returned. If there are no non-blank characters in the response, a −2 is returned.

ASK returns the following in $RETURN1−$RETURN3 and $RETURN_S:

$RETURN1:	0	command executed successfully
	−1	error in executing command
$RETURN2:		contains character length of response string
$RETURN3:	≥0	zero relative index of the first response character in the option string
	−1	first response character is not in the option string
	−2	no non-blank characters were entered
$RETURN_S:		contains the response string

Examples:

```
CI> ASK ‘How are you?‘
```

(Display the question “How are you?” on the next line, read the answer, and pass it to the scheduling program. If no option string is specified, the index return3 is −1)

```
CI> ASK ‘Command>‘
```

(Display the prompt “Command>” on the next line, read the response, and pass it to the scheduling program)

```
CI> ASK ‘Purge this (Yes, No, Abort, Stop asking)?YNAS‘
```

(Display the question: `“Purge this (Yes, No, Abort, Stop asking)?”,` read the response, determine the zero relative index of the first response character in the option string, and pass the information back to the scheduling program.

This will return the following in Return3:

0	if Y is the first character in the response
1	if N is the first character in the response
2	if A is the first character in the response
3	if S is the first character in the response
−1	if the first character in the response is some other character
−2	if there are no non-blank characters in the response

BL (Examine or Modify Buffer Limits)*

Purpose: Allows the general user to examine the current buffer limits and a System Manager to change the current buffer limits.

Syntax: `BL [`*lower limit* `[`*upper limit*`]]`

lower limit Used by the System Manager only; specified in number of words. If upper limit is changed and lower limit is not specified, it defaults to 1.

upper limit Used by the System manager only; specified in number of words. If lower limit is changed and upper limit is not specified, it remains the same as the existing upper buffer limit.

Description:

The BL command is identical to the SYSTEM BL command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

Examples:

```
CI> bl                    (Display lower and upper buffer limits)
100 400

CI> bl 200 500            (Change buffer limits)
```

BR (Break Program Execution)*

Purpose: Sets a flag to allow limited communication with a program.

Syntax: `BR [`*prog*`]`

prog The program name, up to five characters, with an optional session identifier. The default is the last scheduled program.

Description:

This command is used to stop programs in an orderly manner. BR sets a break flag in the program's ID segment, providing a way to signal a running program that it is to be stopped. It is up to the program to check the break flag; otherwise, the command has no effect. The break flag can be checked with system call IFBRK, described in the *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005.

If no program is specified and the startup program (usually CI) has scheduled another program, the BR command executes on that program unless it, in turn, has scheduled a program. The search continues down the program scheduling chain, and the BR command is executed on the last program. However, if the last program is a protected system program, the BR command executes on the program that scheduled the protected system program.

CD (Change Working Directory)

Purpose: Changes the working directory.

Syntax: `CD [-|`*directory*`]`
`CD` *old new*

Description:

The CD command can take one of two forms. In the first form, it changes the current directory to *directory*. If a dash (−) is specified as the argument, the directory is changed to the previous directory ($OLDPWD). The default for *directory* is the value of the $HOME variable.

The second form of CD substitutes the string *new* for the string *old* in the current working directory name, $WD, and tries to change to this new directory.

When either the CD or WD command is used, the variables $WD and $OLDPWD are updated. The WD command always sets $WD to the physical name of the current working directory.

Examples:

```
CI> pwd
/niners/lott
CI> cd /raiders
CI> pwd
/raiders
CI> cd -
CI> pwd
/niners/lott
```

Suppose LOTT.DIR exists in both /NINERS.DIR and /RAIDERS.DIR. The "`cd` *old new*" syntax can be used to switch directories. In the following example, the current working directory is /NINERS/LOTT; by substituting the string "nin" for all occurrences of the string "raid", the current working directory is changed to /RAIDERS/LOTT:

```
CI> pwd
/niners/lott
CI> cd nin raid
CI> pwd
/raiders/lott
```

CL (List Mounted Disks)

Purpose: Displays all mounted disk volumes.

Syntax: `CL`

Description:

The CL command is used to show the mounted disk volumes by their logical unit numbers. It lists separately all LUs mounted as file system LUs and all LUs mounted as FMGR LUs. For FMGR system disks, the LUs and the associated CRNs are listed in the FMGR search order.

Examples:

```
CI> cl
File System Disk LUs:  54  56
FMGR Disk LUs (CRN):  27(DB)      45(TY)      46(PM)      30(XX)
61(SO)      59(GR)
```

CN (Control Device)

Purpose: Controls peripheral devices.

Syntax: `CN` *lu function* [*pram*4*]

lu The logical unit of device to receive the control request.

function The control function code (0-63B) as defined in the function field of CNTWD (listed for each driver in the appropriate driver reference manual) or a two-character mnemonic code from the following:

Mnemonic Code	Equivalent Octal Function Code	Pram 1-4 Definition	Action
TO	11B	# lines on page	Issue top-of-form or line spacing on printer
RW	4	None	Rewind cassette tape
EO	1	None	Write end-of-file
FF	13B	None	Forward space file
BF	14B	None	Backward space file
FR	3	None	Forward space record
BR	2	None	Backward space record

*pram*4* The optional parameters that specify additional device details as appropriate for a given driver. Specific meanings for these parameters may be found in the appropriate driver reference manual for each driver.

For magnetic tapes and cassette tapes, the function parameter defaults to rewind tape; for printers, the default is form feed.

Examples:

```
CI> cn 4 rw
```

(Rewind the tape in cassette tape unit, LU 4)

```
CI> cn 6 to-1
```

(Cause a top-of-form, page feed, on printer LU 6)

Refer to the appropriate driver reference manual for full information on the control requests that can be issued for each driver.

CO (Copy Files)

Purpose: Copies one or more files between directories and/or I/O devices.

Syntax: `CO` <*file1*|*lu*> <*file2*|*lu*> [*option*]

file1|*lu* The source file descriptor or the LU number of an I/O device. (Refer to the CR command syntax description for the definition of file descriptor.) May be masked to operate on more than one file. (Refer to the "File Masks" section in Chapter 3 for the mask syntax.)

file2|*lu* The destination file descriptor or the LU number of an I/O device. May be masked to allow the system to generate destination names. When copying from a device, the default file type is type 3; a different file type must be specified if one is desired. Note that the destination LU should not be a cartridge tape drive.

option The following characters indicate particular actions to be taken:

A	ASCII records, no checksum (default)
B	Binary
C	Clear backup bit on source after copying (note that backup bits for @.DIR files are not cleared with this option)
D	Replace duplicates; existing file with the same name will be replaced
N	No carriage control in source
P	Purge source after copying
Q	Quiet – do not record access time on source
T	Truncate destination to length of valid data
U	Replace duplicates if update time is older

Description:

The CO command can be used to copy a group of files from one directory to another. Masking the file1 parameter allows matches of a number of files. If a wildcard character is used in the name field of file1, an appropriate destination mask must be used to default destination file names.

The file mask is a very powerful but complicated tool, and it should be used with caution. For example, you can copy all type 6 files on several different directories to a particular directory, which can be a global directory or a subdirectory.

An implicit D qualifier is used whenever you use a wildcard mask. This means that if any directory matches the mask, all files in that directory will also be copied. The D qualifier can be overridden with the mask qualifiers K or N, which is particularly useful with time qualified copies, since directory time stamps are not maintained. Note that the D qualifier is automatically appended to the unspecified mask and appears in error messages. For example:

```
CI> co /global/@.ftn /new/@.ftna
No such directory @.FTN.D::GLOBAL      (D appended to file name)
```

When copying a file from one directory to another, the creation and access times are those of the copying process. However, the update time of the new file is that of the current file, to maintain a history of the latest revision date.

When copying a source file with a record length greater than 128 words to a device, the destination file is truncated to 128 words per record. When copying a file to a line printer, the characters in the first column are not printed because they are used by the printer for carriage control. The N option indicates that characters in the first column are printed.

The file type of the destination file is the same as the source file if you do not specify a different one. If the destination file size is not specified, a size will be selected to eliminate extents. The protection of the destination file will be the same as the source file if the source is not an LU or a FMGR file and the other user is the owner of the destination directory. Otherwise, it will have the protection of the directory into which it is copied.

When using the C option, the backup bit on directories is not cleared. The backup bit and the time stamps on directories are never changed, because although directories are structured like files, they are not accessed like files. When a directory is copied, a new directory by the same name is created and all the contents are copied.

The Q option is used when you do not want to have the access time of the file updated. It is useful when you are copying from a file that resides on a write-protected disk. Normally, the file system attempts to update the file access time when it opens the file, and because the LU is write-protected, the CO command would fail.

The T option allows a file with wasted space to be copied 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 is not used with type 1, 2, and 6 files or FMGR files.

The U option allows overwriting of the destination file, but only if the destination file's update time is older than the source's. Since FMGR files do not have update times, they are considered the oldest.

Examples:

```
   CI> co @.src.e /backup/archive/source/@.@
```

This command copies all files with file type extension .SRC on all accessible directories to subdirectory SOURCE of subdirectory ARCHIVE of directory BACKUP. Their names and file type extensions remain unchanged. Note that all files copied by this directive will be copied to the same directory. To copy subdirectories to subdirectories, use the K qualifier in the mask instead of the S qualifier.

```
   CI> co @.rel 8 b
```

This command copies all files with file type extension .REL on the working directory to LU 8. Note that this example shows that CO can be used to copy to an I/O device. The preferred method is to use the TF utility for this type of copying.

```
   CI> co 8 /programs/program.run:::6:1000
```

When copying from a device (such as a tape unit), the default file size is 24 blocks. If the file is longer and extents are not desirable (that is, type 6 files), a longer file size must be explicitly specified. After copying, the file is truncated to its actual size.

```
   CI> co sub.dir.d sub/sub.dir  (SUB.DIR is in the working directory.
                                 SUB/SUB.DIR is not created.  You cannot copy a
                                 directory into its own subdirectory.)
```

If CI were to allow copying a directory into a subdirectory of itself, this command would find subdirectory SUB in the working directory and copy it into subdirectory SUB, creating file SUB/SUB.DIR. Then following the d directive, all files in subdirectory SUB would be copied, including SUB/SUB.DIR. This would continue until the string reached 63 characters.

```
   CI> co @ /dir/@ t
```

This command copies all the files in the working directory into /DIR/, but only copies as much data as the directory information says is valid.

```
   CI> co @ /backup/@ u
```

This command copies into /BACKUP/ the files in the working directory whose update times are newer than the corresponding file in /BACKUP/.

CR (Create File)

Purpose: Creates a disk file.

Syntax: `CR` *file*

`CR` *file* [*user*] >*node*

file File descriptor, up to 63 characters, in any of the following formats:

Standard
[/*dir*/ [*subdir*/]] *filename* [: : : *type* [: *size* [: *rlen*]]] [*ds_port*]

Combined
[*subdir*/] *filename* [: : *dir* [: *type* [: *size* [: *rlen*]]]] [*ds_port*]

FMGR
filename [: *sc* [: *crn* [: *type* [: *size* [: *rlen*]]]]] [*ds_port*]

where:

dir Specifies the unique (global) directory for the file. The directory name can be up to 16 characters long, not counting delimiters (slashes). If omitted, the working directory is used.

subdir Specifies one or more subdirectories for the file, separated by slashes (/). Each subdirectory can be up to 16 characters long not counting delimiters. Any number of subdirectories can be specified with the limit of 63 characters for the full file descriptor.

filename Specifies the name of the file including a file type extension. The file name can be up to 21 characters: 16 characters for the name followed by a period and 4 characters for the file type extension. The file type extension is used to describe the type of information in the file. Standard file type extensions are described in Chapter 3 of this manual.

Mask characters (@ and −) can be used to specify a group of files; @ masks any one or more characters and the dash (−) masks one character position for any character except a blank. Only the first 6 characters are valid for FMGR files; other characters, the type extension, and qualifier options are ignored.

typex A file type extension up to 4 characters appended to file name with a period as the delimiter; can be used to describe the type of information in the file. The @ and − mask characters can be used in the *typex* field. Standard file type extensions are:

.cmd	CI command file
.dat	data file
.dbg	Symbolic Debug/1000 file
.dir	directory or subdirectory entry
.doc	document file
.err	error message file
.ftn	FORTRAN source file
.ftni	FORTRAN source include file
.hlp	Help file
.lib	library of relocatables
.lod	LINK command file
.lst	listing
.mac	MACRO source file
.maci	MACRO source include file
.map	load map list
.merg	merge file for relocatables without headers
.mlb	MACRO library file
.mnf	manual numbering file
.mrg	merge file for relocatable libraries with headers
.pas	Pascal source file
.pasi	Pascal source include file
.rel	relocatable (binary) file
.run	program file
.snp	system snapshot file
.stk	command stack file
.sys	system file
.txt	text file

type A number used to indicate how the file is organized. Standard types are:

1 Type 1 files are random access files that do not have structure information. They can be read and written very quickly, but are not suitable for use as text files. Fixed length records are 128 words long.

2 Type 2 files are fixed-length record, random access files. The record length is defined when the file is created. They are not suitable for use as text files.

3−7 Type 3 and above files are variable length record, sequential files. They are suitable for use as text files. There is no difference in the handling of file types 3 and above. By convention, types 5, 6, and 7 are used for relocatable object, executable program, and absolute binary files, respectively.

If *type* is not specified, 3 is used. Types greater than 7 are user defined.

size Specifies the file size in number of blocks. Default is 24 blocks.

rlen Specifies the record length in type 2 files in number of words.

ds_port Specifies the node and user for files to be accessed via DS transparency. The format of *ds_port* is:

>*node* [*user/password*]

or

[*user/password*] >*node*

sc (optional) A one-word security code that limits read and write access to the file. It can be zero, a positive integer, or a negative integer. Zero allows the file to be opened by any user or program with access to the disk cartridge containing the file. A positive integer (or two ASCII characters) restricts writing to the file, but not reading. A negative integer restricts all access to the file, providing read and write protection; this code must be specified in order to open a file protected by it.

crn (optional) Used in the FMGR compatible format only; can be a positive number (cartridge reference number, CRN), or a negative LU number, or two characters.

Description:

The CR command creates an empty file. The minimum information that must be specified is the name. The remaining parameters can be defaulted. Default values are:

file type extension:	blank
directory:	working directory
type:	3
size:	24 blocks

To create a file, you must have write access to the directory where the file will reside. The owner of this file is the owner of the directory. The protection status of this file is the same as that for the directory it is on. This lets you write into a file or create a file on another directory to which you have write access. Only the owner of the directory can alter the protection status of the file thus created.

Examples:

```
   CI> cr /applications/documentation/compiler
```

This example creates an empty file named COMPILER with the following attributes: blank file type extension, size = 24, type = 3, in subdirectory DOCUMENTATION on global directory APPLICATIONS.

```
   CI> cr /joe/notes.txt:::4:10
```

This example creates file NOTES.TXT with the following attributes: file type 4, size = 10 blocks, in directory JOE.

```
   CI> cr data.dat:::2:5:18
```

This example creates file DATA.DAT as a type 2 file with 5 blocks and a record length of 18 words in the working directory.

```
   CI> cr notes/project.txt
```

This example creates file PROJECT.TXT in subdirectory NOTES on the current working directory. The default attributes are used: type 3, 24 blocks.

CRDIR (Create Directory/Subdirectory)

Purpose: Creates a global directory or a subdirectory.

Syntax: `CRDIR` *directory* [*lu*]

directory The character string that identifies the directory. It can be up to 63 characters and either a global directory or a subdirectory. The directory in which a subdirectory is created must already exist.

The name can include an optional size subparameter specified in number of blocks as follows:

directory::::*size* (for example, /JONES::::24)

The default size is equal to the track size of the disk used, typically 48 or 64 blocks for hard disks and 30 or 16 for flexible disk. Directory size is extended as needed.

lu Specifies where to place a global directory. It must be a mounted disk volume. If it is set to zero, the disk volume of the working directory is used. This parameter is ignored for subdirectories, which go on the same volume as the directory in which it resides.

Description:

The CRDIR command creates a directory or subdirectory. A subdirectory can be created within a subdirectory. There is no limit to the level of subdirectory nesting except for the 63 character limit to any file descriptor.

If the optional disk volume parameter is omitted and there is no working directory, the lowest numbered disk volume is used.

The size of the directory can be specified in the same way as a file is created. There are four directory entries per block, and two directory entries are used for internal information. Thus, if a size of four blocks is specified, the directory can hold 14 file entries (extents require additional entries) before the directory needs to be extended. As is the case with files, extents slow directory search performance. The created size is not a limit on the number of entries in a directory. The maximum size allowed is 64 blocks. Some programs assume that directories contain no more than 32767 files.

If a directory is created with the same name as a FMGR CRN, the FMGR disk cartridge cannot be accessed by any CI command unless the working directory is set to 0.

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.

Examples:

`CI> crdir jones`	(Create Subdirectory JONES in the working directory)
`CI> crdir jones::::12`	(Create Subdirectory JONES in the working directory with 12 blocks)
`CI> crdir smith/jones`	(Create Subdirectory JONES on Subdirectory SMITH in the working directory)
`CI> crdir /smith/jones`	(Create Subdirectory JONES, in global directory SMITH)
`CI> crdir jones::smith`	(Create Subdirectory JONES in global directory SMITH)
`CI> crdir ::HP`	(Create global directory HP on the same LU as the working directory)
`CI> crdir /HU`	(Create global directory HU on the same LU as the working directory)

CU (CPU Utilization)*

Purpose: Displays a bar graph of CPU display registers showing the percentage of CPU utilization.

Syntax: `CU on|off`

`on` Turns display on.

`off` Turns display off.

Description:

The CU command is identical to the SYSTEM CU command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

DC (Dismount Disk Volume)

Purpose: Dismounts a disk volume.

Syntax: `DC` *lu*

lu The positive LU number of the disk volume to be dismounted.

Description:

The DC command dismounts a disk volume, making the global directories on that disk inaccessible. If there are any open files, working directories, or active type 6 files (or the swap file), an error message is displayed and the LU specified is not dismounted. The first problem encountered causes the error message. If there are more problems, it may take several tries to discover and correct all of them.

For SYSTEM disk volumes, use the SYSTEM DC command because it provides more information when there are active programs. If the dismount fails on a FMGR disk cartridge, the disk remains mounted but moves to the bottom of the volume list.

DL (Directory List)

Purpose: Lists files in a directory.

Syntax: `DL` [*mask* [*options* [*file*|*lu* [*msc*]]]]

mask A field specifying the names of files matching the mask to be displayed. The default is all the files in the working directory.

The file mask can include any or all of the file descriptor parameters and a mask qualifier appended to the file name parameter. Refer to the "File Masks" section in Chapter 3 for the file mask syntax description.

options The parameters that specify what particular information from the directory will be displayed. They can be listed without any delimiters, in any order.

A Time last accessed displayed in the following format:
Wed Jun 30, 1988 9:55:48 am

B Files that have not been backed up to be marked with asterisk (*).

C Creation time displayed in the same format as option A.

E File type extension (for sorting only).

F File type.

L Location of the file; displays the block address and LU of the main file entry. The first block on disk is address 0.

M Main file size in blocks, excluding extents.

N Number of records.

O Mark open files by displaying the name of the program that has the file open next to that file. If there are no open files, this field is not displayed.

P Protection on the file is displayed in the following form (read and write abbreviated to first letter):

owner/other The directory containing the entities listed does not have a group associated with the owner (rw/r).

owner/group/other The directory containing the entities listed has a group associated with the owner (rw/r/r).

R Record length; gives length of longest record in the file in words.

S Size; the total number of blocks used by file, including extents.

T Temporary file marked with asterisk (*).

U Time last updated is displayed in the same format as option A.

W Words in the file, up to EOF.

X File with extents to be marked with an asterisk (*).

Y Security code (FMGR files only).

* A useful subset of the above (FWNSXP).

! All of the above.

+ Ascending sort by item specified.

− Descending sort by item specified.

file | lu An optional file or LU number where the DL output is to be stored.

*ms*c The master security code for the system. Needed only if the security codes of FMGR files are requested (Y or ! option).

Description:

The DL command displays a list of the files that match the specified mask in a directory or subdirectory. The display format is as many names as possible per row if no options are specified. If any display option is specified, the format requires one line per file. If several options are specified, multiple lines per file may be required.

The display is normally sorted by name. There are two sort options: '+' for ascending sort order and '−' for descending sort order. Preceding an option specifier with '+' causes the list of files to be sorted with the lowest value first. Preceding an option with '−' causes the reverse. If either '+' or '−' is specified and not followed by an option specifier, the names are ascending or descending sorted. The default is an ascending sort by name. The number of files that can be sorted depends on the amount of free memory the program has.

If there are too many files, as many as possible are sorted and displayed, then another list of files is sorted and displayed until all the files are displayed. Sizing the DL program scheduled by CI to a larger size increases the number of files that can be sorted at one time.

Some of the information in the directory is dynamic and may not always be accurate, particularly if a file is open or the last program that accessed that file failed to close it. This information includes access time, total size, time last updated, and words in file. These fields can be specified with the options A, S, U, and W, respectively. Note that for FMGR files, only the options F, L, M, O, R, and Y are displayed; other fields are not maintained in the directory for FMGR files.

The E option is used only for sorting because the file type extension is always displayed. If specified with '+' or '−', the files are sorted by type extension and file name. The E option is ignored if specified without '+' or '−'.

The protection displayed depends on whether the group is defined for the directory in which the file exists. If the directory is on an older RTE-6/VM system, the group protection will not be displayed; the display will appear as RW/R, for example. Otherwise, the group protection will be displayed, for example, as RW/R/R.

For FMGR files, the master security code parameter is needed only if the Y option is specified. If an incorrect master security code is entered, no security code is displayed. Note that if the master security code is zero, any value (or no value) can be entered for the msc parameter. If necessary, see your System Manager for the system security code.

Examples:

`CI> dl` (Display all files in the working directory)

`CI> dl @.dir` (Display all subdirectories on the working directory)

`CI> dl a@..c83 -s` (Display files that start with ‘a’ and were created during 1983, sorted in descending order by size)

`CI> dl /program/` (Display all files in directory PROGRAM)

`CI> dl /joe/foo` (Display file FOO in directory JOE)

`CI> dl @.txt +s` (Display files with file type extension TXT on the working directory, displaying the size in number of blocks sorted in ascending order)

`CI> dl joe/f@.@.sc80-83` (Display files in directory JOE that start with ‘f’, have any file type extension, and were created during 1980 through 1983. The S option in the mask qualifier directs a search of all subdirectories of directory JOE for similar files)

`CI> dl /joe/@.dir` (Display all subdirectories in JOE)

`CI> dl,@::sc,y,,hp` (Display all files on CRN SC with their security codes; msc is HP)

```
CI> dl,,!
directory DEMO
    name   ex   ba  tmp   prot   type  msize  blks  words  recs  rlen  addr/lu

COPY.REL         *       rw/r/r     5     86    86   6312   127   128  8390/38
create time     Wed Jan 12, 1989     9:16:13 am
access time     Wed Jan 12, 1989     9:47:09 am
update time     Wed Jan 12, 1989     9:39:47 am

COPY.SRC    *    *       rw/r/r     4     92   184  13418   820    38  7908/38
create time     Wed Jan 12, 1989     9:00:33 am
access time     Wed Jan 12, 1989     9:44:29 am
update time     Wed Jan 12, 1989     9:30:35 am
```

The above example gives a complete directory listing of the working directory with two files. The display columns of those shown above and those in the O and Y options are:

`ex` – extent; * indicates file has extents (X option)
`ba` – backup; * indicates file needs to be backed up (B option)
`tmp` – temporary; * indicates file is a temporary file (T option)
`prot` – protection; shows file access for owner/other (P option)
`type` – file type (F option)
`msize` – size of main file (M option)

blks – size of file in blocks (both main and extents) (S option)
words – number of words up to the end-of-file mark (W option)
recs – number of records in the file (N option)
rlen – length of longest record in file (R option)
addr/lu – block address of beginning of file (L option)
open – name of program (if any) accessing the file (O option)
sc – security code; displayed only for FMGR files (Y option)

```
CI> dl @ -1*m
directory DEMO
    name   ex  prot  type msize blks words recs addr/lu

COPY.REL      rw/r/r   5    86    86  6312  127 8390/38
COPY.SRC    * rw/r/r   4    92   184 13418  820 7908/38
```

```
CI> dl ’@::pr !
directory /PR.DIR
     name   ex sc type msize rlen addr/lu

’BORDL          0    4    48    0 5341/40
’DL**        *  0    4    24    0 6080/40
’IN**        *  0    4    24    0 5646/40
’PAL11          0    4   192    0 4150/40
’PROM           0    4    24    0 1615/40
```

This example demonstrates the limited directory information available for FMGR files.

DN (Down a Device or I/O Controller)*

Purpose: Declares a device or I/O controller down (unavailable for use by the RTE system).

Syntax: `DN,,`*lu*

or

`DN,`*eqt*

lu	Specifies the system LU of the device to be declared down.
eqt	Specifies the Equipment Table (EQT) entry number of the I/O controller to be declared down.

Description:

A downed device (or I/O controller) can be made available by the UP command. The EQT and LU number can be displayed with the LU command under CI. The DN command is identical to the SYSTEM DN command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

Examples:

```
CI> dn,,6                (Declares LU 6 down)

CI> dn,28                (Declares EQT 28 down)
```

ECHO (Display Parameters at Terminal)

Purpose: Displays parameters, separated by commas, at the terminal.

Syntax: `ECHO [`*parameters*`]`

parameters One or more parameters separated by blanks or commas. Positional, user-defined, and predefined variables can be included in the string. If this parameter is omitted, a blank line is displayed.

Description:

The ECHO command displays the specified string after CI shifts the input to uppercase, puts commas between the parameters in the string, performs variable substitution, and removes CI quotes (backquotes and backslashes). You can use CI backquotes to keep CI from altering any parameters in the input string.

Positional, user-defined, and predefined variables are referenced by including a dollar sign ($) before the variable name. If you want to examine the value of only one variable, you can use the ECHO command instead of the SET command.

Examples:

```
CI> echo ru edit test.ftn       (Display specified string)
RU,EDIT,TEST.FTN                (String is uppercase and commas separate
                                parameters)

CI> echo $session               (Display value of $SESSION)
45                              (Session number is 45)

CI> wd /mine/temp               (Set working directory)

CI> echo ‘Your working directory is ‘$wd       (Display message indicating
Your working directory is /MINE/TEMP           your current working
                                               directory)
```

EQ (Displays I/O Controller Status)*

Purpose: Displays a description and the status of an I/O controller, as recorded in the Equipment Table (EQT) entry.

Syntax: `EQ` *eqt*

eqt Specifies the EQT entry number of an I/O controller.

Description:

The EQT number can be displayed with the LU command under CI. The EQ command is identical to the SYSTEM EQ command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

EQ (Buffering)*

Purpose: Changes the automatic buffering designation for a particular I/O controller.

Syntax: `EQ` *eqt un|bu*

eqt Specifies the Equipment Table (EQT) entry number of the I/O controller.

`un` Turns off (unbuffer) buffering.

`bu` Turns on buffering.

Description:

The EQ command is identical to the SYSTEM EQ command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

EX (Exit)

Purpose: Terminates the Command Interpreter program.

Syntax: `EX`

Description:

EX terminates CI and prints the message “Finished”.

In addition, if CI is the primary program, EX begins the logoff procedure. Before logging off, all active session programs must be terminated. The following message is issued if any session programs are active:

```
PROG1
PROG2
ABOVE SESSION PROGRAMS ACTIVE
OK TO ABORT ? (Y OR N)
```

If N is entered, the logoff procedure is terminated and CI is scheduled again. If a Y is entered, the active programs (PROG1 and PROG2) are aborted and the logoff sequence continues.

The LGOFF program then updates the session’s user and group accounting information by storing the logoff time, cumulative connect time, and CPU usage in the Account File.

The following is an example of a logoff message issued to the session terminal:

```
SESSION:   9    OFF 10:09 AM  THU., 15  FEB.,  1989
CONNECT TIME:                 00 HRS.,  04 MIN.,   20 SEC.
CPU USAGE:                    00 HRS.,  00 MIN.,   19 SEC., 40 MS.
CUMULATIVE CONNECT TIME:     06 HRS.,  04 MIN.,   03 SEC.
END OF SESSION:
```

The first line of the message is also sent to the system console.

FL (Flush Terminal Buffer)

Purpose: Eliminates buffered output to an auxiliary terminal.

Syntax: `FL`

Description:

The FL command is only valid from CM and break mode prompts. It is illegal if entered from the system console unless it has been enabled to run under session control.

Another method for clearing the buffer is using EXEC calls (refer to the *RTE-6/VM Programmer's Reference Manual* for descriptions of the EXEC calls):

```
   CALL  EXEC(3,32300B+lu)
```

where lu is the LU of the terminal to be flushed (normally LU 1 when under session control). A high priority program could use this call prior to writing an important message to the terminal. Note that the program issuing the call must have a priority of 40 or higher and must have a priority higher than the program that generated the data to be flushed.

Note This is only useful on slow ports (for example, teletypes).

GO (Resume Suspended Program)*

Purpose: Resumes execution of a suspended program.

Syntax: `GO [`*prog* `[`*pram*5*`]]`

prog The name of the suspended program.

*pram*5* The parameters to be passed to the program only if the program has suspended itself.

The GO command is identical to the SYSTEM GO command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

HE (Help)*

Purpose: Provides explanation of an error and guidance in possible corrective action.

Syntax: `HE [`*keyword* `[`*lu*`]]`

keyword A select group of eight or fewer characters identifying the error for which an explanation is requested. All keywords and the corresponding explanations are contained in a disk-resident HELP file. The default is the last error that occurred in that session.

If the keyword contains a space or blank character, it should be enclosed in backquotes (‘). For example:

```
HE ‘FMGR 059‘
```

lu LU of the device where the explanation is to be sent. The default is the session user’s terminal.

Description:

The HE command is identical to the SYSTEM HE command. Refer to the *RTE-6/VM Terminal User’s Reference Manual* for a complete description.

IF-THEN-ELSE-FI (Control Structure)

Purpose: Allows decision making in a command file.

Syntax:

```
IF command_list1
THEN command_list2
[ELSE command_list3]
FI
```

command_list A list of commands. Commands may be entered either one per line or several per line separated by semicolons. A command list can be null.

Description:

The IF-THEN-ELSE-FI control structure lets you control execution of a command file. The control structure can be entered interactively, but is more useful in a command file. The ELSE branch is optional.

The return status of the last command in the command list for IF determines which branch of the IF structure is executed. If it is zero (the command was successful), CI executes the THEN branch. If it is non-zero (the command was unsuccessful), CI executes the ELSE branch, if one exists, or FI, which terminates the IF control structure.

CI determines the end of a command list to be the CI command before the next expected control structure command. For example, the command list for IF ends when CI reaches THEN.

An IF-THEN-ELSE-FI control structure can be nested in either another IF-THEN-ELSE-FI or a WHILE-DO-DONE control structure.

You must end the IF-THEN-ELSE-FI control structure with FI; otherwise, CI does not recognize that it is finished and continues to process succeeding commands as though they were part of the THEN or ELSE command list. Therefore, if an IF-THEN-ELSE-FI control structure was just executed and CI is not executing commands as expected, check to see if you entered an FI command to terminate the control structure.

Examples:

The following interactive IF-THEN-ELSE-FI control structure copies file TEST, if it exists, to another directory or creates file TEST if it does not exist:

```
CI> if dl test; then co test /junk/@; else edit test; fi
```

The following command file compiles a FORTRAN source file. If successful, a library is created from the relocatable, and the intermediate files created during the merging and indexing of the library are purged.

```
IF ftn7x general_stuff.ftn - -
THEN
   * Merge general_stuff
   pu general_stuff.merg
   merge general_stuff.cmd general_stuff.merg
   *
   * Index the merged file
   lindx general_stuff.merg general_stuff.lib
   *
   * Clean up
   pu general_stuff.merg
   pu general_stuff.lst
   pu general_stuff.rel
   FI
```

IN (Initialize Disk Volume)

Purpose: Prepares a blank disk volume for use in the system.

Syntax: `IN` *lu* [*blocks* `[OK]`]

lu The LU number of the disk volume to be initialized.

blocks Specifies the number of blocks at the beginning of the disk to be reserved. A negative number specifies the number of 128-block "chunks" to reserve. For example:

```
CI> in,12,300        reserves 300 blocks
CI> in,12,-300       reserves 38400 blocks (300 * 128)
```

These blocks will not be used by the file system and can be set aside for user software. The default is no reserved space.

`OK` The optional parameter that suppresses the user prompt, indicating that the command should be executed as entered.

Description:

This command is used to clear a disk volume, eliminating all its files. Before reinitializing a disk volume with files on it, a prompt is displayed so that you can confirm your intent. A yes response must be entered to start the process. The OK parameter can be used to suppress this prompt. After the disk volume is initialized, it will be mounted to the file system.

The number of reserved blocks specified will be rounded up, if necessary, to an even multiple of the number of blocks per bit defined for the bit map for the volume. The FREES utility may be used to display the actual number of blocks reserved.

Only a System Manager can initialize a disk volume.

To initialize a disk volume for use with FMGR files, you must run FMGR and use the FMGR IN command. Refer to the FMGR description in the *RTE-6/VM Terminal User's Reference Manual* for details.

IS (Compare Strings or Numbers)

Purpose: Compares two character strings or numbers.

Syntax: `IS` *string1* *<rel operator>* *string2* `[`*option*`]`

string1 A numeric or character string.

rel operator Relational operator indicating the relation being tested. The two sets of operators recognized are as follows:

`=`	or	`EQ`	Equal to
`<>`	or	`NE`	Not equal to
`<`	or	`LT`	Less than
`<=`	or	`LE`	Less than or equal to
`>`	or	`GT`	Greater than
`>=`	or	`GE`	Greater than or equal to

string2 A numeric or character string.

option Specifies special comparison instructions. Possible values are:

`-i` Integer comparison. A suffix of B following *string1* or *string2* in either upper or lowercase indicates an octal value. A leading minus (−) sign is accepted for decimal values.

`-a` Do not fold alphabetic characters before comparison.

Description:

IS compares two strings either with an ASCII comparison or as integers after converting both strings to integers. The ASCII comparison is normally performed with alphabetic characters folded to uppercase. A shorter string is extended with blanks before the comparison is made.

IS is most useful when used in either the IF-THEN-ELSE-FI or WHILE-DO-DONE control structures.

IS returns the following status values in $RETURN1:

0	Relation is true
1	Relation is false
2	Relational operator missing or invalid
3	Option not recognized
4	Non-digit appears with −i option in effect

Examples:

```
CI> is 1024 eq 2000B -i       (The two strings are compared as integers)
```

```
CI> echo $return1             (Display the result)
0                             (The two numbers are equal)
```

```
IF is $wd ne /system/test     (IS compares $WD with a specified working
THEN wd /system/test          directory; $WD is changed if the comparison
FI                            is TRUE.)
```

IT (Interval Timer)*

Purpose: Sets execution time and interval of repetition when a program is scheduled with the ON command. Places a program into the time list.

Syntax: `IT` *program* [*res* [*mpt* [*hr* [*min* [*sec* [*ms*]]]]]]

To take a program out of the time list:

`IT` *program*

program Name of the program to be placed in the time list.

res Time interval resolution:

1	tens of milliseconds
2	seconds
3	minutes
4	hours

mpt Multiplier used in conjunction with time interval resolution value. Can be in the range of 0 to 4095. If 0 is specified, the program runs only once.

hr min
sec ms Optional parameters setting the initial time in terms of hour, minute, second, and tens of milliseconds. Default for any parameter is zero (0).

Description:

The IT command is identical to the SYSTEM IT command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

LI (List Files)

Purpose: Lists files in paged format to the terminal.

Syntax: `LI [–`*flags*`]` *filemask*

flags Specify output format. The options are described below.

filemask A file descriptor mask for the files to be listed.

Description:

After listing each screenful ("page") of the file, a 'More...' prompt is given, which lets you read that page before moving on to the next. Additionally, you may enter commands at the prompt to skip forward or review backward. Commands and options are described in the sections that follow.

The "More..." prompt by default also shows the percentage of lines in the file that were viewed thus far (that is, the fraction of the total number of lines in the file that the current line represents) if this is known. For FMGR and device files this is not known until the end of the file is reached. The "%" command (moves to a percentage through the file) is still legal for these files, but LI first reads to the EOF so that the percentage can be found.

LI is loaded as a VMA program by default. You may also load the LI program as an EMA program or as a non-EMA/VMA program by following the instructions given in the LI.LOD file. To load LI serially reusable as a memory-resident program in a memory-based system, you must load LI as an EMA or as a non-EMA/VMA program. If LI is linked as a non-EMA/VMA program, certain features of LI are not available; these features are noted as "requires VMA" in the following LI command descriptions.

LI Flags

Flags are strings of characters preceded by a dash (–) and can be entered together, as

```
-nhx
```

or separately, as

```
-n -h -x
```

When a flag uses the "next parameter" as an argument, the next unconsumed parameter is consumed as the argument; thus

```
-s 10 -e 15
```

and

```
-se 10 15
```

both set the starting and ending lines to 10 and 15, respectively.

Table 5-3 summarizes the options.

Table 5-3. LI Flags Summary

Flag	Description
A	Lists ASCII text (defaults for file types 0, 3, and 4).
W	Lists octal words (defaults for all other file types).
O	Lists octal bytes.
I	Lists signed integer words.
B	Lists binary words.
H	Lists hexadecimal bytes.
D	Lists ASCII text with display functions around special characters.
N	Lists line/record numbers.
S *ln*	Sets starting line to list at *ln*.
E *ln*	Sets ending line to list at *ln*.
X	Suppresses prompting at the end of the file; quits listing the file when the EOF is reached.
$	Always prompt at the end of file, even if the file is less than one page long.
M	Folds long lines. Lines longer than 79 characters are treated as multiple lines for pagination.
T	Truncates trailing blanks on text listings.
F	Forces type 1 access, lists blocks in octal words by default.
C	File has FORTRAN-style carriage control characters in column 1; LI by default sets the honesty bit so printer drivers do not treat column 1 as carriage control.
Q	Quiets file access; does not record access time.
L *fl*	Diverts listings to file *fl*. *fl* may be preceded by a tilde (~) to overlay an existing file, or a plus (+) to append to an existing file.
Y	Lists each file that matches <filemask> without asking.
R *rsz*	Sets maximum record size, if more than 512 characters are wanted.
pgsz	Sets number of lines per page to *pgsz* (1..32767). If *pgsz* is zero, does not paginate (lists without prompting).
> /*cmds*/	Executes initial command string *cmds* at the start of each file. For example, –> /’start’1+s’end’1–e1.l/ sets the bounds of the file to the lines between the occurances of “start” and “end” and starts listing from the top bound. The delimiter surrounding *cmds* may be any character except a space or comma. Also, backquotes (‘‘) should surround the string to keep CI from inserting commas, and so on.
P /*str*/	Redefines the ‘More...’ prompt. *str* is a string of characters delimited by any character except a space or comma. Backquotes (‘‘) should also surround the string to keep CI from inserting commas, and so on. Within *str*, the following string substitutions occur: %f file name %l current line numbers %p percentage through the file %w window or page number %% percent

LI Commands

Commands entered at the 'More...' or 'End...' prompts can be preceded by a number from 1..2147483647 (this value is referred to as *n* in the LI command summary below). The listing commands are summarized in Table 5-4.

Trailing arguments like *m* and *rex* are prompted for interactively. In the ">" runstring flag, they are entered directly after the command, as in

```
   li -> /f!zow!ka1.ua/ yow
```

which finds string "zow" (delimited by exclamation points (!)) in file "yow", marks that line with "a", goes to line 1, and lists yow until the line marked with "a" is encountered.

When more than one file is selected, LI prompts with

```
   File:  file, list? [Y]
```

before listing each file (the "y" option turns this prompt off). When prompted, you may enter one of the responses shown in Table 5-5.

Regular expressions are the same as those in EDIT/1000; see the documentation for EDIT/1000 for use and examples. In brief, the expressions are shown in Table 5-6.

Backward movement is allowed for device (as opposed to disk) files, but LI cannot back up beyond the number of lines that fit inside an internal buffer.

Table 5-4. LI Commands Summary

Command	Description
Space or L	Lists the next page or the next *n* lines if given.
Return	Lists the rest of the file or goes to line *n*.
A or Q	Aborts list. 'A' moves to the next masked file. 'Q' quits the entire listing.
+	Lists the next line or skips forward *n* lines.
–	Skips backward 1 line or *n* lines from the top line in the window.
B	Skips backward 1 page or *n* pages from the top line in the window.
G or .	Goes to line *n*. 'G' lists a page. A period (.) lists 1 line.
$	Lists the last window.
%	Goes to a line that is *n* percent through the file.
'*rex*'	Searches for the next occurrence of regular expression *rex* from the current window or from line *n*. A null string searches for the last string entered. The trailing quote is not used in interactive mode; simply terminate the pattern with <return>.
F/*rex*/	Same as '*rex*' but with user-defined delimiters for startup commands. In interactive mode, the delimiters are not used; the pattern is terminated with <return>. The delimiters may be any character except space or comma. The delimiters cannot be the same as those used to surround the startup commands string.

Table 5-4. LI Commands Summary (continued)

Command	Description
‘*rex*‘	Searches backward for regular expression *rex* from the current window or from line *n*. A null string searches for the last string entered. The trailing quote is not used in interactive mode; simply terminate the pattern with <return>.
@/*rex*/	Show all lines containing pattern from the current window or from line *n*. The delimiters may be any character except space or comma.
K*m*	Marks top window line or line *n* with *m* which must be an alphabetic character from A to Z.
:*m*	Goes to line marked with character *m* and lists *n* lines.
U*m*	Lists until the line marked with *m* is encountered; lists no more than *n* lines.
P	Sets page size to *n*, if given, and lists a page.
Oc	Toggles or resets the setting of runstring flags, including listing modes (a, b, h, i, o, w).
S	Sets the starting file line to the window top. LI will not back up farther than this line.
E	Sets the ending file line to the window bottom. LI will not advance past this line.
#*f*	Moves to file number *n*, if given, or prompts for file number *f* (requires VMA).
#+[*i*]	Moves forward *i* selected files, or 1 file, if not given (requires VMA).
#−[*i*]	Moves backward *i* selected files, or 1 file, if not given (requires VMA).
#?[*f*]	Shows a window of selected files starting at file *f*, or around the current file, if not given (requires VMA).
=	Displays the file name and current line number on the screen.
N*file*	Adds a new file name to the list of files to be displayed (requires VMA) and moves to this new file.
R	Removes (purges) the file being listed.
? or H	Displays help information.
Z	Suspends LI operation. You can restart with the system GO command.

Table 5-5. LI Responses

Response	Meaning
Y or <*space*> or <*return*>	List the named file.
N	Do not list the file.
S	List the file and set the "y" option (to suppress the prompt).
A	Do not list the file; abort the listing.
#	Show selected files or move to another file (see description of the # command in Table 5-4).
R	Remove (purge) this file.

Table 5-6. Expressions Summary

Expression	Meaning
.	Matches any character.
@	Matches any character zero or more times (same as '.*').
^x	Anchors the pattern to the beginning of the line.
x$	Anchors the pattern to the end of the line.
[ai−k]	Matches any of the characters 'a', 'i', 'j', and 'k'.
[^ai−k]	Matches any character except 'a', 'i', 'j', and 'k'.
x*	Matches zero or more occurrences of pattern x.
x+	Matches one or more occurrences of pattern x.
x<5>	Matches 5 repetitions of pattern x.
a:b	Matches a word boundary between patterns a and b.
*	Matches the character '*'.

LU (Display/Modify Device Assignment)*

Purpose: Displays information associated with a device specified by its LU number. Selected status can be modified by the System Manager using this command.

Syntax: `LU` *lu* `[`*eqt* `[`*subchannel*`]]`

lu Specifies the system LU for which information or reassignment is desired.

eqt Used by the System Manager only. Assigns the EQT entry number to the LU specified. If 0 is specified, LU becomes the bit bucket.

subchannel Used by the System Manager only. Assigns subchannel number (0 to 63) to specified LU.

Description:

The LU command is identical to the SYSTEM LU command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

MC (Mount Disk Volume)

Purpose: Mounts a disk volume and makes its contents available to the system.

Syntax: `MC` *lu*

lu The LU number of the disk volume to be mounted. Must be a positive number.

Description:

The MC command mounts a disk volume to the file system, thus making it accessible to users of the file system.

If the disk volume has a valid FMP or FMGR directory, the volume is mounted; otherwise, you are prompted to confirm that the volume should be initialized. This is to avoid accidental corruption of volumes that are not FMP or FMGR types (special backup utility volumes, for example).

The MC command does not place reserved blocks at the beginning of the volume. Use the IN command if reserved blocks are required.

There is no significance to the order in which disk volumes are mounted, unless there are duplicate global directory names on two or more volumes. If a global directory on the newly mounted disk volume has the same name as a previously mounted global directory, the new directory is inaccessible. To access the new directory you must rename the previously mounted directory, then dismount the new disk volume and remount it.

This command should NOT be used to mount FMGR cartridges. Use the FMGR MC and DC commands to manage FMGR cartridges. Refer to the *RTE-6/VM Terminal User's Reference Manual* for details.

MO (Move Files)

Purpose: Moves files from one directory to another on a given disk volume and renames files.

Syntax: `MO` *file1 file2*

file1 The source file descriptor. (Refer to the CR command syntax description for the definition of file descriptor.) May be masked to move a group of files. (Refer to the "File Masks" section in the DL command discussion in this chapter for the mask syntax.)

file2 The destination file descriptor. The file name may be defaulted to that of the source file name. May be masked to allow the system to generate destination names.

Description:

The MO command can be used to move a group of files from one directory to another. Masking the file1 parameter allows matches of a number of files. If a wildcard character is used in the file name field of file1, an appropriate destination mask must be used to default the destination file names.

Note that this command is very similar to the CO command. It uses the same syntax and performs nearly the same operation, but with the following important differences:

- The files are MOVED, not COPIED. This means that after after you use the MO command, the file will no longer be where it used to be.
- The file contents are not moved, only the directory entry is moved. This is much faster, particularly for large files, and more reliable since the data is not altered.
- Files cannot be moved across disk volumes. This is because the data is not moved, and the data must be on the same volume as the directory entry. If you wish to move files across volumes, the CO command can be used with the P option (purge source after copy) to move the files.

Examples:

```
   CI> mo @.@.a-8306 /backup/archive/@.@
```

This example moves all the files that were not accessed since June 1983 into the archive subdirectory of the backup directory.

The following example causes the subdirectory to become a global directory. A file that formerly had the name /MYGLOBAL/MYSUBDIRECTORY/MYFILE now has the name /MYNEWGLOBAL/MYFILE. The file data has not changed, nor has the directory data in MYNEWGLOBAL.

```
   CI> mo /myglobal/mysubdirectory.dir /mynewglobal
```

OF (Stop/Remove Program)*

Purpose: Stops a scheduled program or releases a program ID segment.

Syntax: OF [*prog* [*pram*]]

prog The program name, up to five characters, with an optional session identifier.

pram An optional parameter used to specify the action to be taken. Possible values are:

0	remove from time list (default)
1	terminate immediately; release disk tracks
8	terminate immediately and remove ID segment
ID	same as 8

Description:

The System Manager can use this command to remove any program if the need arises. General users can only remove non-system programs in their own session and other sessions with the same user name. This command is identical to the SYSTEM OF command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

ON (Schedule Program)*

Purpose: Schedules a program for execution. Up to five parameters and the command string may be passed to the program.

Syntax: `ON [`*program* `[NO [`*pram*5*`]]]`

program Specifies the name of a program to be scheduled.

`NO(w)` Schedules immediately a program that is normally scheduled by the system clock. If the program is placed in the time list, but not scheduled for immediate execution, this parameter and its preceding comma are omitted. It may be entered as NOW.

*pram*5* Up to five parameters may be passed to the program when it is scheduled.

Description:

The ON command is identical to the SYSTEM ON command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

OWNER (Display/Change Owner)

Purpose: Displays or changes the owner of a directory or a subdirectory.

Syntax: `OWNER` *directory* [*newOwner*]

or

`OWNER` *lu*`V` [*newOwner*]

directory The name of the directory or subdirectory. No wildcard characters are allowed.

lu`V` A CI volume number followed by the character “V”.

newOwner The name of the new owner. This is needed only if a change is required. If omitted, the owner of the directory or subdirectory specified is displayed.

The *newOwner* parameter must be a name usable for logon, that is, there must be a user account with that name on the system. The group account name in the *newOwner* parameter is made the associated group of the directory or volume. If a group account name is not specified in the *newOwner* parameter, GENERAL is assumed to be the group account.

Description:

This command assigns or displays ownership of the named directory or CI volume. Only the current owner can assign ownership. Ownership is associated with volumes, directories, and subdirectories, but not with the individual files. The directory cannot be on a remote system (if DS is used) or specified with an account.

When the owner is changed, the current user is no longer the owner of that directory; thus the current user is unable to change the owner back. This change can also make all subdirectories of this directory inaccessible to the original owner. Note that the ownership of subdirectories is not changed when the ownership of the directory they are in is changed.

Ownership is maintained through owner numbers, rather than owner names, so ownership remains correct even if the user’s logon name is changed with the USERS program. Note that if a removable disk is moved to another system with different user accounts, ownership will not be correct.

PATH (Display/Modify UDSP)

Purpose: Allows you to display or modify a User-Definable Directory Search Path (UDSP).

Syntax: `PATH [-E]`

`PATH [-E] [-N:`*n*`]` *udspnum* `[`*dirname1* `[`*dirname2* `[...[`*dirnameN*`]]]]`

`PATH[ -E] -F` *file* | *lu*

`-E` Turn off echo; non-error messages are not displayed. This is used when echoing is not desired from a command file or when information is desired only in the return parameters and is not to be displayed.

`-N:`*n* Display or modify the specified entry. Set *n* equal to 1 for UDSP #0 (home directory); otherwise, set *n* to a value between 1 and the UDSP depth.

udspnum Specifies the UDSP number. The values for *udspnum* are as follows:

- `0` Home directory.
- *n* UDSP number between one and the number of UDSPs defined for this session (maximum is eight).
- `-A` All UDSPs defined for current session.

dirname Specifies the directory name. The following special characters can be used:

- `.` Use the working directory that is current when the UDSP is referenced.
- `!` Delete this UDSP or entry; this character must be the only *dirname* in the command line.

−F *file* | *lu* Indicates that the commands will be input from the specified file or LU.

Description:

The first format of the PATH command, without the −E parameter, sets the $RETURN variables and displays current UDSP information: the total number of UDSPs defined for the session, the depth (number or entries per UDSP), and the next available UDSP. If the −E parameter is specified, the $RETURN variables are set, but no information is displayed.

The second format displays or defines a specific UDSP or a specific entry of a UDSP and sets the $RETURN variables. If the −E parameter is specified, the specific UDSP or specific entry of a UDSP is defined and/or the $RETURN variables are set, but no information is displayed.

The third format indicates that the specified file or LU contains commands to define or display the UDSPs. Specifying the −E parameter inhibits echoing of commands from the specified file. The file or LU can contain one or more command lines. The syntax for a command line is as follows:

`[-N:`*n*`]` *udspnum* `[`*dirname1* `[`*dirname2* `[...[`*dirnameN*`]]]]`

A unique set of UDSPs is associated with your session. The number of UDSPs and the depth (number of entries) for each UDSP are set when your user account is created or modified. You can have from zero through eight separate UDSPs; each UDSP has the same depth.

At logon, all UDSPs are undefined. You must issue a separate PATH command for each UDSP you want to define. The UDSPs created by the PATH command are valid only for the current session. By placing PATH commands in your HELLO file, you ensure that the UDSPs are defined the same each time you log on.

Eight UDSPs are available; the first three have the following special meanings:

UDSP #0 Represents the home directory and has a predefined depth of one.

UDSP #1 Used by the RU command. Whenever you enter an RU command, implied or explicit, without specifying any directory information, the search path defined for UDSP #1 is used. If you do not define UDSP #1, the default search sequence is used. We recommend that you always make /PROGRAMS the last entry in the search path. Note that if you define UDSP #1, your executable file must have .RUN as the file type extension.

UDSP #2 Used by the TR command. Whenever you enter a TR command, implied or explicit, without specifying any directory information, the search path defined for UDSP #2 is used. If you do not define UDSP #2, the default search sequence is used. We recommend that you always make /CMDFILES the last entry in the search path. Note that if you define UDSP #2, your command file must have .CMD as the file type extension.

UDSP #3 Used by some subsystems to find library files.

UDSPs #4 through #8 can be used for your application programs. See the description of FmpOpen in Chapter 6 and the description of the #n directory specifier in Chapter 3 in this manual.

Only CI hierarchical directories can be entered as part of a UDSP; FMGR cartridges cannot be specified. However, if a period (.) is defined as a UDSP entry and the working directory is set to zero before the UDSP is referenced, all mounted FMGR cartridges are searched.

PATH returns the following values in the five $RETURN variables:

$RETURN1 If zero, the command was successful; otherwise, an FMP error code is returned. THEN and ELSE test this parameter.

$RETURN2 Number of UDSPs defined for this account.

$RETURN3 Depth value.

$RETURN4 Next available UDSP (first UDSP that is undefined).

$RETURN5 Zero (not used).

The name of the directory is returned in $RETURN_S when a specific entry (−N:*n* option) or the home directory (PATH 0) is requested.

Examples:

```
   CI> path                        (Display current UDSP information)

   CI> path 1                      (Display UDSP #1)

   CI> path -a                     (Display all UDSPs)

   CI> path 0 /mine                (Set home directory to /MINE)

   CI> path 2 . /mine/cmdfiles /cmdfiles
                                   (Set UDSP #2 to the following:
                                      (1) current working directory
                                      (2) /MINE/CMDFILES
                                      (3) /CMDFILES)

   CI> path -e -f setpath.cmd      (Read PATH commands from the file
                                   SETPATH.CMD without echoing messages
                                   where SETPATH.CMD contains the
                                   following command lines to set UDSPs
                                   #0, #1, and #2:

                                      0 /mine
                                      1 . /mine/progs /programs
                                      2 . /mine/cmds /cmdfiles)

   CI> path -n:3 1                 (Display the third entry of UDSP #1)

   CI> path -n:1 2 /groups/cmds    (Set the first entry of UDSP #2)

   CI> path 3 !                    (Delete all entries of UDSP #3)

   CI> path -n:2 4 !               (Delete the second entry of UDSP #4)

   CI> path -e -n:2 3              (Return the contents of the second entry of
                                   UDSP #3 in $RETURN_S without echoing
                                   the name to the terminal)

   CI> path -e 0                   (Return the name of the home directory
   CI> wd $return_s +s             without echoing it and then set the
                                   working directory to the home directory)
```

POLL (Polling Function)

Purpose: Executes a specified CI command at a specified time interval.

Syntax: `POLL` *interval*|`OFF` *command*

interval|`OFF` if a number, it is the approximate number of minutes between executions of the poll command.

if OFF, the poll function is turned off.

command is any CI command/program to be executed at the poll interval.

Description:

Each time CI prepares to issue a prompt, it first checks to see if the current time minus the base time is greater than the poll interval. If it is, CI executes the poll command, sets the base time for the next poll interval to the current time, and issues the CI prompt.

The command to be executed and the poll interval are stored in CI variables $POLL and $POLLINT, respectively, so that you can see them by doing a SET command in CI.

Examples:

`CI> poll 1 dl`	(Execute the DL command and set the poll interval to one minute.)
`CI> poll`	(Execute previously set poll command and reset the base time.)
`CI> poll 7`	(Execute previously set poll command and reset the poll interval to seven minutes.)
`CI> poll off`	(Turn off the polling function.)

Notes:

1. $POLL can be altered with the SET command. This has the effect that the new command will be executed the next time the poll interval is exceeded.
2. $POLLINT cannot be altered with the SET command.
3. $POLL and/or $POLLINT can be deleted with the UNSET command. If either or both variables are deleted, the polling function is turned off.

PR (Change Program Priority)*

Purpose: Changes priority of a restored program. It can also be used to display the priority of a program.

Syntax: `PR` *prog priority*

prog	Program name, up to five characters, with an optional session identifier.
priority	Range is between 1 and 32767.

Description:

The PR command is identical to the SYSTEM PR command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

PROT (Display/Change Protection)

Purpose: Displays or changes the protection status of a file, directory, or volume.

Syntax: `PROT` *fileMask* `[`*newProtection*`]`

or

`PROT` *lu*`V` `[`*newProtection*`]`

fileMask	A file mask that includes all fields of the file descriptor and a qualifier. Refer to the "File Masks" section in Chapter 3 for a full description of file masking.
lu`V`	Describes a CI volume LU to display or change (for example, 15V).
newProtection	Defines the new protection status for the owner, members of the owner's group, and others.

The syntax for the newProtection parameter is:

owner `[`*/group*`]` */others*

The slash is a required delimiter. The protection values are:

`R`	=	allow read access
`W`	=	allow write access

If both R and W are specified, they may be given in either order. If no protection value is given in a particular position, it disallows all access. If the group protection is not specified, it will remain unchanged.

As an alternative to the R and W symbols, a set of default symbols may be specified. These symbols will allow the current protection values to be transferred into the new protection setting. The symbols are:

`U` = user (owner)	Place current owner protection here.
`G` = group	Place current group protection here.
`S` = system (others)	Place the current other protection here.

Description:

If new protection is not specified, this command displays the current protection on the files that match the mask or on the CI volume. If new protection is specified, all files that match the mask or the CI volume will have their protection changed to the new protection.

When the current protection is displayed for a file mask, PROT actually executes the command “DL <mask> P”, which shows the current protection for all files matching the mask (volume protection is displayed without DL).

When a global directory is created, the owner is set to the creator and the default protection is RW/R/R. When a CI volume is initialized, the owner is set to the one doing the initialization and the default protection is set to RW/RW/RW.

To change protection on a file or volume, the user must be the owner of the directory on which the file resides or of the volume itself. Protection of a CI volume restricts the displaying, creating, and purging of global directories on that volume.

Examples:

`CI> prot /libraries/@.@`	(See protection for all files in /LIBRARIES)
`CI> prot message rw/rw/rw`	(Allow full access by everyone)
`CI> prot groupbox rw/rw/`	(Allow access only to owner and group members)
`CI> prot oldfile rw/r`	(Set access but leave previous group value)
`CI> prot myfile rw//`	(Only the owner can access the file)
`CI> prot myfile /rw/rw`	(Only the owner cannot access the file)
`CI> prot safe //`	(Noone (except superusers) can access the file)
`CI> prot 15v`	(Display the protection for CI volume 15)
`CI> prot 15v rw/rw/r`	(Allow the owner and group members only to create global directories on CI volume 15)

Assume protection on FILE is currently rw/w/r:

`CI> prot file r/g/s`	(Change owner access only (result is R/W/R))
`CI> prot file u/g/rw`	(Change others access only (result is RW/W/RW))
`CI> prot file u/s/s`	(Do not change owner and others access, but change group access to the same as others (result is RW/R/R))
`CI> prot file s/u/g`	(Shuffle protection around (result is R/RW/W))

PU (Purge Files)

Purpose: Purges files.

Syntax: `PU` *mask* `[OK]`

mask A file mask that may include all fields of the file descriptor and a qualifier. Refer to the “File Masks” section in Chapter 3 for a full description of file masks.

`OK` The optional parameter that instructs the program to purge the specified files without prompting.

Description:

A wildcard purge is one in which a mask is used to specify one or more files to be purged. It is a powerful capability, but should be used with great care in order to avoid purging files that you meant to keep.

If a mask is not specified, no files will be purged.

If the file mask you enter specifies a single file, only that file is purged and the following message is displayed:

```
Purging file_descriptor
```

FMGR files with a security code must be purged individually with the security code specified.

When a group of files is to be purged, the program provides some checking to make sure you really want to purge all the files by prompting you interactively to confirm each file that matches the mask:

```
Purging file_descriptor (Yes, No, Abort, Stop Asking) ? [Y]
```

The program steps through the files, prompting you to respond for each file. This allows you to confirm the file selection or change your mind before the purge is done by entering one of the responses shown in Table 5-7 below:

Table 5-7. PU Responses

Response	Action
Y or <cr>	Purge the file named.
N	Skip this file.
A	Abort the purge.
S	Purge all the files that match the mask (you will not be prompted again).

You can avoid having the program perform this checking by using the OK parameter, which indicates that the purge is set up as intended. OK causes all the files that match the mask to be purged without prompting or any other intervention. The program displays the file names on the log device as the files are purged, but no further confirmation is required.

If the input device is not an interactive device and the OK parameter is not specified, wildcard purges will not be executed.

To purge a file, you must have write access to the directory that contains it. The file must not be an active type 6 file, the system swap file, an opened file, or a directory containing files.

The PU command can be used to purge an empty directory. Note that the format of the command to purge a global directory is "PU /GLOBAL". The command "PU ::GLOBAL" or "PU /GLOBAL/" will purge all the files on the GLOBAL directory, but not the directory itself. In this case, the form /GLOBAL is not the same as ::GLOBAL and does not produce the desired results.

If there are non-empty subdirectories under the GLOBAL directory, you can purge them and their contents by repeatedly entering the following command:

```
CI> pu /global/@.@.s
```

Examples:

```
CI> pu /goal/file1
```
(Purge FILE1 in directory GOAL)

```
CI> pu @.temp ok
```
(Purge all files in working directory with file type extension .TEMP)

```
CI> pu /joe/
```
(Purge all files in global directory JOE)

```
CI> pu /joe
```
(Purge global directory JOE)

```
CI> pu /test/two.dir
```
(Purge subdirectory TWO. Note that the file type extension .DIR is required here to avoid confusion with files named TWO)

PWD (Display Working Directory)

Purpose: Displays present working directory.

Syntax: `PWD`

Description:

The PWD command displays the current working directory.

QU (Timeslice Quantum)*

Purpose: Displays examination of the current system timeslice quantum and the program priority level at which timeslicing begins. The System Manager may change the timeslice parameters with this command.

Syntax: `QU [`*quantum* `[`*limit*`]]`

quantum Specifies the new system slice quantum; value must be in the range between 0 and 32767 milliseconds. Default is 1500.

limit Specifies the priority level at which timeslicing begins; default is 50. All programs of equal or lower priority (higher priority number) will be timesliced.

Description:

The QU command is identical to the SYSTEM QU command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

RETURN (Return from Command File)

Purpose: Returns to previous level of command file nesting or to interactive mode.

Syntax: `RETURN[,`*return1*`[,`*return2*`[,`*return3*`[,`*return4*`[,`*return5*`[,`*return_s*`]]]]]]`

return1 Integer return status indicating success or failure of command file. Zero means success; nonzero means failure. If omitted, return1 is set to zero.

return2−5 Four integer values made available to return additional status information. Each omitted parameter is set to zero.

return_s A string, up to 80 characters; if omitted, it is set to null.

Description:

RETURN lets you exit from a command file at any point. Note that LU 1 (your terminal) is treated as a command file. If RETURN is entered interactively when a parent program is waiting, CI returns to the parent program; otherwise, the command is ignored.

All the parameters for the RETURN command are position dependent; therefore, you must include commas to mark the positions of any omitted parameters.

The values returned are available in the predefined variables $RETURN1 through $RETURN5, and $RETURN_S.

You can include a RETURN command anywhere in IF-THEN-ELSE-FI or WHILE-DO-DONE control structures. CI always executes a RETURN command when the end of a command file is reached, whether or not you included the command at the end of the file.

Examples:

```
CI> return 0,2,3,4,5, ‘Command file successful‘
```

(Exits a command file and specifies 5 integer values and a string)

```
CI> return
```

(Exits a command file using the default return values; $RETURN1 through $RETURN5 are set to 0, and $RETURN_S is set to a null string)

```
CI> return,,,,,,‘Successful Completion‘
```

(Exits a command file, returning a value in only the variable $RETURN_S. $RETURN1 through $RETURN5 are set to the default value of zero)

RN (Rename File, Directory, or Subdirectory)

Purpose: Renames a file, a directory, or a subdirectory.

Syntax: `RN` *file1 file2*

file1 The source file descriptor; can be masked to operate on more than one file (refer to the “File Masks” section in Chapter 3).

file2 The destination file descriptor; can be masked to allow the system to generate destination names.

Description:

RN changes the name, file type extension, or any combination of the above of file1 to those for *file2*. The new name must not already exist on the directory. You must have write access to the directory.

Directories and subdirectories can be renamed. This command does not move files into a different directory. If the directory field of the destination file name is blank, the source directory is used. If source and destination directories are different, an error message is displayed. In this case, use the MO command.

Examples:

`CI> rn foo joe`	(Rename file FOO on the working directory to JOE)
`CI> rn foo.txt @.ftn`	(Change the file type extension of FOO from .TXT to .FTN)
`CI> rn @.src @.ftn`	(Change all files with file type extension .SRC to .FTN)

RP (Restore Program File)

Purpose: Establishes a permanent program ID segment.

Syntax: `RP` *filename* [*prog*] [*options*]

filename The file descriptor of the type 6 program file to be restored. The first five characters of the file name are used as the program name, unless the optional parameter is specified.

prog The new program name to be used instead of the file name, up to five characters.

options A character string that contains “C”, “P”, “T”, or both “C” and “P” to select the following options:

`C` Create a clone name if the specified or assigned name is already assigned to an RP’d program. The program is cloned if:

- There is an active program with that name that is not a system utility.
- There is a dormant, temporary program with that name that is not a system utility.

`P` Create a permanent program ID segment that will not be released when the program terminates.

`T` Create a temporary program ID segment that will be released when the program terminates.

Description:

The RP command sets up an ID segment for the type 6 program file specified. This restores the program ID segment, making it available for use by program control commands and subroutines that require a restored program; for example, the WS, VS, and SZ commands. If a new name is not specified, it is derived from the file name. Refer to the RU command description for details on searching for the correct file to restore.

The RP’d program remains associated with the session that RP’d it and will be removed when the user logs off.

RS (Restart Session Progenitor)

Purpose: Aborts and reschedules the session progenitor

Syntax: `RS`

Description:

The RS command can be used to restart a session progenitor (usually CI) that is not executing properly (for example, when CI becomes hung on a downed device). This command is especially useful if you OF the progenitor, as the session will terminate if it is the only program in the session.

The RS command is available only in CM and break mode.

RU (Run Program)*

Purpose: Immediately schedules a program for execution and waits for its completion.

Syntax: `[RU]` *prog|file* `[`*pram*5*`]`

`RU` An optional parameter that is only required if the program name is two characters that can be interpreted as a CI command or if the *prog|file* parameter can be confused with a command file (see sections on TR command and predefined variables).

prog|file The 5-character program name or a file descriptor that identifies a type 6 file. Including the optional ":IH" in the program name (for example, PROG1:IH) inhibits cloning of the program.

*pram*5* The parameters to be passed to the program. The maximum runstring length, including the implied RU and delimiter, is 256 characters. This can be five numeric parameters or a character string.

Description:

If the program is not restored, CI restores it and frees the program ID segment after it finishes running. CI modifies the program name if necessary to make it unique when it restores the program. The last two characters will be changed to .A, .B, and so forth.

We recommend that you always use the .RUN file type extension in the program file name.

If you are executing more program files than command files, set the predefined variable $RU_FIRST to TRUE. When $RU_FIRST is set to TRUE, CI assumes that any file name entered without a CI command or file type extension is a program file and immediately attempts to execute the file as a program.

When you enter an implied or explicit RU command, the procedure described below is used to find the program file.

1. If a directory is specified, this directory is searched for the file. If the file is found, it is restored. If the file is not found and a file type extension was not specified, .RUN is assumed, and the directory is searched again. If the file is still not found, an error is returned.

2. If no directory information is given, the following occurs:

   a. If a program with the specified or assigned name is already restored and can be cloned, this program is cloned. If the program cannot be cloned and is dormant, the original program is used.

   b. If the program was not restored, a search is made for the program file. If UDSP # 1 is defined, a default file type extension of .RUN is assumed and the search path defined by UDSP #1 is used to find the file. If the file is not found, an error is returned. (Refer to Chapter 3 for a description of UDSPs.)

c. If UDSP #1 is not defined, the following default search sequence is used:

- The current working directory is searched. If the file is not found, a default file type extension of .RUN is assumed and the working directory is searched again.
- If you do not have a working directory, all mounted FMGR cartridges are searched.
- If the file is still not found, global directory PROGRAMS is searched, using the .RUN default file type extension. If the file is not found, an error is returned.

For example, if a working directory exists and UDSP #1 is undefined, the search sequence for program EDIT specified in "RU,EDIT" is as follows:

- Search for a restored (RP'd) EDIT.
- Search for EDIT in the working directory.
- Search for EDIT.RUN in the working directory.
- Search for EDIT.RUN in directory PROGRAMS.

If there is no working directory, the search sequence is:

- Search for a restored (RP'd) EDIT.
- Search for EDIT in FMGR disk cartridges.
- Search for EDIT.RUN in directory PROGRAMS.

Parameters passed to the program can be integer, octal, or ASCII. If an ASCII string is specified for a program that uses RMPAR, the string is parsed into two-character words that are each passed as separate parameters up to the maximum of five. Specify octal numbers by immediately following the number with the letter b; for example, 30b.

SET (Display/Define Variables)

Purpose: Displays all positional, user-defined, and predefined variables or defines a user-defined or predefined variable.

Syntax: `SET [`*variable* `=` *string*`]`

variable A string of up to 16 letters, digits, and underscores, not starting with a digit.

string A string of up to 83 characters.

Description:

CI provides variables that you can define (positional and predefined variables) and also allows you to create variables (user-defined variables). The SET command displays or defines these variables.

If a variable is not specified, all positional, user-defined, and predefined variables are displayed.

Examples:

```
CI> set filename = /mine/stuff/my_progs/test.ftn

CI> set auto_logoff = 3

CI> set greeting = ‘How are you today?‘

CI> set                                    (Display all variables)
```

SL (Display/Modify Session LU Information)*

Purpose: Displays or modifies the Session Switch Table (SST) for either a specified session LU or all session LUs.

Syntax: `SL [`*sessionlu* `[`*systemlu*`]]`

sessionlu Specifies the session LU to display or modify; if ommitted, the entire Session Switch Table (SST) is displayed.

systemlu Specifies the system LU as follows:

number = system LU to which the session LU will point.
– = removes the mapping for the given session LU.

Description:

The session LU information is always displayed on the user terminal. The display from the SL command is of the following form:

```
SLU sess=LU  # sys = E eqt S subc status
```

where:

sess is the session LU.
sys is the system LU.
eqt is the EQT to which the system LU points.
subc is the subchannel of the EQT to which the system LU points.
status is the current device status; if the device is down, this will be the character D, otherwise it will be blank

For example:

```
SLU  14=LU  # 27 = E 13 S 3 D
```

indicates that session LU 14 is mapped to system LU 27, which points to EQT 13, subchannel 3. The device is currently down.

The SST (Session Switch Table) is defined for a particular account and modified (for station configuration information) at logon time. See the *RTE-6/VM Terminal User's Reference Manual*, part number 92084-90004, for a detailed description of the SST and its purpose.

Compare the SL command with the LU command which deals only with system LUs.

Examples:

`CI> sl`	(Display entire SST)
`CI> sl 5`	(Display only the mapping for session LU 5)
`CI> sl 5 114`	(Map session LU 5 to point to system LU 114)
`CI> sl 5 -`	(Remove the mapping for session LU 5)

SS (Suspend Program)*

Purpose: Suspends an active program.

Syntax: `SS [`*prog*`]`

prog The name of an active program, session identifier optional.

Description:

The SS command places the program in operator suspend state. This is done immediately if the program is scheduled or executing. If the program is currently suspended for any reason other than an operator suspend, or if the program is dormant, the SS command is queued until the program is rescheduled. At that time the program is placed in the operator suspend state.

The SS command is similar to the EXEC 7 program suspend call.

Execution of programs suspended with the SS command may be resumed with the GO command or aborted with the OF command.

If prog is not specified and the startup program (CI or FMGR) has scheduled another program, this command is executed on the scheduled program unless it, in turn, has scheduled a program. The search continues down the program scheduling chain and the SS command is executed on the last program. The only exception is that if the last program is a protected system program, the program that scheduled it will be suspended.

The System Manager can suspend any program in the system. The general user can suspend only programs scheduled within that session.

Example:

```
CI> ss timer                (Suspend program TIMER)
```

ST (Display Program Status)*

Purpose: Displays the status of a program. Information requested can be program priority, current list, time values, or the partition number of the program currently executing. A special case is to display the name of the program occupying a specified partition.

Syntax: `ST` [*program* | *partition_#* | `0`]

program Specifies the name of the program whose status is to be displayed.

partition_# Specifies the number of a partition (1 to 64) to display the program occupying that partition. If the partition is empty, 0 is displayed. If an undefined partition number is entered, an error message “NO SUCH PROG” is displayed.

`0` Entering zero (0) displays the name of currently executing program and its partition number. If there is no program executing, 0 is displayed.

SZ (Display or Modify Program Size)*

Purpose: Displays program size information of a restored program or modifies the program size requirements.

Syntax: `SZ` *prog* [*size* [*msegSize*]]

prog The program name, up to five characters, with an optional session identifier.

size The program size in pages for non-VMA programs or the EMA size for EMA programs, not including PTE. Range is 2 ≤ *size* ≤ 1022 for EMA size.

msegSize The new MSEG size for EMA programs. Range is 1 ≤ *msegSize* ≤ 30.

Description:

This command changes the amount of memory that the specified program can use when it runs. The program must be restored with the RP command and must be dormant.

Increasing the program size will help programs that use memory at the end of their partition for buffer or table space. Such programs include EDIT, LINK, MACRO, and CI. To change the size permanently, use the LINK program.

This command is identical to the SYSTEM SZ command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

TI (Display Time)*

Purpose: Displays the system real-time clock.

Syntax: `TI`

Description:

The current system time is displayed in the following format: year, day (Julian), hour (24-hour format), minutes, and seconds.

Example:

```
CI> ti
1988 285 18 45 48
```

TM (Display or Set System Clock)*

Purpose: Displays or sets the system clock.

Syntax: `TM` [*month day year hr*`:`*min*`[:`*sec*`[`*pm*`]]]`

month	Jan to Dec
day	1 to 31
year	1976 to 2144
hr	0 to 23
min	0 to 59
sec	0 to 59
pm	Defaults to am

Description:

This command displays the system clock in the format shown in the following example:

```
   Mon Sep 27, 1988 11:37:13 am
```

Parameters can be entered as they would be printed; time can be specified in 24-hour format if desired, with or without seconds.

Only the System Manager can set the time.

Resetting the time affects programs in the time list but not programs set to run after a particular length of time. It also affects the time stamping of files, so use this command with caution.

TO (Display or Modify Device Timeout)*

Purpose: Displays or sets the timeout limit for a device.

Syntax: `TO` *eqt* `[`*interval*`]`

eqt The EQT number of the device.

interval The number of 10-milliseconds intervals to be used as the timeout value for device EQT. Value can be in the range of 0 ≤ *interval* ≤ 65534.

If *interval* equals zero, the timeout limit does not apply to this device.

Description:

The timeout value is displayed in the form:

```
TO # eqt = interval
```

or

```
INPUT ERROR                  (Indicates that the given EQT number does not exist or the
                             value entered was illegal)
```

The time base generator (TBG) generates an interrupt every 10 milliseconds. When a program sends an unbuffered I/O request to a device, the system puts the program into I/O suspension and begins counting the number of TBG interrupts. When the request is fulfilled, the program resumes execution. If, however, the number of TBG interrupts exceeds the timeout value defined, the program is put into a downed device wait state and the device may be set down. This prevents an offline or downed device from causing a program to remain I/O suspended indefinitely. When the program goes into this wait state, it can be swapped out to the disk and another program can begin execution. When the device is once again available to the system, the original program can resume execution after the device has been UP'd.

If you set a timeout value too low for a device, that device may appear to be failing, when in fact it is performing properly. If the driver times the device out before it can respond to a request, the device will appear to be downed.

To calculate the interval parameter, multiply the desired timeout value (in seconds) by 100.

When the system is rebooted, timeout values revert to those set at system generation time.

Examples:

To display the timeout value:

```
CI> to 6
TO # 6 = 500                      (5-second timeout)
```

To modify EQT 6 timeout to 10 seconds:

```
CI> to 6 1000
TO # 6 = 1000                     (New value displayed)
```

TR (Transfer to Command File)

Purpose: Transfers control to a command file.

Syntax: `[TR]` *file* [*pram*9*]

TR An optional parameter that is only required if the command filename is two characters that can be interpreted as a CI command, or if the file parameter can be confused with a program file (see the sections on the RU command and predefined variables).

file The file containing the commands. Refer to the CR command for a definition of a file descriptor.

*pram*9* One to nine parameters that are used to replace occurrences of the positional variables $1 through $9 in the command file. Defaults to zero-length strings.

Description:

A command file (also known as a transfer file) contains a sequence of CI commands. The commands are executed as if you entered them from the terminal. Command files are useful for executing command sequences repeatedly.

Command files can be nested by using the TR command in the command file. Control is returned either to the terminal or to the command file, depending on whether the TR command was issued from the terminal or another command file, when the end of the file is reached or a RETURN command is encountered. A “TR,1” command transfers control to the terminal and a “RETURN” command returns control to the file from which the “TR,1” command was issued.

Positional variables $1 through $9 can be used in command files. A parameter in the runstring is substituted wherever the corresponding positional variable appears in the command file. Positional variables can be concatenated with characters in the command file.

We recommend that you always use the .CMD file type extension in the command file name. This is required if you use a user-defined search path to find the file (see the description of the PATH command earlier in this chapter).

If you will be executing more command files than program files, set the predefined variable $RU_FIRST to FALSE. When $RU_FIRST is set to FALSE, CI assumes that any file name entered without a CI command or file type extension is a command file and immediately attempts to execute the file as a command file.

When you enter an implied or explicit TR command, the following procedure is used to find the command file:

1. If a directory is specified, this directory is searched for the file. If the file is found, it is executed. If the file is not found and a file type extension was not specified, .CMD is assumed and the directory is searched again. If the file still is not found, an error is returned.

2. If no directory information is given, the following occurs:

   a. The TR command checks User-Definable Directory Search Path (UDSP) number 2. If defined, the search path specified by UDSP #2 is used to find the file. If a file type extension is not specified, .CMD is assumed. If the file is not found, an error is returned.

   b. If UDSP #2 is not defined, the following default search sequence is used:

      - The current working directory is searched. If the file is not found, a default file type extension of .CMD is assumed and the working directory is searched again.

      - If you do not have a working directory, all mounted FMGR cartridges are searched.

      - If the file is still not found, global directory CMDFILES is searched, using the .CMD default file type extension. If the file is not found, an error is returned.

For example, if MYCMD is the name of the command file specified in the TR command, a working directory exists, and UDSP #2 is undefined, the default search sequence is as follows:

- Search for MYCMD in the working directory.

- Search for MYCMD.CMD in the working directory.

- Search for MYCMD.CMD in directory /CMDFILES.

If there is no working directory, the search sequence is as follows:

- Search for MYCMD in FMGR cartridges.

- Search for MYCMD.CMD in directory /CMDFILES.

Examples:

In the following example, COMP.CMD, a command file that compiles, links, restores, sizes, and runs program TEST4 and then removes its ID segment, is executed:

```
   CI> tr comp.cmd
```

where COMP.CMD contains the following commands:

```
   ftn7x test4.ftn test4.lst -
   link test4.rel
   rp test4
   sz test4 28
   test4
   of test4 id
```

The following example shows executing a command file that uses positional variables:

```
   CI> comp2 test4 28
```

where file COMP2.CMD contains the following commands:

```
   ftn7x $1.ftn $1.lst -
   link $1.rel
   rp $1
   sz $1 $2
   $1
   of $1 id
```

By specifying a different file name and program size in the TR command, this command file can be used with any FORTRAN program and program size.

UL (Unlock Shareable EMA Partition)*

Purpose: Unlocks a shareable EMA partition so that it can be used by other programs.

Syntax: `UL` *label*

label A name that identifies a shareable EMA partition label, up to five characters.

Description:

This command is used to unlock a shareable EMA partition. The partition is unavailable to other programs when the program that used the partition aborted. The UL command allows you to release the partition. This command is identical to the SYSTEM UL command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

Example:

```
CI> ul carea
```

UNPU (Unpurge Files)

Purpose: Recovers purged files.

Syntax: `UNPU` *mask*

mask A file mask that specifies what files to unpurge. Refer to the “File Masks” section in Chapter 3 for a full description of file masks.

Description:

UNPU restores a purged file to active status. This can be done only if the directory entry of the specified file and the disk space of the file were not reclaimed by the file system. There are no guarantees as to how long these conditions may last. If there are multiple purged files with the same name, it is uncertain which one will be recovered. In this case, the file can be unpurged and renamed, then the next copy of the file with the same name can be unpurged, until all copies have been unpurged. Files recovered with the UNPU command retain the same attributes in effect when they were purged, including their time stamps.

FMGR files and directories cannot be unpurged.

Examples:

The following command unpurges file CHARTER.TXT:

```
CI> unpu charter.txt
Unpurging CHARTER.TXT ...   [ok]
```

The following command attempts to unpurge AREA.FTN but is unsuccessful:

```
CI> unpu area.ftn
Unpurging AREA.FTN ...   [failed]
No such file AREA.FTN
```

UNSET (Delete User-Defined Variable)

Purpose: Deletes a user-defined variable.

Syntax: `UNSET` *variable*

variable A string of up to 16 letters, digits, and underscores; must start with a character. The variable must exist.

Description:

The UNSET command deletes a variable that you defined earlier in the session. Deleting unneeded, user-defined variables frees space that CI can use for defining other variables. You cannot use the UNSET command to delete positional and predefined variables.

Examples:

The following command removes user-defined variable $TEST_NAME:

```
   CI> unset test_name
```

The following command attempts to delete predefined variable $SESSION, which causes an error to occur:

```
   CI> unset session
   Cannot unset SESSION
```

UP (Up a Device)

Purpose: Notifies the system that a specified device is available.

Syntax: `UP` *eqt*

eqt	The EQT number of the device.

Description:

The system downs a device when an error such as a timeout occurs. The EQT remains unavailable until the UP command is given with that EQT number. When a device is UP'd, any pending requests are retried. It is not an error to UP a device that is not down.

This command is identical to the SYSTEM UP command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for more information.

Example:

```
CI> up 8
```

(Make EQT 8 available)

UR (Release Reserved Partition)*

Purpose: Releases a partition previously reserved during system generation or reconfiguration.

Syntax: `UR` *partition*

partition Specifies the number of the partition to be released. The number can be in the range of 1 to 64 depending upon the system configuration.

Description:

This command is identical to the SYSTEM UR command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete description.

VS (Display or Change VMA Size)*

Purpose: Displays the VMA size or changes the VMA size requirements of a restored program.

Syntax: `VS` *prog* `[`*lastpg*`]`

prog The program name, up to five characters.

lastpg The last page of VMA. Range is 31 ≤ *lastpg* ≤ 65535. Note that the actual VMA size will be one page greater than the value entered. For example, if the last page specified is 53, then 54 pages of VMA will be allocated. The default value of *lastpg* is 8191 pages.

Description:

The virtual space is the disk area used for paging data that does not fit in memory. Increasing this space may allow the program to process more data. Decreasing it cuts down on disk space required to run the program. The program must be dormant when this command is given. It must have been linked as a VMA program.

This command is indentical to the SYSTEM VS command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for a complete descriptions.

Examples:

```
CI> vs test4 199                         (Set VMA size to 200 pages)

CI> vs test4                             (Display VMA size of TEST4)
    2000       31   63   32   1   8000
```

In this example,

`20000`	=	logical address
`31`	=	program size in pages
`63`	=	minimum partition size
`32`	=	working set size
`1`	=	program's MSEG size
`8000`	=	virtual memory size

WD (Display or Change Working Directory)

Purpose: Displays or changes the working directory.

Syntax: `WD` [*directory* [*file* | `+S`]]

directory The name of the new working directory. This may be a subdirectory name.

file The command stack file descriptor associated with the new working directory (.STK file type extension recommended). All subsequent posting of command stack contents will be to this file until another file is designated with another WD command. This must be a type 3 or type 4 file.

`+S` This parameter causes posting of the command stack contents either to the default file CI.STK or to a file previously specified. Then the command stack is reinitialized with the contents of CI.STK on the new working directory or cleared if the file does not exist.

Description:

This command sets up the working directory that is used when no directory is specified in a file name. It is searched first by the file system in the file search path. The new working directory can be a subdirectory.

The WD command changes the working directory associated with a session. The working directory cannot be set in a non-session environment such as MTM (Multi-Terminal Monitor) or from the system console when not in session. The working directory does not have to be owned by the user. The user does not have to have read or write access to the named directory, but setting the working directory to a read or write protected directory may cause programs (such as EDIT) problems when they try to create scratch files.

The working directory cannot be defined as a FMGR cartridge, but it can be set to zero. This causes all FMGR cartridges to be searched when a directory is not specified in a file-referencing CI command.

The second parameter provides the option of manipulating the command stack files. It lets you post the contents of the command stack to a file on a particular directory so that the same commands can subsequently be used. If a new command stack file is requested via a WD command, the command stack in memory is posted to the current command stack file if one exists, and the new file is opened and the command stack rewritten with the contents of the new file. If the new file does not exist, the stack is cleared. It will be created at logoff or when another file is requested. Refer to the examples shown below for details.

Examples:

The following commands are entered in sequence with the following assumptions: working directory is DEBBIE with the associated command stack file CI.STK.

```
CI> wd,,+s
```

```
CI> wd /tsmas ts.stk
```

(New working directory is TSMAS. Command stack contents posted to /DEBBIE/CI.STK. Contents of /TSMAS/TS.STK are written into command stack. If TS.STK does not exist, the command stack is cleared and TS.STK will be created at logoff or when the next WD command with the command stack option is executed.)

```
CI> wd /debbie +s
```

(Working directory is changed to DEBBIE. Command stack contents are posted to /TSMAS/TS.STK; if it does not exist, it is created. Contents of CI.STK are written into the command stack.)

WH (System Status Reporting)*

Purpose: Runs the system status program WH to report system information.

Syntax: `WH [`*pram*`]`

pram Specify the information to be displayed:

`AL`	all programs
`PA`	memory partitions
`SM`	all system programs
`PR`|`PL [`*prog*`]`	all or specified ID segments

Description

The information produced from these runstrings is described in the WHZAT utility program description in the *RTE-6/VM Utility Programs Reference Manual,* part number 92084-90007.

WHILE-DO-DONE (Control Structure)

Purpose: Allows repeated execution of a group of commands. WHILE-DO-DONE can be used only in a command file.

Syntax:

```
WHILE command_list1
DO command_list2
DONE
```

command_list A list of commands, either one command per line or multiple commands per line separated by semicolons. A *command_list* can be null.

Description:

The WHILE-DO-DONE control structure allows you to control execution of a command file.

The return status of the last command in the command list for WHILE determines if the command list for DO is executed. If the return status is zero (the command was successful), CI executes the DO branch. If the return status is non-zero (the command was unsuccessful), CI executes the DONE, which terminates the WHILE control structure.

CI determines the end of a command list to be the CI command before the next expected control structure command. For example, the command list for WHILE ends when CI reaches DO.

A WHILE-DO-DONE control structure can be nested in either another WHILE-DO-DONE or in an IF-THEN-ELSE-FI control structure.

DONE is required to end the WHILE-DO-DONE control structure. If you do not include DONE, CI does not recognize the control structure as being finished and continues to process succeeding commands as though the commands were part of the DO command list. Therefore, if a WHILE-DO-DONE control structure was just executed and CI is not executing commands as expected, make sure you entered a DONE command to terminate the control structure.

Example:

The following command file compiles a program and, if the compilation is errorless, links the program. The WHILE loop is repeated eight times, once for each file TEST_FILE1.FTN through TEST_FILE8.FTN. CALC is a user-written program that performs the specified math operation on the two integers and returns the result in variable $RETURN2.

```
set count = 0
WHILE IS $count lt 8
DO
   *
   * Increment counter and
   * retrieve result from $RETURN2
   *
   calc $count + 1
   set count = $return2
   *
   * Compile and link file if no errors
   *
   if ftn7x test_file$count.ftn 0 -
      then link test_file$count.rel
      else echo $return1 ‘errors in test_file ‘$count
   fi
DONE
```

WHOSD (Report User of Directory or Volume)

Purpose: Reports the session that is using a specified directory or a directory on a specified volume as a working directory or as part of a UDSP.

Syntax: `WHOSD [-t] [-m` *idmask*`]` *file*|*directory*|*lu*

`-t`	Trace ID segments back to a file name.
`-m` *idmask*	Check only those ID segments whose names match the mask.
file|*directory*|*lu*	Specifies the file, directory, or volume of which you want to report the user.

Description:

WHOSD displays all users of a specified file, directory, or volume. Directories may not be purged while they are in use. CI volumes may not be dismounted from the system while they are in use.

A file is in use when it is open by any process, or if it is an active type 6 file.

A directory is in use when it is being used as a working directory, when it is included in a user's UDSP, when the type 6 file of an active program resides in the directory, or when any file is open in the directory.

The output of WHOSD can be redirected to an output file by specifying either ">*filename*" or ">>*filename*" in the runstring. The output file specified must be delimited by commas and is position independent. If the file already exists, it will be overwritten. To append to a file, ">>*filename*" can be used. If the file does not already exist, it will be created.

If the output file is not specified, WHOSD breaks the output into screen pages. Paging is disabled when an output file is specified. For example, to output to the terminal without paging, specify ">1" in the command line. Multiple redirection strings may occur in the runstring; however, only the last redirection is executed.

Return Values:

WHOSD returns, in $RETURN1, the number of users found for the specified file, directory, or volume. If an error is encountered, WHOSD returns a negative number.

Examples:

The following example shows all of the current users of LU 32.

```
CI.73> whosd 32
Directory - /PROGRAMS/ is being used by DONP.GENERAL(73) in a UDSP
Directory - /CMDFILES/ is being used by DONP.GENERAL(73) in a UDSP
Program   - DL is RPed from LU 32
Program   - IS is RPed from LU 32
Program   - HELP is RPed from LU 32
Program   - EDIT is RPed from LU 32
Program   - CMD is RPed from LU 32
Program   - CALLS is RPed from LU 32
Program   - LINK is RPed from LU 32
Program   - FTN7X is RPed from LU 32
Program   - MACRO is RPed from LU 32
Program   - DEBUG is RPed from LU 32
Program   - LI is RPed from LU 32
Program   - ALARM is RPed from LU 32
Program   - TIMER is RPed from LU 32
File      - /SYSTEM/ALARM.DATA open to ALARM
```

In the following example, the /PROGRAMS directory is scanned for all uses. Any active program that resides on the same LU as /PROGRAMS.DIR is traced back to its type 6 file to determine if it resides in /PROGRAMS.DIR.

```
CI.73> whosd -t /programs
Directory - /PROGRAMS/ is being used by DONP.GENERAL(73) in a UDSP
Program   - DL is RPed from /PROGRAMS/DL.RUN
Program   - IS is RPed from /PROGRAMS/IS.RUN
Program   - HELP is RPed from /PROGRAMS/HELP.RUN
Program   - EDIT is RPed from /PROGRAMS/EDIT.RUN
Program   - CMD is RPed from /PROGRAMS/CMD.RUN
Program   - CALLS is RPed from /PROGRAMS/CALLS.RUN
Program   - LINK is RPed from /PROGRAMS/LINK.RUN
Program   - FTN7X is RPed from /PROGRAMS/FTN7X.RUN
Program   - MACRO is RPed from /PROGRAMS/MACRO.RUN
Program   - DEBUG is RPed from /PROGRAMS/DEBUG.RUN
Program   - LI is RPed from /PROGRAMS/LI.RUN
Program   - ALARM is RPed from /PROGRAMS/ALARM.RUN
Program   - TIMER is RPed from /PROGRAMS/TIMER.RUN
```

Scan /PROGRAMS for working directories, UDSPs, and also check only those ID segments starting with the letter C:

```
CI.73> whosd -m c@ /programs
Directory - /PROGRAMS/ is being used by DONP.GENERAL(73) in a UDSP
Program   - CMD is RPed from /PROGRAMS
Program   - CALLS is RPed from /PROGRAMS
```

The following example enables the trace option and logs all of the users of LUs 19 and 20 to the file USERS.LST:

```
CI> whosd -t >users.lst 19
CI> whosd -t 20  >>users.lst
```

WS (Display or Modify VMA Working Set Size)*

Purpose: Displays the VMA working set size or modifies the working set size requirements of a restored program.

Syntax: `WS` *prog* [*wrksz*]

prog The program name, up to five characters, with an optional session identifier.

wrksz The working set size in pages (not including PTE). Range is 2 ≤ *wrksz* ≤ 1022. Default is 31 pages.

Description:

The working set is a number of pages in a VMA user's partition that is used to hold a portion of the virtual memory space, including the page currently being accessed. Increasing the working set generally improves performance at a cost of more memory for running the program. The program must be dormant when this command is used. This command only works for VMA programs. For EMA programs, use the SZ command.

This command is identical to the SYSTEM WS command. Refer to the *RTE-6/VM Terminal User's Reference Manual* for more information.

XQ (Run Program without Wait)

Purpose: Immediately schedules a program for execution.

Syntax: `XQ` *prog|file* `[`*pram*5*`]`

prog|file The program name, up to five characters, or a file descriptor that identifies a type 6 program file to be executed.

*pram*5* The parameters to be passed to the program. The maximum runstring length, including the implied RU and delimiter, is 256 characters. This can be five numeric parameters or a character string.

Description:

The XQ command performs a “schedule without wait” operation. All other actions (comments and error handling) are identical to that of the RU command.

6

FMP Routines

The File Management Package (FMP) for the CI file system is a set of routines that manage disk files. 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.

The FMP calls used in the CI file system are analogous to the File Manager (FMGR) FMP calls described in the *RTE-6/VM Programmer's Reference Manual*. Appendix B of this manual is a guide to converting FMGR FMP calls to FMP calls for use in the CI file system environment.

All FMP calls in this chapter and in Appendix B refer to those used in the CI file system environment. These are referred to as CI FMP calls. The FMP calls described in the *RTE-6/VM Programmer's Reference Manual* are referred to as FMGR FMP calls. Note that most of the FMP calls described in this chapter can be used to access files in the FMGR file system, observing the restrictions imposed by the differences in naming conventions and file system properties as described in the Preface.

The most common usage of FMP calls is to create or purge files and to read or write data at various locations in the files. The FMP calls provided for these basic tasks are listed in Table 6-1, File Manipulation FMP Routines, and Table 6-2, Directory Access FMP Routines. These calls can be used in the FMGR file system as long as the FMGR file system conventions are followed. There are other special-purpose FMP calls listed in Tables 6-3 through 6-6 under the following categories: Masking FMP Routines, Device FMP Routines, Parsing FMP Routines, and Utility FMP Routines.

The first category of special purpose calls consists of subroutines used primarily in the CI file system environment. These calls are used to create or manipulate the hierarchical directories as well as handling time stamps, file masks, and other CI file properties.

The utility calls are subroutines that perform a variety of functions, for example, copy data, error handling, device control, and mount or dismount disk volumes or disk cartridges (FMGR). Most of these calls can be used in both FMGR and CI file systems.

The FMP calls used in the optional DS environment are described in the "Special Purpose DS Communication Subroutines" section of this chapter.

General Considerations

Most FMP calls access files or file directories. Files contain programs or data; file directories identify and describe files. Refer to Chapter 3, “Manipulating Files,” for a detailed description of files, directories, and the file descriptor parameters.

The calls described in the following pages let your program 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 as 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 programs.

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.

Refer to the "Manipulating Files" section in Chapter 3 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 filename 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, defaulting 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 filedescriptor 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 Chapter 3.

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 Chapter 3 of this 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:

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. Any special processing based on file type must be supplied by the application program.

size (Optional) The number of 128-word blocks in the file.

reclen (Optional) For type 2 files, specifies the length of the records in the file.

user (Optional) Used only with the optional DS network. The user account name under which the file exists; delimited by square brackets. The full form is:

user_name .group / password

node (Optional) Used only with the optional DS network. The DS node where the file resides; preceded by a right angle bracket (>).

The first format is the same as an 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 filename. 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 place holders 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 (filedescriptor, 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 will 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 *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 6-1.

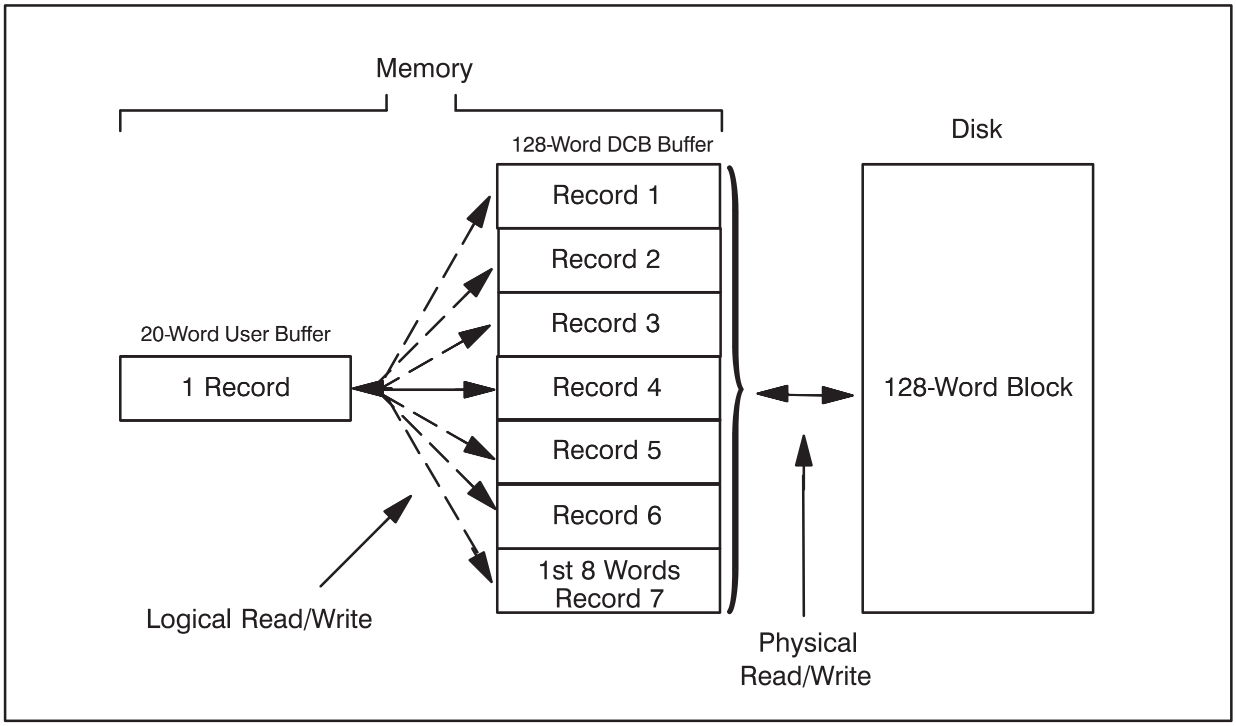


Figure 6-1. Logical Transfer Between Disk File and Buffers

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 6-1), 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 6-2. 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.

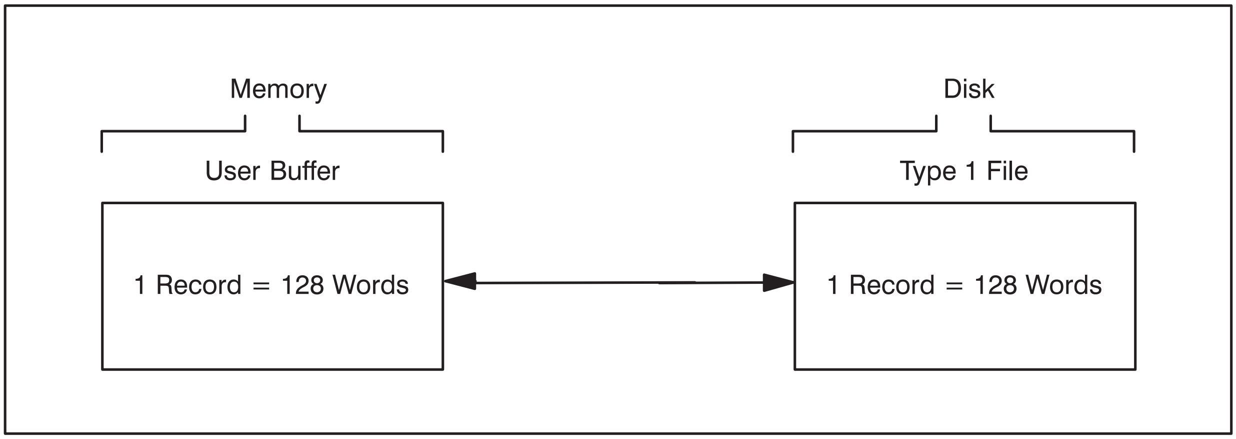


Figure 6-2. Data Transfers with Type 1 Files

Descriptions of FMP Routines

This section contains descriptions of all FMP routines; the routines are listed alphabetically. Tables 6-1 through 6-6 present functional groupings of the routines.

Table 6-1. 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

Table 6-2. 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
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 6-3. 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 6-4. 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 6-5. 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 6-6. 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 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)

character*(*) filedescriptor
integer*2 error
integer*4 time
```

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 January 1, 1970.

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 zero 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 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 (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 filename. *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, where PROG is the original program name and the session number is 78, is as follows:

```
   PROG, PRO78, PR78A, PR78B, . . . , PR78Z
```

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
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 T option lets you 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 lets you 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)

character*(*) filedescriptor
integer*4 time
```

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.

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)

integer*2  error
character*(*)  filedescriptor
integer*4  eofpos
```

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.

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 "`CAN'T 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 will be 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 it can be used 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 any 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 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 let FmpNextMask perform the search in the correct order. For a complete description of the mask qualifier, refer to the file mask syntax description in Chapter 3.

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 discussion in the *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005.

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 22 lines. 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 22 lines
<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 22 lines.

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.

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 FmpOpen cannot open a FMGR type 0 file unless it is specified as an LU.

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 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
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 an 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.

FmpOpen cannot open a FMGR type 0 file unless it is specified as an LU. If the file descriptor specifies 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 opened 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 file descriptor 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 filedescriptor 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 with a length of 24 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 lets the filedescriptor parameter 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.

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 let other programs 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 cleanup 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 lets them 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 through 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 Chapter 5 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 boot-up to be used as a scratch cartridge. The BOOTEX SC command or the FMGR VL command 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 program’s 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 and the associated group 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. The associated group of the directory or volume is given by the group account portion of the logon name.

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 any 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	22 lines per page
1	22 lines (1 page) to be listed before breaking
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 in Chapter 3 for a description of the DS node name and user account name.

FmpParseName should be used to upgrade programs designed to manipulate FMGR files 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 in Chapter 3 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 DL command description in Chapter 5 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*.

The *qual* parameter permits FmpParsePath to correctly parse file descriptors that contain masks. Mask qualifiers are described in the DL command description in Chapter 5.

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 description in the *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005, 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

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.

The associated group of a directory or volume is either the associated group of the user who created it or the associated group assigned via the FmpSetOwner routine. The associated group of all files within a directory is the same as that of the directory.

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. Maxlength 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 the 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 zero (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.

FmpReadString

FmpReadString is an integer function that allows reading characters 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*`)`

```
character*(*) filedescriptor
integer*4 nrecords
```

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.

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*`)`

```
character*(*) filedescriptor
integer*2 len
```

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.

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 of the variable length records in the file.

Note The length returned for type 3 or higher 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 do 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", "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:

- there is a system program with the assigned or specified name.
- there is already a program with the assigned or specified name, but it is not RP'd.
- there is no program with that name currently RP'd.
- the program was RP'd without the temporary (ID) bit clear; therefore, it is permanent.

P (permanent) Do not release the ID segment when the RP'd program completes.

If the program already exists and cloning cannot occur, error −239 is returned.

If FmpRpProgram needs to clone, it will replace the fourth and fifth characters of the program name with the session number. If that name is also taken, it will replace the third and fourth characters with the session number and the fifth character with A. If that name is taken, it will use B as the fifth character, and so on.

If the RPL checksum on the type 6 file does not match the system, the file's checksum is changed and the program is RP'd, but FmpRpProgram returns error −240 (RPL checksum changed). This error is a warning and can be displayed to you or ignored.

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 unless the :IH option is used. 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.

FmpRpProgram is used by FmpRunProgram, CI, and most other program scheduling requests to search for an existing program with the specified or assigned name. FmpRpProgram searches for the program to be RP'd as follows:

1. If a directory is specified, this directory is searched for the file. If the file is found, it is RP'd. If the file is not found and a file type extension was not specified, .RUN is assumed and the directory is searched again. If the file still is not found, an error is returned.

2. If no directory information is given, the following occurs:

    a. If a program with the specified or assigned name is already RP'd and is dormant, this program is used. If the program is busy and cannot be cloned, an error is returned.

    b. If the program has not been RP'd or is busy but can be cloned, a search is made for the program (type 6) file. If User-Definable Directory Search Path (UDSP) number one is defined, a default file type extension of .RUN is assumed and the search path defined by UDSP #1 is used to find the file. If the file is not found, an error is returned.

    c. If UDSP #1 is not defined, the following default search sequence is used:

        - The current working directory is searched. If the file is not found, a default file type extension of .RUN is assumed and the working directory is searched again.

        - If you do not have a working directory, all FMGR cartridges are searched.

        - If the file is still not found, global directory PROGRAMS is searched, using the .RUN default file type extension. If the file is not found, an error is returned.

UDSP #1 can be defined using the CI PATH command. Refer to the PATH command description in Chapter 5 for more information.

If a working directory exists, programs on a FMGR cartridge cannot be run unless the directory is specified by PROG::0 or PROG::crn.

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, the prams are 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, an attempt will be made to create a clone name by replacing the fourth and fifth characters of the program name with the session number; for example, EDI77. If that name is already taken, it will replace the third and fourth characters with the session number and the fifth character with A; for example, ED77A. If that name is taken, it will use B as the fifth character, and so on. It is usually not necessary to clone a program, because programs are identified by their name plus their session number. If :IH follows the program name (for example, RU,PROG:IH), cloning is inhibited.

The order of search for the program is the same as for FmpRpProgram.

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 description in the *RTE-6/VM Programmer's Reference Manual*, part number 92084-90005. 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 the 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 will always be 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 is a character string specifying the name of the file or the CI volume number.

owneraccess is a character string specifying the access rights of the owner of the file/directory/volume.

othersaccess is a character string specifying 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, then 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 and 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 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)

character*(*) filedescriptor
integer*2 error
integer*4 size
```

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.

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
integer*2 error
```

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 will never be 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 unique name 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 an FMGR directory, the prefix should be chosen to minimize the chance of a duplicate file name when the unique name 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
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 specifying the name of the file to be unpurged.

FmpUnPurge verifies the directory entry for the file and any extents, and makes sure 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 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)

character*(*) filedescriptor
integer*2 error
integer*4 time
```

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.

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 0 (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. *string* 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 dirdcb(*), 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 0-7 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. *wild* 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 file descriptor 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 some 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 amount of data to be read. A maximum of 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 demonstrates the use of the simplest FMP routines (open, close, read, write). The second shows how file masking, a somewhat more advanced FMP function, is used. The third combines many of the FMP routines in a 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 big DCB’s 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 which 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 close down 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 database.

```
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 what record number to change
```

```
   5  write(1,*) ’Record to add? _’
      read(1,*,end=10,err=10) currec

  c Position to this record (let FMP trap bad record c number)

      call FmpSetPosition(dcb,err,currec,–1J)
      if (err .eq. –12) then
        write(1,*) ’That record doesn’t 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
  c 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
```

7

Exception Condition Handling

Cleaning Up Open Files

If a program opens a file and terminates (or is aborted) without closing it, the file is left open. The file system (specifically D.RTR) attempts to clean up open files automatically. How this clean up 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 a limited amount of time only 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 whenever the creating program is no longer using the file, whether the program closed the file or left it open at termination. 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 temporary. 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 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 that 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 paragraphs describe how clean up is done for files that are left open. The four possible cases are:

- Normal CI files
- Temporary CI files
- Normal FMGR files
- 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). Contained in the open flag is a pointer to the file, a pointer to the ID segment of the program that opened the file, and status bits that 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 its 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, 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 each time a 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 D.RTR finds an invalid open flag and determines that it came from the FmpOpen that created the file and that the T option was used in that FmpOpen call, the file is purged.

Note the restriction that the file is purged only if the invalid open flag 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 disk. 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 disk. 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, and locking, mounting, and dismounting cartridges. When D.RTR finds an open flag, it first checks to see if the program in the associated ID segment is dormant. If so, 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 only scans until it finds the file to open; if the invalid flag is further down the directory, D.RTR will not find the clear bit. 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. The CI DL command does not do this because it does not do the cartridge lock/unlock sequence.

Because the sequence counter has only 32 potential values, it is possible, though highly unlikely, that programs will have run in an ID segment and terminated 32 times before the open flag is checked. This would cause the sequence counter to roll back 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 as follows:

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 the location of the temporary file entry. 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 ensure 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 encounters 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.

A

Error Messages and Codes

Most of the error messages caused by an operator action are simple and self-explanatory. However, some are displayed in the form of an error code or in a particular format where a number of variables may be displayed. The common error formats are described below. Operator error messages are listed in alphabetical order, along with the explanation and suggestions for corrective action, in the “Error Messages” section.

Error Formats

Error messages have different formats, depending upon the operation being performed. Errors reported by CI in response to commands such as AS, RU, and SZ are in the form of brief descriptive messages. For example:

```
Illegal variable name <name>
Usage: RP file [progname]
```

There may be occasions when error messages are reported by the system. For example:

```
No SAM available at this time to perform the request
The specified LU is not assigned on this system
```

Some errors are reported in the form of an error code. These are reported in the form:

```
FMP error -59
```

The error codes are listed and described under the heading FMP Error Codes in this appendix.

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
```

In this message, the error code −250 indicates that D.ERR was not available and xxx is the FMP error that occurred.

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 the specified name; repeat with a new name or purge the existing file.

−003 Backspace illegal

An attempt was made to backspace a device (or type 0 file) that cannot be backspaced; check the device type.

−004 Record size illegal

You made an attempt to create a type 2 file with a zero record length.

−005 Bad record length

You tried to read or position to a record not written, or you tried, on update, to write an illegal record length; check the position or size parameters.

−006 No such file

You attempted to access a file that cannot be found. Check the file name or cartridge number.

−007 Incorrect security code

You cannot access a file without the correct security code. Use the correct code or do not try to access the file.

−008 File is already open

You attempted to open a file already open exclusively or open to eight programs, or the cartridge containing the file is locked; use CL or DL to locate the lock.

−009 Must not be a device

Type 0 files cannot be positioned or be forced to type 1; check the file type.

−010 Not enough parameters

Required parameters were omitted from call; enter the parameters.

−011 DCB is not open

You made an attempt to access an unopened DCB. Check the error code on open attempt.

−012 Illegal file position

You made an attempt to read or write or position beyond the file boundaries; check the record position parameters, as the result depends on the file type and call.

−013 Disk is locked

The cartridge is locked; initialize the cartridge if not initialized, otherwise, try again.

−014 Directory is full

There is no more room in the file directory; purge the files and pack the directory if possible, or try another cartridge.

−015 Illegal name

The file name does not conform to syntax rules; correct the name.

−016 Size = 0 or illegal type 0 file access

The wrong type code was supplied, or you attempted to create or purge type 0 file or create 0-length file; check the size and type parameters.

−017 Device I/O failed

You attempted to read/write or position type 0 file that does not support the operation; check the file parameters, namr.

−018 Illegal LU

Do not attempt to access an undefined LU.

−019 Illegal LU 2 or 3 access

The System Manager is the only user with the capability to write on a system disk.

−020 Illegal access LU

1. The logical unit number specified in the LU 2 or CS command was not a positive logical unit number. Re-enter the correct command.

2. There is an LU entry in the cartridge list that does not point to a disk device. After the disk was mounted, the LU command switched the device. Switch the LU back to its disk definition. If desired, dismount the disk. The LU can then be switched to a non-disk device.

−021 Illegal destination LU

The specified LU was not allocated by GASP. Try again using an LU allocated by GASP.

−022 No available spool LUs

All spool logical units are currently being used. Re-run the job after a spool LU becomes available.

−023 No available spool files

All spool files are currently being used. Re-run the job after a spool file becomes available.

−024 No more batch switches

The LU switch table is full; the size of the switch table specified at system generation is inadequate. Notify the System Manager.

−025 No SPLCON room

The SPLCON control-record area is full. This error may occur when the spool system is competing with programs using their own spooling and running outside of batch. Re-run the job when SPLCON control-record entry space is available.

−026 Queue full or too many pending spools

The spool queue is full or the maximum number of spools pending has been exceeded. Re-run the job when the space becomes available.

−030 Value too large for parameter

The value is greater than the legal maximum.

−032 No such cartridge

The specified cartridge is not mounted. Check the disk specification in call.

−033 Ran out of disk space

The disk specified for a disk file has insufficient room for file creation. This could occur during a WRITF if an extent is being created.

−034 Disk is already mounted

The 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 the exclusive use of a device that 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 OF'd 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.

−039 Spool LU not mapped to spool driver

A spool LU must point to a spool EQT. Switch all spool LUs to point to spool EQTs and try the spool file setup again.

−040 LU not found in SST

You are trying to access an LU that is not in your Session Switch Table (SST). Use the SL command to add the LU to your SST.

−041 No room in SST

There are no spare entries left in the Session Switch Table. Spare entries can be recovered by using the :SL,lu,- command, where lu is a session logical unit number that is not needed.

−046 More than 255 extents

An attempt to create more than 255 extents was made. Use a file with a larger initial size.

−047 No session LU available for spool file

If the session LU to be used for the spool file is not specified during setup, SMP allocates a session LU less than 64 that is not already used in the Session Switch Table. Use the :SL,lu,- command to release a session LU in the spare part of your Session Switch Table.

−048 Spool not initialized

Spooling has not been initialized (run GASP to do so), the SMP program cannot be found, or there are no partitions large enough to run it. The default for SMP is real-time (type 2) file sized to 6 pages.

−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.

−052 Spool shut down. Spool file setup failed

Spool shut down is in progress. A write (WR) or read/write (BO) spool file cannot be set up at this time. Start up spooling using the GASP SU command, and try to set up the spool file again.

−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 that is “bad” due to a parity error in the partition or a reserved partition that 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 that 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

The shareable EMA table already contains 15 entries. If possible, OF,,ID any programs not in use that access shareable EMA. NOTE: All programs that access a certain shareable EMA area must be OF’d for the shareable EMA table entry to be deleted.

−056 SHEMA assigned to non-existent partition

The shareable EMA area used by the program is assigned to a reserved partition that 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 that 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 63 programs using shareable EMA area

There are already 63 programs RP'd that access the shareable EMA area specified by the program. No more programs may be RP'd (or 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

This indicates a possible operator error; recheck your previous entries for illegal or misplaced parameters.

−102 D.RTR not available

D.RTR is not RP'd or has been OF'd; the 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 that 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 that 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 is 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 when mounting or creating a directory. 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

The system 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 run strings. If a shorter string does not work (it probably will not), rebooting may help as SAM is fragmented, or you may need to regenerate the system to get more SAM.

−229 Active working directory

You tried to purge a working directory or dismount a disk containing a working directory.

−230 Illegal use of directory

A directory was used illegally (for example, 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

You requested unavailable information (for example, time stamp) about an old file.

−233 No such user

The user name was not found by FmpSetOwner.

−234 Size mismatch on copy

The 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

A 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

You cannot RP a 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.

−242 Disk I/O failed

D.RTR got an EXEC error 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.

−246 System common changed

You tried to RP a program that defines a system common differently than it is defined on the current system.

−247 UDSP not defined

The UDSP is not defined for one the following reasons:

- None of the entries in the UDSP has 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.

−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.

−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.

−256 No such session

From FmpRpProgram.

−257 No such program

From FmpRpProgram.

−270 Update time already current

From FmpCopy.

DS Transparency Software Errors

The following error codes reflect errors in DS transparency software.

−300 Illegal remote access

This 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

You 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.

−308 Connection broken

The remote system monitor TRFAS was restarted since the connection was open.

DS/1000 Software Errors

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]

The other system is probably down or not running DS.

−313 No TRFAS at remote system [DS06]

The 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.

Native Language Support Utilities Errors

FMP errors −401 through −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 FMP calls used in the FMGR file system environment to CI calls for use in the CI file system environment. The conversion procedures are written to assist you in converting programs with which you may be unfamiliar.

General Considerations

File system calls usually make up a small percentage of a program, so the conversion effort is minimal because in most cases the program logic should not have to change. Many of the FMGR calls will still work, although it is recommended that you convert programs to allow full usage of the enhancements available with the FMP calls.

All required parameters in FMP calls must be supplied.

When using an FMP call as a function, the FMP call must be typed according to the type declaration of the return value.

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. A sample file descriptor is shown below:

```
   /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. CI 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   /USR/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.

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 need to 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 distinquish subfield types 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 name in a runstring as separate character variables. (Fparm is not available to Pascal users.) Descriptions of these routines can be found in the *RTE-A • RTE-6/VM Relocatable Libraries Reference Manual*, part number 92077-90037.

Examples:

The following 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 type and size values can be changed. In the example, the code sequence constructs the name of the 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, then 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)
```

OPEN and OPENF Calls

ALL OPEN and OPENF calls are replaced by FmpOpen calls. Handling of file name parsing and character string is the same as previously described. In addition, 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 CI 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 allowed to open an FMGR file, but not to create a new one. (This is further discussed under the section describing the CREAT call.) Other options available and their octal equivalents in the option word of the 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 a CI call. In converting a FMGR call, start with rwo, then add the other options to match the FMGR call options. For example, an option word 45B (open, permitting type 1 and type 2 extents, forced to type 1 and shared) would be option word “rwoxfs”. The option characters can be 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 only for reading files; do not use it for writing without providing synchronization.

Note that the buffer size is specified in blocks, rather than in words. The buffer size needed is the OPEN buffer size divided by 128:

```
   type  =  fmpopen(dcb,err,name,options,256.128)
```

The buffer size parameter must be supplied; if the FMGR call did not supply a buffer size, use a value of 1.

FmpOpen call accepts a logical unit number as in the OPENF call, 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 open is shared.

Note that CI files can be opened to a large number of programs (more than 7), but there must be room in D.RTR's internal table for the open flag. If there is not enough room, the open will fail. One program can have the same file open several times (if file is shared); this is different from how it used to be when a file is only open to a program once.

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 are similar to the FMGR calls, 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, the following FMGR call sequence

```
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. (Refer to the FmpSetPosition description for details.)

For example, the following code

```
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
```

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 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 CLOSE requires 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  lofc(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 CLOSE call specified the number of blocks to truncate, while the converted code specifies the desired file size. The new sequence will truncate extra extents, which was not possible before. All sizes are double integers. There is no call provided for this sequence, since 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 they can create the file. These calls are similar to the OPEN and OPENF call. Refer to the description of OPEN and OPENF for conversion details. Additional size, type, and record length information is passed as ASCII, appended to the name; FmpBuildName is useful for creating ASCII strings.

Any options used in an OPEN call can be specified when creating a file. CREAT sets up default options of nonshared update mode, so to create the equivalent code sequence, use the string rwcu.

For example,

```
call  creat(dcb,err,pbuf,pbuf(8),pbuf(7),pbuf(5),pbuf(6))
```

is replaced by:

```
call  fmpopen(dcb,err,name,’rwcu’,1)
```

This will give an error −2 if the file exists. Specifying both o and c will open the existing file or create a new one if necessary. Note that this sequence 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
```

Handling of scratch files consists of creating a name that is unique and a bit that indicates that this file is not important. The program that creates the scratch file must purge it before exiting. 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 extendable type 1 scratch file with a starting size of 24 blocks, the FMGR calling sequence:

```
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
```

is replaced by:

```
call  fmpuniquename(’TEMP’,name)
call  fmpopen(dcb,err,name//’:::1:24’,
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 differentiate the name from a number. If there is a chance the scratch file will go on an FMGR cartridge, the prefix should be short (one character) to avoid having duplicate six-character names. In any case, the name must be known in order to purge the file.

APOSN, LOCF and POSNT Calls

File positioning is also discussed in the section on random access READF and WRITF calls. APOSN and LOCF use internal file pointers, while POSNT positions by record number. These functions are performed with FmpPosition and FmpSetPosition for CI 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 LOCF call

```
call  locf(dcb,err,record,block,offset)
if (err .1t. 0) goto 900
```

is replaced with

```
integer*4  drecord, dposition
call  fmpposition(dcb,err,drecord,dposition)
if (err .1t. 0) goto 900
```

The new internal position value is related to the previous value: position = block * 128 + offset.

Use caution when changing LOCF calls. They contain much information, and it is not always easy to tell what is used and what is not. FmpPosition only returns file position. Other LOCF information can be obtained using the FmpSize and FmpEof calls. The FmpSize call returns the total size of the file in blocks, not the size of the main part of the file in sectors. The FmpEof call tells how much of the file is being used. There is no CI call to return the logical unit of a file, because the logical unit cannot be used in place of the directory name. The FmpRecordLength call returns file record length; FmpOpen returns file type when it opens the file.

To restore file position to a place recorded with APOSN, use FmpSetPosition. For example:

```
call  aposn(dcb,err,record,block,offset)
if (err .1t. 0) goto 900
```

is replaced by:

```
integer*4  drecord, dposition
call  fmpsetposition(dcb,err,drecord,dposition)
if (err .1t. 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 later use by calls that position by record number.

FmpSetPosition is also used to position files by record number. Positioning type 1 and 2 files was previously discussed under READF and WRITF. Positioning type 0 and type 3 and above files was described in the FmpPosition description in Chapter 6. POSNT can 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, using POSNT:

```
call  posnt(dcb,err,100,1)
if (err .1t. 0) goto 90O
          .
          .
          .
call  posnt(dcb,err,-10,0)
if (err .1t. 0) goto 900
```

The above sequence can be replaced by:

```
integer*4 drecord, dposition
call  fmpsetposition(dcb,err,100J,-lJ)
if (err .1t. 0) goto 900
          .
          .
          .
call  fmpposition(dcb,err,drecord,dposition)
if (err .1t. 0) goto 900
call  fmpsetposition(dcb,err,drecord-10,-1J)
if (err .1t. 0) goto 900
```

The −1J parameter passed as the file position indicates that 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 CI 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.

The following PURGE call,

```
call  purge(dcb,err,pbuf,pbuf(5),pbuf(6))
if (err .1t. 0) goto 900
```

is replaced by:

```
call  fmpclose(dcb,err)
err = fmppurge(name)
if (err .1t. 0) goto 900
```

The following NAMF call,

```
call  namf(dcb,err,pbuf,newname,pbuf(5),pbuf(6))
if (err .1t. 0) goto 900
```

is replaced by:

```
call  fmpclose(dcb,err)
err =  fmprename(oldname,errl,newname,err2)
if (err .1t. 0) goto 900
```

Extended Calls

Extended calls (ones that start with E, that is, EREAD, EWRIT, ECREA) are replaced in the same way as their non-extended equivalents. These 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 which 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

CI calls that perform the functions done by Rwndf, Post and Fcont are FmpRewind, FmpPost and FmpControl, respectively. Their use is described in Chapter 6.

Accessing FMGR Files

This section describes what happens when a CI call refers to a FMGR file, which provides the same level of service as is obtained with FMGR calls referring to FMGR files. The caller can open, create, or purge files on FMGR disk cartridges. 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 paragraphs discuss the cases where the cartridge is not specified.

If there is a CI directory with the same name as an FMGR cartridge CRN, then that cartridge cannot be accessed via the CI calls, although it can be with FMGR calls. (CI calls first check CI directories, while FMGR calls first check disk cartridges.) In general, it is confusing to have a CI directory with the same name as a disk cartridge, so it is not recommended (although it is allowed).

If the directory is not specified, for example, 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::O, or FOO::O:3, then all of the FMGR disk cartridges mounted to this user will be searched. This also applies to the case where there is no working directory and a directory is not specified. This is one way to get a multiple disk search with the CI calls, although it only searches FMGR disk cartridges.

Calls that specify a file name only work with FMGR files if the information is available in the FMGR directory. Thus, a user can get the name of an FMGR file, but cannot get the timestamps or position of end-of-file. In the latter cases, the FMGR cartridges are not even searched, even if a disk cartridge name is specified. A summary is given below.

- Calls that pass file names and work with FMGR files:

  FmpAccess, FmpOpen, FmpPurge, FmpRename, FmpSize,

- Calls that pass file names but do not work with FMGR files:

  FmpAccessTime, FmpCreateDir, FmpCreateTime, FmpEof,
  FmpRecordCount, FmpRecordLen, FmpSetAccess, FmpSetOwner,
  FmpSetWorkingDir, FmpUnpurge, FmpUpdateTime

- Calls that do not pass file names and 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 a disk cartridge, then the FMGR rules for parsing namrs apply. Dots and slashes in names are not significant on FMGR directories. The name is truncated to six characters.

Accesses to FMGR directories follow all rules for the FMGR file system, such as that for open flags and extent creation. The same protection checks (security code, and so on) are made, although it is not guaranteed that all invalid requests will be caught (for example, illegal characters in file names). FMGR file system error codes are returned when appropriate.

Calls that specify a DCB work regardless of whether the file is FMGR or CI, including read, write, position, and so on. This includes files with extents and files with odd byte length records.

Standard Type Extensions

File type extensions are used to replace the special characters used in FMGR to designate a group of files, for example, % for relocatables and & for source. Table B-1 lists the standard file type extensions used in the CI file system.

Table B-1. Standard File Type Extensions

File Type Extension	Description
.cmd	CI command file
.dat	Data file
.dbg	Symbolic Debug/1000 file
.dir	Directory or subdirectory entry
.doc	Document file
.err	Error message file
.ftn	FORTRAN source file
.ftni	FORTRAN source include file
.hlp	Help file
.lib	Library of relocatables
.lod	LINK command file
.lst	Listing
.mac	Macro source file
.maci	Macro source include file
.map	Load map list
.merg	Merge file for relocatables without headers
.mlb	Macro library file
.mnf	Manual numbering file
.mrg	Merge file for relocatable libraries with headers
.pas	Pascal source file
.pasi	Pascal source include file
.rel	Relocatable (binary) file
.run	Program file
.snp	System snapshot file
.stk	Command stack file
.sys	System file
.txt	Text file

Index