

# HP 3000 Series II Computer System



## MPE Intrinsic Reference Manual



5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA 95050

**HP Computer Museum**  
**[www.hpmuseum.net](http://www.hpmuseum.net)**

**For research and education purposes only.**

## NOTICE

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

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

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

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

# LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the most recent date on which the technical material on any given page was altered. If a page is simply re-arranged due to a technical change on a previous page, it is not listed as a changed page. Within the manual, changes are marked with a vertical bar in the margin.

Page	Effective Date	Page	Effective Date
Title . . . . .	Feb 1977	2-119 to 2-138 . . . . .	Jun 1976
ii . . . . .	Jun 1976	2-139 . . . . .	Oct 1976
iii to iv . . . . .	Feb 1977	2-140 to 2-154 . . . . .	Jun 1976
v to vi . . . . .	Jun 1976	3-1 to 3-58 . . . . .	Jun 1976
viii . . . . .	Jan 1977	3-59 . . . . .	Jan 1977
ix . . . . .	Jun 1976	3-60 to 3-78 . . . . .	Jun 1976
x to xi . . . . .	Jan 1977	4-1 to 4-45 . . . . .	Jun 1976
1-1 to 1-12 . . . . .	Jun 1976	5-1 to 5-2 . . . . .	Jun 1976
1-13 . . . . .	Jan 1977	5-3 to 5-4 . . . . .	Oct 1976
2-1 to 2-2 . . . . .	Jun 1976	5-5 to 5-20 . . . . .	Jun 1976
2-3 . . . . .	Jan 1977	5-21 to 5-23 . . . . .	Oct 1976
2-36A to 2-36C . . . . .	Oct 1976	5-24 to 5-26 . . . . .	Jan 1977
2-36D . . . . .	Jan 1977	5-27 to 5-32 . . . . .	Oct 1976
2-37 to 2-44 . . . . .	Jun 1976	6-1 to 6-9 . . . . .	Jun 1976
2-45 to 2-46 . . . . .	Jan 1977	7-1 to 7-16 . . . . .	Jun 1976
2-47 . . . . .	Oct 1976	8-1 . . . . .	Oct 1976
2-48 to 2-70 . . . . .	Jun 1976	8-2 to 8-16 . . . . .	Jun 1976
2-71 . . . . .	Jan 1977	9-1 . . . . .	Jan 1977
2-72 to 2-87 . . . . .	Jun 1976	9-2 to 9-9 . . . . .	Jun 1976
2-88 . . . . .	Oct 1976	10-1 to 10-6 . . . . .	Jun 1976
2-89 . . . . .	Jun 1976	10-7 to 10-10A . . . . .	Jan 1977
2-90 . . . . .	Jan 1977	A-1 . . . . .	Jun 1976
2-91 to 2-99 . . . . .	Jun 1976	B-1 to B-2 . . . . .	Jan 1977
2-100 . . . . .	Jan 1977	I-1 . . . . .	Jun 1976
2-101 to 2-106 . . . . .	Jun 1976	I-2 . . . . .	Jan 1977
2-106A to 2-107 . . . . .	Jan 1977	I-3 . . . . .	Jun 1976
2-108 to 2-116 . . . . .	Jun 1976	I-4 to I-5 . . . . .	Jan 1977
2-117 . . . . .	Jan 1977	I-6 to I-7 . . . . .	Jun 1976
2-118 . . . . .	Oct 1976	I-8 . . . . .	Jan 1977

# PRINTING HISTORY

New editions incorporate all update material since the previous edition. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover changes only when a new edition is published. If minor corrections and updates are incorporated, the manual is reprinted but neither the date on the title page and back cover nor the edition change.

First Edition . . . . . Jun 1976  
Update Package #1 . . . . . Oct 1976  
Update Package #2 . . . . . Jan 1977  
Second Edition . . . . . Feb 1977  
Updates through #2 . . . . . Incorporated Feb 1977

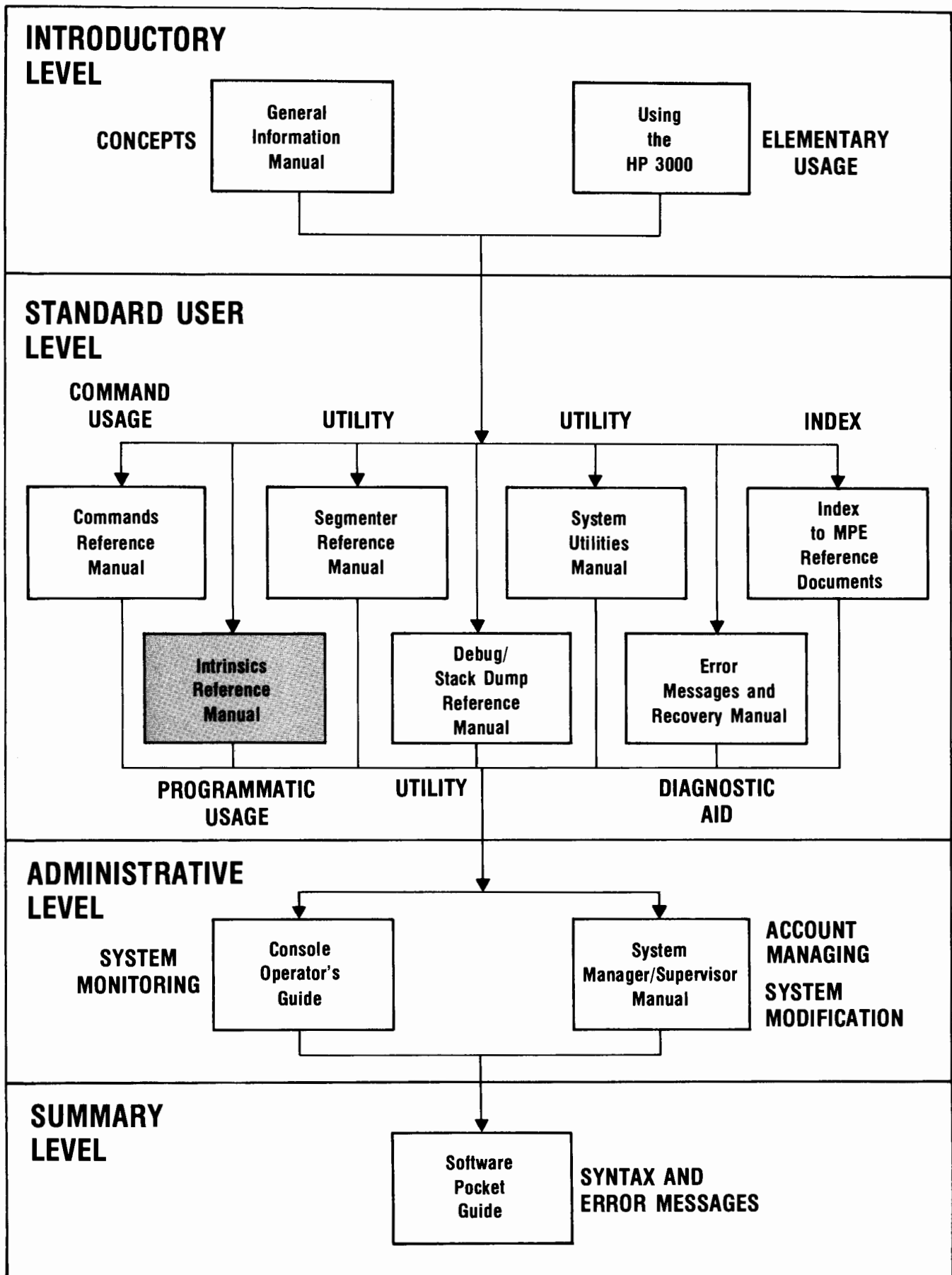
This manual is one of the set of manuals that document the Multiprogramming Executive-II Operating System (MPE-II). The manual plan on the next page indicates the position of this manual (shaded block) in the overall set.

This manual describes the set of intrinsics available with the MPE Operating System and tells you how to communicate with MPE programmatically. In addition, capabilities available to users with special capability-class attributes are described.

An introduction to MPE intrinsics is presented in Section I. The specifications for all intrinsics, in alphabetical order, are contained in Section II. Functional descriptions of the intrinsics, including those intrinsics for which special capabilities are required, are presented in the remaining sections, as follows:

Section III	Accessing and Altering Files.
Section IV	Utility Functions of MPE Intrinsics.
Section V	Device Characteristics.
Section VI	Resource Management.
Section VII	Process-Handling Capability.
Section VIII	Data Segment Management Capability.
Section IX	Privileged Mode Capability.
Section X	MPE Diagnostic Messages.

# MANUAL PLAN



## CONVENTIONS USED IN THIS MANUAL

The normal conventions (braces, brackets, etc.) used for MPE Commands do not apply for MPE intrinsic calls.

See page 2-1 for a description of the conventions used in this manual.



# CONTENTS

Section I	Page
<b>INTRODUCTION TO MPE INTRINSICS</b>	
Purposes and Uses of MPE Intrinsic	1-1
Intrinsic Calls	1-2
Calling Intrinsic from SPL	1-2
Calling Intrinsic from Other Languages	1-10
Intrinsic Call Errors	1-10
Optional Capabilities	1-12

Section II	Page
<b>INTRINSIC DESCRIPTIONS</b>	
ACTIVATE	2-4
ADJUSTUSLF	2-6
ALTDSEG	2-8
ARITRAP	2-10
ASCII	2-11
BINARY	2-13
CALENDAR	2-14
CAUSEBREAK	2-15
CLOCK	2-16
COMMAND	2-17
CREATE	2-18
CTRANSLATE	2-23
DASCII	2-25
DBINARY	2-27
DLSIZE	2-28
DMOVIN	2-30
DMOVOUT	2-32
EXPANDUSLF	2-34
FATHER	2-36
FCARD	2-36A
FCHECK	2-37
FCLOSE	2-41
FCONTROL	2-44
FGETINFO	2-48
FLOCK	2-56
FOPEN	2-58
FPOINT	2-69
FREAD	2-70
FREADDIR	2-72
FREADLABEL	2-74
FREADSEEK	2-75
FREEDSEG	2-76
FRELOCRIN	2-77
FRELATE	2-78
FRENAME	2-80
FSETMODE	2-82
FSPACE	2-84
FUNLOCK	2-85
FUPDATE	2-86
FWRITE	2-87
FWRITEDIR	2-91
FWRITELABEL	2-93
GETDSEG	2-94
GETJCW	2-96
GETLOCRIN	2-97
GETORIGIN	2-98
GETPRIORITY	2-99

GETPRIVMODE	2-100
GETPROCID	2-101
GETPROCINFO	2-102
GETUSERMODE	2-104
INTUSLF	2-105
IODONTWAIT	2-106A
IOWAIT	2-107
KILL	2-109
LOADPROC	2-110
LOCKGLORIN	2-111
LOCKLOCRIN	2-113
MAIL	2-115
MYCOMMAND	2-117
PAUSE	2-120
PRINT	2-121
PRINTFILEINFO	2-122
PRINTOP	2-123
PRINTOPREPLY	2-124
PROCTIME	2-126
PTAPE	2-127
QUIT	2-128
QUITPROG	2-129
READ	2-130
READX	2-131
RECEIVEMAIL	2-132
RESETCONTROL	2-134
SEARCH	2-135
SENDMAIL	2-136
SETJCW	2-138
SUSPEND	2-139
SWITCHDB	2-140
TERMINATE	2-141
TIMER	2-142
UNLOADPROC	2-143
UNLOCKGLORIN	2-144
UNLOCRIN	2-145
WHO	2-146
XARITRAP	2-149
XCONTRAP	2-151
XLIBTRAP	2-152
XSYSTRAP	2-153
ZSIZE	2-154

Section III	Page
<b>ACCESSING AND ALTERING FILES</b>	
File Management System	3-1
File Characteristics	3-2
Record Formats	3-3
File Device Relationships	3-6
Non-Sharable Device Access	3-6
File Domains	3-6
File Label	3-7
File Accessing	3-7
System Defined Files	3-7
User Pre-Defined (Back-Referenced) Files	3-8
New Files	3-8
Old Files	3-9
Input/Output Sets	3-9

# CONTENTS (continued)

<p>Accessing Files Already in Use ..... 3-10</p> <p>Files on Non-Sharable Devices ..... 3-13</p> <p>How to Use Files ..... 3-14</p> <p>Internal Operations for File Accessing ..... 3-14</p> <p>Opening Files ..... 3-24</p> <p>Opening a New Disc File ..... 3-24</p> <p>Opening an Old Disc File ..... 3-27</p> <p>Opening a File On a Device Other Than Disc... 3-29</p> <p>Issuing FREAD and FWRITE Intrinsic Calls for \$STDIN and \$STDLIST ..... 3-32</p> <p>Opening \$STDIN ..... 3-32</p> <p>Opening \$STDLIST ..... 3-34</p> <p>Closing Files ..... 3-35</p> <p>Closing a New File as a Temporary File ..... 3-36</p> <p>Closing a New File as a Permanent File ..... 3-38</p> <p>Renaming a File ..... 3-40</p> <p>Writing a File System Error-Check Procedure ..... 3-43</p> <p>Reading a File in Sequential Order ..... 3-43</p> <p>Writing Records into a File in Sequential Order... 3-46</p> <p>Reading a File in Direct-Access Mode ..... 3-47</p> <p>Optimizing Direct-Access File Reading ..... 3-49</p> <p>Writing Records into a File in Direct-Access Mode ..... 3-49</p> <p>Locking and Unlocking Files ..... 3-52</p> <p>Updating a File ..... 3-54</p> <p>Using the IOWAIT Intrinsic ..... 3-57</p> <p>Writing and Reading User File Labels ..... 3-59</p> <p>Writing a User File Label ..... 3-59</p> <p>Reading a User File Label ..... 3-60</p> <p>Obtaining File Access Information ..... 3-63</p> <p>Obtaining File-Error Information ..... 3-65</p> <p>Magnetic Tape Considerations ..... 3-65</p> <p>FWRITE ..... 3-65</p> <p>FREAD ..... 3-67</p> <p>FSPACE ..... 3-67</p> <p>FCONTROL (Write EOF) ..... 3-67</p> <p>FCONTROL (Forward Space to File Mark) ..... 3-67</p> <p>FCONTROL (Backward Space to File Mark) ..... 3-67</p> <p>End-of-File Marks on Magnetic Tape ..... 3-68</p> <p>Spacing File Marks ..... 3-68</p> <p>Using the FCLOSE Intrinsic with Magnetic Tape ..... 3-69</p> <p>Updating Magnetic Tape Files ..... 3-69</p> <p>Reading and Writing a Magnetic Tape File ..... 3-72</p> <p>Spacing on Disc or Tape Files ..... 3-75</p> <p>Directing File Control Operations ..... 3-76</p> <p>Resetting the Logical Record Pointer ..... 3-77</p> <p>Declaring Access-Mode Options ..... 3-77</p> <p>Determining Interactive and Duplicative File Pairs ..... 3-78</p> <p><b>Section IV</b> <span style="float: right;">Page</span></p> <p><b>UTILITY FUNCTIONS OF MPE INTRINSICS</b></p> <p>Dynamic Loading and Unloading of Library Procedures ..... 4-2</p> <p>Dynamic Loading ..... 4-2</p> <p>Dynamic Unloading ..... 4-3</p>	<p>Searching Arrays ..... 4-3</p> <p>Formatting Command Parameters ..... 4-4</p> <p>Executing MPE Commands Programmatically ..... 4-9</p> <p>Determining the User's Access Mode and Attributes ..... 4-10</p> <p>Converting Numbers from Binary Code to ASCII Strings ..... 4-10</p> <p>Converting Numbers from an ASCII Numeric String to a Binary Coded Value ..... 4-13</p> <p>Translating Characters from EBCDIC to ASCII or from ASCII to EBCDIC ..... 4-13</p> <p>Transmitting Program Input/Output from Job/Session.Input/Output Devices ..... 4-16</p> <p>Reading Input from the Job/Session Input Device ..... 4-16</p> <p>Writing Output to the Job/Session List Device .. 4-18</p> <p>Writing Output to the Operator's Console ..... 4-18</p> <p>Writing Output to the Operator's Console and Requesting a Reply ..... 4-18</p> <p>Suspending the Calling Process ..... 4-19</p> <p>Requesting a Process Break ..... 4-19</p> <p>Terminating a Process ..... 4-20</p> <p>Aborting a Process ..... 4-20</p> <p>Aborting a Program ..... 4-20</p> <p>Changing Stack Sizes ..... 4-22</p> <p>Changing the DL to DB Area Size ..... 4-22</p> <p>Changing the Z to DB Area Size ..... 4-27</p> <p>Enabling and Disabling Traps ..... 4-29</p> <p>Arithmetic Traps ..... 4-30</p> <p>Standard Traps ..... 4-31</p> <p>Extended Precision Floating-Point Traps ..... 4-31</p> <p>Commercial Instruction Traps ..... 4-32</p> <p>Library Trap ..... 4-34</p> <p>System Trap ..... 4-35</p> <p>Control-Y Traps ..... 4-38</p> <p>Time and Date Ininsics ..... 4-42</p> <p>Obtaining System Timer Information ..... 4-42</p> <p>Obtaining the Current Time ..... 4-44</p> <p>Obtaining the Calendar Date ..... 4-44</p> <p>Obtaining Process Run Time (Use of the Central Processor) ..... 4-44</p> <p>Interprocess Communication ..... 4-44</p> <p><b>Section V</b> <span style="float: right;">Page</span></p> <p><b>DEVICE CHARACTERISTICS</b></p> <p>Device Characteristics ..... 5-1</p> <p>Paper Tape Reader ..... 5-1</p> <p>Binary Mode ..... 5-1</p> <p>ASCII Mode ..... 5-1</p> <p>Paper Tape Punch ..... 5-3</p> <p>Binary Mode ..... 5-3</p> <p>ASCII Mode ..... 5-3</p> <p>Card Reader ..... 5-3</p> <p>Line Printer ..... 5-3</p> <p>Magnetic Tape ..... 5-4</p> <p>Printing Reader/Punch ..... 5-4</p>
---	--

# CONTENTS (continued)

Line Printer and Terminal Carriage-Control Codes .....	5-6	Interprocess Communication .....	7-10
End-of-File Indication .....	5-6	Testing Mailbox Status .....	7-10
Terminals .....	5-8	Sending Mail .....	7-11
Terminal Types .....	5-8	Receiving (Collecting) Mail .....	7-12
Special Keys .....	5-9	Avoiding Deadlocks .....	7-13
Changing Terminal Characteristics .....	5-10	Rescheduling Processes .....	7-13
Changing Terminal Speed .....	5-10	Determining Source of Activation .....	7-14
Changing Input Echo Facility .....	5-11	Determining Father Process .....	7-14
Enabling and Disabling System Break Function .....	5-13	Determining Son Processes .....	7-15
Enabling and Disabling Subsystem Break Function .....	5-14	Determining Process Priority and State .....	7-15
Enabling and Disabling Parity Checking ..	5-14		
Enabling and Disabling Tape-Mode Option .....	5-15	Section VIII .....	Page
Enabling and Disabling the Terminal Input Timer .....	5-16	<b>DATA SEGMENT MANAGEMENT CAPABILITY</b>	
Reading the Terminal Input Timer .....	5-19	Creating an Extra Data Segment .....	8-2
Defining Line-Termination Characters for Terminal Input .....	5-20	Deleting an Extra Data Segment .....	8-15
Enabling and Disabling Binary Transfers ..	5-21	Transferring Data from an Extra Data Segment to the Stack .....	8-15
Enabling and Disabling User Block Transfers .....	5-22	Transferring Data from the Stack to an Extra Data Segment .....	8-15
Enabling and Disabling Line Deletion Echo Suppression .....	5-23	Changing the Size of an Extra Data Segment .....	8-15
Setting Parity .....	5-23		
Allocating a Terminal .....	5-24	Section IX .....	Page
Setting Terminal Type .....	5-25	<b>PRIVILEGED MODE CAPABILITY</b>	
Obtaining Terminal Type Information .....	5-25	Permanently Privileged Programs .....	9-1
Obtaining Terminal Output Speed .....	5-26	Temporarily Privileged Programs .....	9-2
Setting Unedited Terminal Mode .....	5-26	Entering Privileged Mode .....	9-3
Reading Paper Tapes Without X-OFF Control .....	5-27	Entering Non-Privileged Mode .....	9-5
Using the FCARD Intrinsic to Operate the HP 7260A Optical Mark Reader .....	5-28	Moving the DB Pointer .....	9-5
		Scheduling Processes .....	9-5
Section VI .....	Page		
<b>RESOURCE MANAGEMENT</b>		Section IX .....	Page
Inter-Job Level (Global) RIN's .....	6-2	<b>MPE DIAGNOSTIC MESSAGES</b>	
Acquiring Global RIN's .....	6-2	Run-Time Messages .....	10-2
Releasing Global RIN's .....	6-3	User Messages .....	10-12
Locking and Unlocking Global RIN's .....	6-3	Operator Messages .....	10-13
Inter-Process (Local) Level RIN's .....	6-6	System Messages .....	10-13
Acquiring Local RIN's .....	6-8	File Information Display .....	10-14
Locking and Unlocking Local RIN's .....	6-8		
Freeing Local RIN's .....	6-9	Appendix A .....	Page
Section VII .....	Page	<b>ASCII CHARACTER SET</b> .....	A-1
<b>PROCESS-HANDLING CAPABILITY</b>			
Processes .....	7-1	Appendix B .....	Page
Organization of User Processes .....	7-2	<b>DISC FILE LABELS</b> .....	B-1
Process Substates .....	7-2		
Process to Process Communication .....	7-2	Appendix C .....	Page
Creating and Activating Processes .....	7-3	<b>END-OF-FILE INDICATION</b> .....	C-1
Suspending Processes .....	7-8		
Deleting Processes .....	7-8	<b>INDEX</b> .....	I-1

# ILLUSTRATIONS

Title	Page	Title	Page
Calling the PRINTOP Intrinsic from SPL .....	1-9	Using the FCLOSE Intrinsic with Magnetic Tape Files .....	3-70
Using Numeric Values as Parameters in an Intrinsic Call .....	1-9	Magnetic Tape Example .....	3-73
Condition Code Checks .....	1-11	Using the MYCOMMAND Intrinsic .....	4-5
Foptions Bit Summary .....	2-49	Using the WHO Intrinsic .....	4-11
Aoptions Bit Summary .....	2-51	Using the ASCII Intrinsic .....	4-12
Carriage-Control Directives .....	2-89	Using the DASCII Intrinsic .....	4-14
Carriage-Control Summary .....	2-90	Using the BINARY Intrinsic .....	4-15
Actions Resulting from Multiple Access of Files .....	3-11	Using the PRINT and READ Intronics .....	4-17
File Access Interface for New Disc Files .....	3-15	Using the QUIT Intrinsic .....	4-21
File Name and Sector Address Storage .....	3-18	Expanding and Contacting the DL to DB Area ...	4-23
File Access Interface for Old Disc Files .....	3-19	Using the DLSIZE Intrinsic .....	4-25
Device Allocation Flowchart .....	3-23	Changing the DL to DB Area Size .....	4-28
Opening a New Disc File .....	3-25	Using the XARITRAP Intrinsic .....	4-33
Opening an Old Disc File .....	3-28	Using the XCONTRAP Intrinsic .....	4-41
Opening a File on a Device Other Than Disc .....	3-30	Using the TIMER Intrinsic .....	4-43
Opening \$STDIN and \$STDLIST .....	3-33	Echo Facility vs Duplex Mode .....	5-12
Closing a New File as a Temporary File .....	3-37	Using the FCONTROL Intrinsic to Enable and Read the Terminal Input Timer .....	5-18
Closing a New File as a Permanent File .....	3-39	FCARD Intrinsic Example .....	5-30
FRENAME Intrinsic Example .....	3-41	Using the LOCKGLORIN and UNLOCKGLORIN Intronics .....	6-4
Error-Check Procedure Example .....	3-44	Using the CREATE and ACTIVATE Intronics ...	7-4
FREAD and FWRITE Intronics Example .....	3-45	Process Deletion .....	7-9
FREADDIR and FREADSEEK Intronics Example .....	3-48	Using the GETDSEG and DMOVOUT Intronics (Program DSINIT) .....	8-3
FWRITEDIR Intrinsic Example .....	3-51	Creating and Activating Two Son Processes (Program DSBOSS) .....	8-4
FLOCK and FUNLOCK Intronics Example .....	3-53	Using the GETDSEG and DMOVIN Intronics (Program DSACCS) .....	8-5
FUPDATE Intrinsic Example .....	3-56	Array CALENDAR .....	8-8
Using the IOWAIT Intrinsic .....	3-58	Using the GETPRIVMODE and GETUSERMODE Intronics .....	9-4
FWRITELABEL Intrinsic Example .....	3-61	MPE Master Queue Structure .....	9-6
FREADLABEL Intrinsic Example .....	3-62		
FGETINFO Intrinsic Example .....	3-64		
FCHECK Intrinsic Example .....	3-66		

# TABLES

Title	Page	Title	Page
Summary of MPE Intronics .....	1-3	File System Error Table .....	10-8
Device-Dependent Restrictions .....	3-21	Loader Error Table .....	10-9
Classification of Devices .....	3-22	Create Error Table .....	10-11
Line Printer Differences .....	5-4	Activate Error Table .....	10-11
Line Printer and Terminal Carriage-Control Codes ..	5-7	Suspend Error Table .....	10-12
Program Error Table .....	10-5	Mycommand Error Table .....	10-12
Intrinsic Table .....	10-6	Lockglorin Error Table .....	10-12
Run-Time Error Table .....	10-7	System Messages .....	10-13



# INTRODUCTION TO MPE INTRINSICS

SECTION

I

In the MPE Operating System, individual programming operations are handled by sets of code known as *procedures*. These procedures are coded in SPL (Systems Programming Language for the HP 3000 Series II Computer System) and are defined by a procedure declaration consisting of

- A procedure head, containing the procedure name and type, parameter definitions, and other information about the procedure.
- A procedure body, containing executable statements and data declarations local to this procedure.

As part of their function, several procedures also return values to the processes that invoke them.

## NOTE

A *process* is the basic executable entity in MPE. A process is not a program itself, but the unique execution of a program by a particular user at a particular time. See the HP 3000 Series II Computer System *General Information Manual* for a more complete discussion of processes.



Each procedure is invoked by a corresponding *procedure call*. When a procedure call is encountered in a program, control is transferred to the procedure. The procedure runs until an exit is encountered, at which time control returns to the statement following the procedure call.

In addition to the procedures provided by the operating system, MPE allows the user to write special-purpose procedures in SPL. To distinguish MPE *system procedures* (which are always available to the user, either directly or indirectly) from any other procedures, the term *intrinsic* is applied to MPE system procedures. Similarly, the term *intrinsic call* is used to denote the procedure call that references an MPE system procedure.

## PURPOSES AND USES OF MPE INTRINSICS

With MPE intrinsics, it is possible to

- Access and alter files. Files can be opened, read, written on, updated, and otherwise manipulated using intrinsics.
- Request various utility operations, such as:

- Listing date, time, and accounting information.
- Determining job status.
- Determining device status.
- Obtaining devicefile information.
- Transmitting messages.

Inserting comments in command stream.  
Requesting ASCII/binary number conversion.  
Reading input from job/session input device.  
Writing output to job/session list device.  
Obtaining system timer information.  
Determining the user's access mode and attributes.  
Searching arrays and formatting parameters.  
Executing MPE commands programmatically.  
Enabling and disabling error traps.  
Requesting program break, termination, or abort.  
Changing the lengths of the user-managed area (DL to DB) and stack area (Z to DL) and altering DL to DB and Z to DL register offsets.  
Managing interprocess communication through the job control word.  
Changing terminal speed and echo mode.

- Access and manage a system *resource* such as an input/output device, file, program, subroutine, procedure, code segment, or the data stack such that no other program may use the resource simultaneously.
- In addition, users with certain *optional capabilities* (see OPTIONAL CAPABILITIES, page 1-12) may use intrinsics to

Create and delete processes.  
Activate and suspend processes.  
Send information (mail) between processes.  
Change the scheduling of processes.  
Obtain information about existing processes.  
Create and access extra data segments.  
Lock as many resources as desired simultaneously.

To help you determine what you can accomplish with MPE intrinsics, a summary is presented in table 1-1. Table 1-1 lists each intrinsic, and the capability necessary to use it.

## INTRINSIC CALLS

Intrinsic calls invoke MPE system procedures which are requested *programmatically* (that is, from within a user program). In SPL programs (see CALLING INTRINSICS FROM SPL, below), you write the intrinsic calls *explicitly*. In FORTRAN, COBOL, BASIC, and RPG programs, for most general applications, the compiler for that language generates any necessary intrinsic calls automatically — they are invisible to you. It is possible, however, to call intrinsics directly from these languages (see CALLING INTRINSICS FROM OTHER LANGUAGES, page 1-10).

All MPE intrinsics are treated as external procedures by user programs. External linkages from user programs to intrinsics are satisfied when the user programs are segmented (at PREPARATION time) and allocated residence in virtual memory (at RUN time). See the *MPE Segmenter Reference Manual* for a discussion of *segments*, *segmentation*, and *allocation*.

### CALLING INTRINSICS FROM SPL

Before an intrinsic can be called from an SPL program, it must be *declared* at the beginning of the program, following all data declarations, like any other SPL procedure. This could be done by

Table 1-1. Summary of MPE Intrinsic

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
ACTIVATE	Activates a process.	Process Handling
ADJUSTUSLF	Adjusts directory space in a USL file.	Standard
ALTDSEG	Alters the size of an extra data segment.	Data-Segment Management
ARITRAP	Enables or disables internal interrupt signals from all hardware arithmetic traps.	Standard
ASCII	Converts a number from binary to ASCII code.	Standard
BINARY	Converts a number from ASCII to binary code.	Standard
CALENDAR	Returns the calendar date.	Standard
CAUSEBREAK	Requests a session break	Standard
CLOCK	Returns the actual time.	Standard
COMMAND	Executes an MPE command programmatically.	Standard
CREATE	Creates a process.	Process Handling
CTRANSLATE	Converts a string of characters from EBCDIC to ASCII or from ASCII to EBCDIC.	Standard
DASCII	Converts a value from double-word binary to ASCII code.	Standard
DBINARY	Converts a number from ASCII code to a double-word binary value.	Standard
DLSIZE	Changes size of DL to DB area.	Standard
DMOVIN	Copies block from data segment to stack.	Data-Segment Management
DMOVOUT	Copies block from stack to data segment.	Data-Segment Management
EXPANDUSLF	Changes length of a USL file.	Standard
FATHER	Requests Process Identification Number (PIN) of father process.	Process Handling
FCHECK	Requests details about file input/output errors.	Standard
FCLOSE	Closes a file.	Standard
FCONTROL	Performs control operations on a file or terminal device.	Standard



Table 1-1. Summary of MPE Intrinsic (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FGETINFO	Requests access and status information about a file.	Standard
FLOCK	Dynamically locks a file.	Standard
FOPEN	Opens a file.	Standard
FPOINT	Resets the logical record pointer for a sequential disc file.	Standard
FREAD	Reads a logical record from a sequential file (on any device) to the user's data stack.	Standard
FREADDIR	Reads a logical record from a direct-access file to the user's data stack.	Standard
FREADLABEL	Reads a user file label.	Standard
FREADSEEK	Prepares, in advance, for reading from a direct-access file.	Standard
FREEDSEG	Releases an extra data segment.	Data-Segment Management
FREELOCRIN	Frees all local Resource Identification Numbers (RIN's) from allocation to a job.	Standard
FRELATE	Determines if a file pair is interactive or duplicative.	Standard
FRENAME	Renames a disc file.	Standard
FSETMODE	Activates or de-activates file-access modes.	Standard
FSPACE	Spaces forward or backward on a file.	Standard
FUNLOCK	Dynamically unlocks a file.	Standard
FUPDATE	Updates a logical record residing in a disc file.	Standard
FWRITE	Writes a logical record from the user's stack to a sequential file (on any device).	Standard
FWRITEDIR	Writes a logical record from the user's stack to a direct-access disc file.	Standard
FWRITELABEL	Writes a user's file label.	Standard
GETDSEG	Creates an extra data segment.	Data-Segment Management
GETJCW	Fetches contents of job control word.	Standard
GETLOCRIN	Acquires local RIN's.	Standard

Table 1-1. Summary of MPE Intrinsic (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
GETORIGIN	Determines source of process activation call.	Process Handling
GETPRIORITY	Changes the priority of a process.	Process Handling
GETPRIVMODE	Dynamically enters privileged mode.	Privileged Mode
GETPROCID	Requests PIN of a son process.	Process Handling
GETPROCINFO	Requests status information about a father or son process.	Process Handling
GETUSERMODE	Dynamically returns to non-privileged mode.	Privileged Mode
INITUSLF	Initializes a USL file to the empty state.	Standard
IOWAIT	Initiates completion operations for an I/O request.	Standard
KILL	Deletes a process.	Process Handling
LOADPROC	Dynamically loads a library procedure.	Standard
LOCKGLORIN	Locks a global RIN.	Standard
LOCKLOCRIN	Locks a local RIN.	Standard
MAIL	Tests mailbox status.	Process Handling
MYCOMMAND	Parses (delineates and defines parameters) for user-supplied command image.	Standard
PAUSE	Suspends calling process for a specified number of seconds.	Standard
PRINT	Prints character string on job/session list device.	Standard
PRINTOP	Prints a character string on the Operator's Console.	Standard
PRINTOPREPLY	Prints a character string on Operator's Console and solicits a reply.	Standard
PROCTIME	Returns a process' accumulated central processor time.	Standard
PTAPE	Accepts input from paper tapes which do not contain X-OFF control characters.	Standard
QUIT	Aborts a process.	Standard
QUITPROG	Aborts the user process structure.	Standard

Table 1-1. Summary of MPE Intrinsic (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
READ	Reads an ASCII string from the job/session input device (\$STDIN).	Standard
READX	Reads an ASCII string from the job/session input device (STDINX).	Standard
RECEIVEMAIL	Receives mail from another process.	Process Handling
RESETCONTROL	Resets terminal to accept CONTROL-Y signal.	Standard
SEARCH	Searches an array for a specified entry or name.	Standard
SENDMAIL	Sends mail to another process.	Process Handling
SETJCW	Sets bits in job control word.	Standard
SUSPEND	Suspends a process.	Process Handling
SWITCHDB	Switches DB-register pointer.	Privileged Mode
TERMINATE	Terminates a process.	Standard
TIMER	Returns system timer bit count.	Standard
UNLOADPROC	Dynamically unloads a library procedure.	Standard
UNLOCKGLORIN	Unlocks a global RIN.	Standard
UNLOCKLOCRIN	Unlocks a local RIN.	Standard
WHO	Returns user attributes.	Standard
XARITRAP	Arms or disarms the software arithmetic trap.	Standard
XCONTRAP	Arms or disarms the CONTROL-Y trap.	Standard
XLIBTRAP	Arms or disarms the software library trap.	Standard
XSYSTRAP	Arms or disarms the system trap.	Standard
ZSIZE	Changes size of Z to DB area.	Standard

writing the entire intrinsic declaration but, because some intrinsic declarations are rather long, you can save time by declaring intrinsics with the *INTRINSIC declaration statement*.

The format of the INTRINSIC declaration statement is

```
INTRINSIC intrinsicname, intrinsicname, . . . ,intrinsicname;
```

In the *intrinsicname* list, you name all intrinsics that you intend to call within your program. When more than one intrinsic is named, the names must be separated by commas. For example, to use the INTRINSIC declaration statement to declare the FOPEN, FREAD, FWRITE, and FCLOSE intrinsics, you could write

```
INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE;
```

Regardless of whether you declare an intrinsic as a procedure or in an INTRINSIC declaration statement, you must know the number and type of parameters which the intrinsic uses in order to call it correctly. Parameters can be passed to a procedure (intrinsic) either by *value* or by *reference*. When a parameter is passed by reference (the default case), its address in the caller's data area is made available to the called procedure. If the variable is changed by the called procedure, the storage in the caller's data area is updated. When a parameter is passed by value, the called procedure receives a local (private) copy of the actual data value. If the called procedure changes this private copy, the corresponding variable in the calling routine remains unchanged.

You call an intrinsic in your program exactly as you would any SPL procedure: that is, you write the intrinsic name, followed by a parameter list enclosed in parentheses. These parameters must follow the positional format shown in each intrinsic description (Section II). Parameters must be separated from each other by commas. For example, a call to the FREAD intrinsic could be written as

```
FREAD(FN,TAR,TC);
```

where the *filenum*, *target*, and *tcount* parameters (see Section II, page 2-70) are represented by FN, TAR, and TC, respectively. If numeric values are to be specified for the *filenum* and *tcount* parameters (which are VALUE parameters), the following call could be used:

```
FREAD(3,TAR,-80);
```

If the OPTION VARIABLE notation appears in the intrinsic description shown in Section II, some of the intrinsic parameters are optional. Since all intrinsic parameters are *positional*, however, you must indicate a missing parameter *within* a parameter list by omitting the parameter itself but retaining the preceding and following commas. For example, if the second parameter is missing

```
FOPEN(FILENAME,,3);
```

If the first parameter is omitted from a list, this is indicated by following the left parenthesis with a comma. If one or more parameters are omitted from the *end* of a list, however, this is indicated by simply writing the terminating right parenthesis after the last parameter included.

## NOTE

In some intrinsic calls, input parameters are passed to the intrinsic as words whose individual bits or fields of bits signify certain functions or options. In cases where some of the bits within a word are described in this manual as “reserved for MPE”, you are advised to set such bits to zero. This will help insure the compatibility of your current program with future releases of MPE.

In cases where output parameters are passed by an intrinsic to words referenced by a calling program, bits within such words that are described as “reserved for MPE” are set to zero unless otherwise noted in the discussion of the particular parameter.

To call an intrinsic from an SPL program, follow the steps listed below:

1. Refer to the intrinsic description in Section II to determine the parameter types and their positions in the parameter list.
2. Declare the variables or array names to be passed as parameters by type at the beginning of the program.
3. Include the name of the intrinsic in an INTRINSIC declaration statement.
4. Issue the intrinsic call at the appropriate place in your program.

For example, refer to Section II, page 2-123 for a description of the PRINTOP intrinsic. This intrinsic is shown as

```
      A   IV   IV  
PRINTOP(message,length,control);
```

The *bold face italics* shown for *message*, *length*, and *control* signify that these are required parameters. (Optional parameters are signified by *regular italics*.)

The superscripts A, IV, and IV over *message*, *length*, and *control* denote array, integer by value, and integer by value, respectively.

The array name to be used as the *message* parameter must be declared as an array at the beginning of the program. If variable names are used for the *length* and *control* parameters, they must be declared as type integer at the beginning of the program.

Figure 1-1 shows the intrinsic PRINTOP being called from an SPL program after being declared with the INTRINSIC declaration statement. Note that MESSAGE is declared as an array and the variables LENGTH and CONTROL are declared as type integer.

Figure 1-2 shows the same intrinsic being called with numeric values, instead of symbolic identifiers, being specified for the parameters *length* and *control*.

```
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0
00003000 00000 0 << USING THE INTRINSIC DECLARATION STATEMENT >>
00004000 00000 0
00005000 00000 0 BEGIN
00006000 00000 1 ARRAY MESSAGE(0:9):="MESSAGE TO OPERATOR ";
00007000 00012 1 INTEGER LENGTH, CONTROL;
00008000 00012 1 INTRINSIC PRINTOP;
00009000 00012 1
00010000 00012 1 LENGTH:=10;
00011000 00002 1 CONTROL:=%60;
00012000 00004 1 PRINTOP(MESSAGE, LENGTH, CONTROL);
00013000 00010 1
00014000 00010 1 END.
PRIMARY DB STORAGE=%003; SECONDARY DB STORAGE=%00012
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:00; ELAPSED TIME=0:01:23
```

Figure 1-1. Calling the PRINTOP Intrinsic from SPL

```
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0
00003000 00000 0 << USING NUMERIC VALUES AS PARAMETERS >>
00004000 00000 0
00005000 00000 0 BEGIN
00006000 00000 1 ARRAY MESSAGE(0:9):="MESSAGE TO OPERATOR ";
00007000 00012 1 INTRINSIC PRINTOP;
00008000 00012 1
00009000 00012 1 PRINTOP(MESSAGE, 10, %60);
00010000 00004 1
00011000 00004 1 END.
PRIMARY DB STORAGE=%001; SECONDARY DB STORAGE=%00012
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:00; ELAPSED TIME=0:00:53
```

Figure 1-2. Using Numeric Values as Parameters in an Intrinsic Call

## CALLING INTRINSICS FROM OTHER LANGUAGES

For most applications in FORTRAN, COBOL, BASIC, and RPG programs, the compiler for the specific language generates any necessary intrinsic calls automatically. It is possible, however, to call intrinsics, or other library procedures, from these languages. The procedures for calling intrinsics from these languages are described in the applicable language reference manuals.

## INTRINSIC CALL ERRORS

Some intrinsics alter the *condition code* returned to FORTRAN and SPL programs through two bits (6 and 7) in the Status register. These two bits have four states which are defined as follows:

- 00 Defined as CCG, or condition code *greater than*.
- 01 Defined as CCL, or condition code *less than*.
- 10 Defined as CCE, or condition code *equal*.
- 11 Undefined.

Since bits 6 and 7 of the Status register are affected by many instructions, you should check for condition codes immediately upon return from an intrinsic (see figure 1-3). A condition code is always CCG, CCL, or CCE, and has the general meaning indicated below. The specific meaning, of course, depends upon the intrinsic called and these meanings are described in Section II.

Condition Code State	General Meaning
CCE	Condition code equal. This generally indicates that the request was granted.
CCG	Condition code greater. A special condition occurred but may not have affected the execution of the request. (For example, the request was executed, but default values were assumed as intrinsic call parameters.)
CCL	Condition code less. The request was not granted, but the error condition may be recoverable. Beyond this condition code, some intrinsics return further error information in the program through their return values.

Two types of errors may occur when an intrinsic is executed. The first, denoted by the CCG or CCL condition codes, is generally recoverable (control returns to the calling program) and is known as a *condition code error*. The second type is an *abort error*, which occurs when a calling program passes illegal parameters to an intrinsic, or does not have the capability demanded by the intrinsic. Intrinsic (system) traps are handled by a special procedure designed for that purpose. Normally, if an intrinsic causes the trap to be invoked, the system trap handler aborts the user program. You may, however, specify a procedure to be used instead of the default system trap handler and try to recover from such errors. If the program is aborted in a batch job, MPE removes the job from the system (unless a :CONTINUE command, defined in the *MPE Commands Reference Manual*, precedes the error). If the program is aborted in an interactive session, MPE returns control to the terminal. Abort-error messages are described in Section X.



```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0
00003000 00000 0 << CONDITION CODE CHECKS >>
00004000 00000 0
00005000 00000 0 BEGIN
00006000 00000 1 ARRAY MESSAGE(0:9):="MESSAGE TO OPERATOR ";
00007000 00012 1 ARRAY OKBUF(0:9):="MESSAGE TRANSMITTED ";
00008000 00012 1 ARRAY ERRBUF(0:8):="I/O ERROR OCCURRED";
00009000 00011 1 INTRINSIC PRINTOP,PRINT;
00010000 00011 1
00011000 00011 1 PRINTOP(MESSAGE,10,%60);
00012000 00004 1
00013000 00004 1 IF = THEN GOTO OK;
00014000 00005 1 IF < THEN GOTO ERR;
00015000 00006 1
00016000 00006 1 OK:
00017000 00006 1 PRINT(OKBUF,10,%60);
00018000 00012 1 GOTO STOP;
00019000 00013 1
00020000 00013 1 ERR:
00021000 00013 1 PRINT(ERRBUF,9,%60);
00022000 00017 1
00023000 00017 1 STOP:
00024000 00017 1 END.
PRIMARY DB STORAGE=%003; SECONDARY DB STORAGE=%00035
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:01; ELAPSED TIME=0:01:55
    
```

Figure 1-3. Condition Code Checks

NOTE

Whenever an intrinsic is invoked by a process and the DB register is pointing to the DB area in the user's stack, a bounds check takes place to insure that all parameters in the intrinsic call reference addresses that lie between the DL and S addresses in the stack (prior to the intrinsic call). If an address outside of these boundaries is referenced, an abort error occurs.

When an intrinsic is invoked by a process running in the privileged mode, and the DB register points to a data segment other than the user's stack segment (split stack), the results depend on the particular intrinsic. Most intrinsics abort immediately in this case. Others, indicated in Section II, are



allowed to execute following a bounds check that insures that all parameters in the intrinsic call reference addresses that lie within the data segment. Any boundary violation results in an abort error. Any additional special actions taken by a particular intrinsic are described in the discussion of that intrinsic in Section II.

Figure 1-3 illustrates the use of condition code checks in a program. If the condition code is CCE (meaning that the request was granted), the program displays "MESSAGE TRANSMITTED". For a CCL condition code, the message "I/O ERROR OCCURRED" is displayed and the program terminates normally.

## OPTIONAL CAPABILITIES

Users with the *Standard MPE Capability* can perform most functions available through the operating system. There are some functions, however, which can only be performed by users with certain *optional capabilities* assigned to them when the Accounts, Groups, and Users are created by the System Manager.

The *Process-Handling Optional Capability* allows you to programmatically

- Create and delete processes.
- Activate and suspend processes.
- Send mail between processes.
- Change the scheduling of processes.
- Obtain information about existing processes.

The *Process-Handling Optional Capability* is described in Section VII.

The *Data-Segment Management Optional Capability* allows you to create and access extra data segments from processes during a job or session. This capability is described in Section VIII.

*Multiple Resource Identification Number Optional Capability.* Users having standard MPE capability can lock only one global or local Resource Identification Number (RIN) at a time. The Multiple Resource Identification Number Optional Capability, however, allows you to lock as many RIN's as desired simultaneously, without checking by the operating system. The Multiple RIN Optional Capability is described in Section VI.

The *Privileged Mode Optional Capability* allows you to access all areas of the system and use all features of the hardware. This capability allows you to access all system tables and invoke all system instructions, including those in the privileged central processor unit instruction set. In short, this capability allows you to use the computer on the same terms as the operating system itself. The *Privileged Mode Optional Capability* is described in Section IX.

#### IMPORTANT NOTE

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard will investigate and attempt to resolve problems resulting from the use of privileged mode code. This service, which is not provided under the standard Service Contract, is available on a time and materials billing basis. However, Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.



# INTRINSIC DESCRIPTIONS

SECTION

II

This section contains descriptions of all intrinsics, arranged alphabetically. Each intrinsic description includes the following information:

- The intrinsic name, a brief summary of its function, and the number of the intrinsic.
- The complete intrinsic call description highlighted by being enclosed in a shaded box. The intrinsic call descriptions are in the format shown below for the `ACTIVATE` intrinsic:

```
IV LV O-V  
ACTIVATE(pin,susp);
```

Required parameters, such as *pin*, are shown in *bold face italics*; optional parameters (*susp*) are shown in *regular italics*. Superscripts are used to describe the types of parameters and whether they must be passed by *value*, instead of by *reference* (the default case). See Section I, page 1-7 for a discussion of passing parameters by value and by reference. The superscripts have the following meanings:

BA Byte array  
BP Byte pointer  
D Double  
DA Double array  
DV Double by value  
I Integer  
IA Integer array  
IV Integer by value  
L Logical  
LA Logical array  
LV Logical by value  
O-P Option privileged  
O-V Option variable  
R Real

In addition to the superscripts shown over the parameters, the superscript O-V is shown for some intrinsics to denote *option variable*. Option variable means that the intrinsic contains *optional* parameters. Additionally, O-P is shown for those intrinsics which can be called only when running in privileged mode. The `ACTIVATE` intrinsic shown, therefore, contains two parameters: *pin*, which is a required integer parameter that must be passed by value; and *susp*, an optional logical parameter that, if included in the intrinsic call, must be passed by value. Additionally, the intrinsic is *option variable*, meaning that some parameters are optional.

- **FUNCTIONAL RETURN:** For those intrinsics which return a value to the calling program (type procedures), the return is described. If the intrinsic is not a type procedure, this portion of the description is omitted. The intrinsic call description format for type intrinsics is as shown below for the READ intrinsic:

<sup>I</sup>                    <sup>LA</sup>                    <sup>IV</sup>  
*length:=READ(message,expectedl);*

The READ intrinsic returns the positive length of the input actually read. This value is returned to an integer variable. In the intrinsic call description, a word, representing what is returned, is shown in italics (as is *length*, above) to denote that the intrinsic is a *type procedure*. The type (integer, double, etc.) is signified by a superscript above the descriptive word. Thus,

<sup>I</sup>                    <sup>LA</sup>                    <sup>IV</sup>  
*length:=READ(message,expectedl);*

is an *integer* procedure, *message* is a *required* logical array, and *expectedl* is a *required* integer parameter which must be passed by value.

#### NOTE

:= means “is assigned” or “is replaced by.”

- **PARAMETERS:** All parameters are described. In the intrinsic call description, required parameters are shown in *bold face italics* and optional parameters are shown in *regular italics*. Elsewhere in this manual, this distinction is not shown for required and optional parameters and all parameters are shown in *regular italics*.
- **CONDITION CODES:** Condition codes are included for each intrinsic.
- **SPECIAL CONSIDERATION:**

*Required Capability.* When you run a program file, the program file’s capability (established at PREPARATION time) is checked against the capability of the group in which the file resides. If the file’s capability does not exceed the capability of the group, the program executes. Additional capability checking, however, is done if the program calls an intrinsic. Some intrinsics require that the program file have sufficient capability to call them. If an intrinsic requires a special capability, it will be noted in the discussion of that intrinsic.

#### NOTE

The optional capabilities are discussed in Section I, page 1-12.

*Split Stack Operations.* During normal operation, the DB register points to the user process' stack. Some operations with extra data segments require that DB be set to the base of the extra data segment while DL and all other data registers remain associated with the stack. When a process is operating in this mode it is said to have a *split stack*. Several of the MPE intrinsics deal with DB in this manner and you need not be concerned with the mechanics of the operation because while the stack is "split" only system code is executing. It is possible, however, if you are a privileged user, to force your process to operate in split-stack mode explicitly by calling the SWITCHDB intrinsic.

#### IMPORTANT NOTE

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard will investigate and attempt to resolve problems resulting from the use of privileged mode code. This service, which is not provided under the standard Service Contract, is available on a time and materials billing basis. However, Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.

If you do this, you must recognize that some of the normal callable intrinsics may not be called when DB does not point to the stack. Such intrinsics, if called by a privileged process in split stack mode, can result in system failures. If you are a normal user, you need not concern yourself with this restriction and you may assume in all the intrinsics described in this section that unless it is otherwise stated, an intrinsic will not operate in split stack mode.

The SPECIAL CONSIDERATIONS portion of the description is omitted unless the intrinsic operates in split stack mode, a special optional capability is required, or the intrinsic requires a privileged call. Therefore, *unless otherwise stated*:

The intrinsic does not operate in split stack mode.

The intrinsic requires only standard capabilities.

The intrinsic does not require a privileged call.

- **TEXT DISCUSSION:** This references the page in this manual where usage of the intrinsic is discussed.

# ACTIVATE

INTRINSIC NUMBER 104

Activates a process.

```
IV LV 0-V  
ACTIVATE(pin,susp);
```

After a process has been created, it must be activated in order to run. Once activated, the process runs until it is suspended or deleted. A newly-created process can only be activated by its father. A process that has been suspended (with the SUSPEND intrinsic, see page 2-139) can be reactivated by its father or any of its sons, as specified in the *susp* parameter of the ACTIVATE and SUSPEND intrinsics.

The operating system guarantees that there will be no process switching (to some other process) between activation of the called process and suspension of the calling process.

The ACTIVATE intrinsic aborts the calling process (and possibly the entire job/session) if:

1. The log-on group does not have the Process-Handling Capability and the program was not prepared with Process-Handling Capability.
2. The required parameter *pin* is omitted.
3. A job or session main process, or a system process, is specified.

## PARAMETERS

*pin*

*integer by value (required)*

Process Identification Number (PIN). An integer specifying the PIN for the son or father process to be activated. The PIN number to activate a father process is always zero. The called process *must always* be expecting an activation from the caller as noted in the discussion of the SUSPEND (see page 2-139) and CREATE (see page 2-18) intrinsics.

*susp*

*logical by value (optional)*

A word that specifies:

The calling process is to be suspended while the called process is activated and commences execution.

or

The called process is activated by the operating system *but* does not commence execution immediately. Instead, control is returned to the calling process which will continue execution.

When *susp* is omitted or is zero, the calling process remains active. When *susp* is specified, the calling process is suspended. The 14th and 15th bits of *susp* specify the anticipated source of the call that later will reactivate the calling process.

Bit (15:1) — If *on*, the process expects to be activated by its father.

Bit (14:1) — If *on*, the process expects to be activated by one of its sons.

If both bits are *on*, the suspended process can be activated by either the father or sons.

Bits (0:14) — Reserved for MPE. Should be set to zero.

*Default: Calling process remains active.*

## CONDITION CODES

CCE	Request granted. Called process is activated. The calling process is suspended if <i>susp</i> was specified.
CCG	The called process already is active. The calling process is suspended if <i>susp</i> was specified.
CCL	Request denied, because the called process was not expecting activation by this calling process; an illegal <i>pin</i> parameter was specified; or the <i>susp</i> parameter was specified improperly.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

Process-Handling Capability required.

## TEXT DISCUSSION

Page 8-10.



# ADJUSTUSLF

INTRINSIC NUMBER 83

Adjusts directory space in a USL file.

```
I          IV  IV  
errnum:=ADJUSTUSLF(uslfnm,records);
```

The ADJUSTUSLF intrinsic moves the start of the information block forward or backward on a user subprogram library (USL) file, thereby increasing or decreasing, respectively, the space available for the file directory block. Note that this does not change the overall length of the file. This intrinsic is intended for programmers writing compilers. See the *MPE Segmenter Reference Manual* for a discussion of USL's, the ADJUSTUSLF intrinsic, information blocks, and directory blocks.

## FUNCTIONAL RETURN

This intrinsic returns an error number if an error occurs. If no error occurs, no value is returned.

## PARAMETERS

<i>uslfnm</i>	<i>integer by value (required)</i> A word supplying the file number of the USL file (as returned by FOPEN).
<i>records</i>	<i>integer by value (required)</i> A word supplying a signed record count. If <i>records</i> is greater than zero, the information block is moved toward the end-of-file in the USL file, increasing the space available for the directory block and decreasing the space available for the information block. If <i>records</i> is less than zero, the information block is moved toward the start of the USL file, decreasing the directory-block space and increasing the information-block space.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied. One of the following error numbers is returned.

Error Number	Meaning
0	The file specified by <i>uslfnm</i> was empty, or an unexpected end-of-file was encountered when reading the old <i>uslfnm</i> , or an unexpected end-of-file was encountered when writing on the new <i>uslfnm</i> .

# ADJUSTUSLF

Error Number	Meaning
1	Unexpected input/output error occurred. This can occur on the old <i>uslfnm</i> or the new <i>uslfnm</i> to which the intrinsic is copying the information.
3	Your request attempted to exceed the maximum file directory size (32,768 words).
6	Insufficient space was available in the USL file information block.
7	The intrinsic was unable to open the new USL file.
8	The intrinsic was unable to close (purge) the old USL file.
9	The intrinsic was unable to close (purge) the new USL file.
10	The intrinsic was unable to close \$NEWPASS.
11	The intrinsic was unable to open \$OLDPASS.

## TEXT DISCUSSION

See the *MPE Segmenter Reference Manual*.



# ALTDSEG

INTRINSIC NUMBER 134

Alters the size of an extra data segment.

```
LV I IV  
ALTDSEG(index,inc,size);
```

The ALTDSEG intrinsic alters the current size of an extra data segment. ALTDSEG can be used to reduce the storage required by the segment when it is moved into main memory, then used again to expand storage as required, thus allowing more efficient use of memory. In no case, however, may ALTDSEG be used to increase the segment size beyond that originally assigned through GETDSEG (see page 2-94).

Expansion and contraction is accomplished in even multiples of 4, which are rounded up. For example,

Present Segment Size (Words)	Change Value (Words)	New Segment Size (Words)
128	-3	128
128	-4	124
128	+1	132
128	+3	132
128	+4	132

## PARAMETERS

<i>index</i>	<i>logical by value (required)</i> A word containing the logical index of the extra data segment, obtained from the GETDSEG call.
<i>inc</i>	<i>integer (required)</i> The value, in words, by which the data segment is to be changed. A positive integer value requests an increase, and a negative integer value requests a decrease.
<i>size</i>	<i>integer by value (required)</i> A word to which is returned the new size of the data segment after incrementing or decrementing takes place.

## CONDITION CODES

CCE	Request granted.
CCG	Request not fully granted. An illegal decrement, requesting a new total segment size of zero or less, or an illegal increment, requesting a new size greater than the size originally assigned by GETDSEG, was attempted. In the first case, the current size remains in effect. In the second case, the original maximum size from the GETDSEG intrinsic is granted and this size is returned through the <i>size</i> parameter.
CCL	Request denied because an illegal <i>index</i> parameter was specified.

## **SPECIAL CONSIDERATIONS**

Data-Segment Management Capability required.

## **TEXT DISCUSSION**

Page 8-15.

# ARITRAP

INTRINSIC NUMBER 51

Enables or disables all hardware arithmetic traps.

```
LV  
ARITRAP(state);
```

The interrupts listed below are collectively called the *arithmetic user traps*.

When a user process begins execution, all internal arithmetic user traps are enabled. That is, if an arithmetic error occurs in the user process, it is aborted in the trap mechanism. The various interrupts which can occur are:

- Integer overflow.
- Floating point overflow.
- Floating point underflow.
- Integer divide by zero.
- Floating point divide by zero.
- Privileged instruction.
- Illegal instruction.
- Double precision overflow.
- Double precision underflow.
- Double precision divide by zero.
- Decimal overflow.
- Invalid ASCII digit.
- Invalid decimal digit.
- Invalid source word count.
- Invalid decimal operand length.
- Decimal divide by zero.

The traps may be collectively enabled/disabled with the ARITRAP intrinsic call.

The ARITRAP intrinsic *always* clears the overflow indicator located in the caller's status word.

## PARAMETERS

*state*                      *logical by value (required)*  
A word specifying whether all traps are to be enabled or disabled.  
If *state* is TRUE (bit 15 = 1), all traps are enabled.  
If *state* is FALSE (bit 15 = 0), all traps are disabled.  
Bits 0 through 14 are reserved for MPE and should be set to zero.

## CONDITION CODES

CCE                      Request granted. The arithmetic traps were originally disabled.

CCG                      Request granted. The arithmetic traps were originally enabled.

CCL                      Not returned by this intrinsic.

## TEXT DISCUSSION

Page 4-30.

Converts a one-word binary number to a numeric ASCII string.

INTRINSIC NUMBER 63

```
I      LV IV BA
numchar:=ASCII(word,base,string);
```

Any 16-bit binary number can be converted to a different base and represented as a numeric character ASCII string by using the ASCII intrinsic call.

## FUNCTIONAL RETURN

This intrinsic returns the number of characters in the resulting string.

## PARAMETERS

- word** *logical by value (required)*  
The number to be converted to an ASCII string.
- base** *integer by value (required)*  
An integer indicating octal or decimal conversion.  
8 = octal  
10 = decimal (left justified)  
-10 = decimal (right justified)  
If any other number is entered in this parameter, the intrinsic causes the user process to abort.
- string** *byte array (required)*  
A byte array into which the converted value is placed. This array must be long enough to contain the result. No result, however, exceeds six characters. For octal conversion (base = 8), six characters, including leading zeros, are always returned in *string*, showing the octal representation of *word*. In octal conversions, the length returned by ASCII is the number of significant (right-justified) characters in *string* (excluding leading zeros). If *word* = 0, the length (*numchar*) returned by ASCII is 1.
- For decimal conversions, *word* is considered as a 16-bit, 2's complement integer ranging from -32768 to +32767. If the value of *word* is negative, the first byte of *string* contains a minus sign. If *word* = 0, only one zero character is returned in *string*. The length (*numchar*) returned by ASCII is the total number of characters in *string* (excluding the sign). If *word* = 0, the length returned by ASCII is 1.
- For decimal left-justified conversions (base = 10), leading zeros are removed and the numeric ASCII result is left justified in *string*.
- For decimal right-justified conversions (base = -10), the result is right justified in *string*.

# ASCII

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-10.

Converts a number from an ASCII string to a binary word.

INTRINSIC NUMBER 62

```
L      BA  IV  
bineqv:=BINARY(string,length);
```



## FUNCTIONAL RETURN

This intrinsic returns the binary equivalent of the numeric string.

## PARAMETERS

<i>string</i>	<i>byte array (required)</i> Contains the octal or signed decimal number (ASCII characters) to be converted. If the character string in this array begins with a percent sign (%), it is treated as an octal value. If the string begins with a plus sign, minus sign, or a number, it is treated as a decimal value.
<i>length</i>	<i>integer by value (required)</i> An integer representing the length (number of bytes) in the byte array containing the ASCII-coded value. If the value of <i>length</i> is 0, the intrinsic returns 0 to the calling process. If the value of <i>length</i> is less than 0, the intrinsic causes the user process to abort.

## CONDITION CODES

CCE	Successful conversion. A one-word binary value is returned to the user's process.
CCG	A word overflow, possibly resulting from too many characters ( <i>string</i> number too large), occurred in the word ( <i>bineqv</i> ) returned.
CCL	An illegal character was encountered in the byte array specified by <i>string</i> . For example, the digits 8 or 9 specified in an octal value.

## TEXT DISCUSSION

Page 4-13.



# CALENDAR

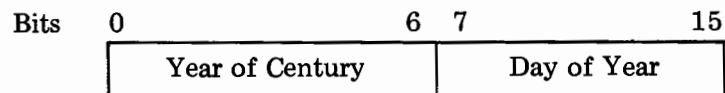
INTRINSIC NUMBER 43

Returns the calendar date.

```
L  
date:=CALENDAR;
```

## FUNCTIONAL RETURN

This intrinsic returns the calendar date in the format



## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-44.

# CAUSEBREAK

Places a session in break mode.

INTRINSIC NUMBER 56

```
CAUSEBREAK;
```

Using the CAUSEBREAK intrinsic is the programmatic equivalent to using the BREAK key in a session. Execution of the process can be resumed where the interruption occurred by entering the command

```
:RESUME
```

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because the intrinsic was not called from an interactive session.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-19.

# CLOCK

INTRINSIC NUMBER 44

Returns the time of day.

```
D  
time:=CLOCK;
```

## FUNCTIONAL RETURN

This intrinsic returns the actual time (wall time), as monitored by the system timer, as a double word. The first word contains the hour of the day and the minute of the hour, the second word contains seconds and tenths of seconds as follows:

Bits 0	7	8	15	
Hour of Day		Minute of Hour		Word 1
Seconds		Tenths of Seconds		Word 2

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-44.

Executes an MPE command programmatically.

INTRINSIC NUMBER 68

```
      BA  I  I  
COMMAND(comimage,error,parm);
```

## PARAMETERS

*comimage*

*byte array (required)*

Contains an ASCII string consisting of a command and parameters terminated by a carriage return. The carriage return character *must* be the last character of the command string. No prompt character, however, should be included in this string. The *comimage* array may be altered by the COMMAND intrinsic (for example, characters in it may be shifted from lowercase to uppercase), but will be returned in a form that can be resubmitted to this intrinsic without adjustment.

*error*

*integer (required)*

A word to which any error code set by the command is returned. This is the same error code that would appear on a job/session list device if the command was part of an input stream.

*parm*

*integer (required)*

A word to which the number (index) of the erroneous parameter is returned. If no parameters are in error, *parm* contains zero. This is the same parameter that would appear on the job/session list device if the command was part of an input stream.

## CONDITION CODES

CCE

The command was executed successfully.

CCG

An executor-dependent error, such as an erroneous parameter, prevented execution of the command. The *error* parameter contains the numeric error code. The *parm* parameter, if not zero, contains the number (index) of the erroneous parameter element. The *error* and *parm* parameters are returned only if the condition code is CCG.

CCL

The command was an undefined command.

## TEXT DISCUSSION

Page 4-9.

# CREATE

INTRINSIC NUMBER 100

Creates a process.

```
          BA      BA I  IV LV      IV  IV      IV      LV
CREATE(progrname,entryname,pin,param,flags,stacksize,dsize,maxdata,priorityclass,
      IV  0-V
      rank);
```

Any running process, if it has the Process-Handling Capability, can request the creation of a son process by issuing the CREATE intrinsic call. The CREATE intrinsic loads the program to be run by the new process into virtual memory, creates the new process as the son of the calling process, initializes its data stack, schedules the process, and returns the new Process Identification Number (PIN) to the requesting process.

The creating process is aborted if:

1. Request was rejected because of illegal parameters; a PIN of zero is returned. Specifically, this occurs:
  - If *progrname* is illegal.
  - If *entryname* is illegal.
  - If *stacksize* is less than 512 (decimal) and is not -1. (Note that if -1 is specified, the default value is taken.)
  - If *dsize* is less than 0 and is not -1.
  - If *maxdata* is less than or equal to 0, and is not -1.
  - If (*dsize* + *globsize* + *stacksize* + 128) exceeds *maxdata*. Note that *dsize* may have been modified to satisfy condition 2 under CCG. The *globsize* value is the sum of the primary DB plus the secondary DB values (the total DB given at program preparation time by the program map (PMAP)).
  - If (*dsize* + *globsize* + *stacksize* + 128) exceeds the maximum *stacksize* defined during system configuration. Note that *dsize* may have been modified to satisfy condition 2 under CCG.
  - If (*maxdata* + 90) exceeds 32768, where *maxdata* is either the value passed as a parameter or a value re-computed by the Loader under condition 1 of CCG.
2. The program file does not have the Process-Handling Optional Capability.
3. An illegal value (a non-existent subqueue) was specified for the *priorityclass* parameter.
4. A required parameter (*progrname* or *pin*) is omitted.
5. A reference parameter was not within the required range.

## PARAMETERS

- progrname* *byte array (required)*  
Contains a string, terminated by a blank, specifying the name, and optionally, the account and group (*filereference* format, see Section III, page 3-8) of the file containing the program to be run.
- entryname* *byte array (optional)*  
Contains a string, terminated by a blank, specifying the entry point (label) in the program where execution is to begin when the process is activated. The *primary* entry point in the program can be specified by setting the array equal to a blank character alone.  
*Default: The primary entry point is used.*
- pin* *integer (required)*  
A word in which the PIN of the new process is returned to the requesting process. This PIN is used in other intrinsics to reference the new process. The PIN can range from 1 to 255. If an error is detected, a PIN of zero is returned to the requesting process.
- param* *integer by value (optional)*  
A word used to transfer control information to the new process. Any instruction in the outer block of code in the new process can access this information in location Q-4.  
*Default: Word is filled with zeros.*
- flags* *logical by value (optional)*  
A word whose bits, if on, specify the loading options:

### NOTE

Bit groups are denoted using the standard SPL notation. Thus bit (15:1) indicates bit 15, bits (10:3) indicates bits 10, 11, and 12.

Bit (15:1) — ACTIVE bit. If *on*, MPE reactivates the calling process (father) when the new process terminates. If *off*, the calling process is not activated at that time.  
*Default: Off.*

Bit (14:1) — LOADMAP bit. If *on*, a listing of the allocated (loaded) program is produced on the job/session list device. This map shows the Code Segment Table (CST) entries used by the new process. If *off*, no map is produced.  
*Default: Off.*

Bit (13:1) — DEBUG bit. If *on*, a call to DEBUG is made at the first executable instruction of the new process. If *off*, the breakpoint is not set. This bit is ignored if the user is non-privileged and the new process requires privileged

# CREATE

mode. It also is ignored if the user does not have read/write access to the program file of the new process.

*Default: Off.*

Bit (12:1) — NOPRIV bit. If *on*, the program is loaded in *non-privileged* mode. If this bit is *off*, the program is loaded in the mode specified when the program file was prepared.

*Default: Off.*

Bits (10:2) — LIBSEARCH bits. These bits denote the order in which libraries are to be searched for the program:

00 — System Library.

01 — Account Public Library, followed by System Library.

10 — Group Library, followed by Account Public Library, followed by System Library.

*Default: 00.*

Bit (9:1) — NOCB bit. If *on*, file system control blocks are established in an extra data segment. If *off*, control blocks may be established in the Process Control Block Extension (PCBX) area.

*Default: Off.*

## NOTE

This bit should be set *on* if you are using a large stack.

Bits (7:2) — Reserved for MPE. Should be set to zero.

Bits (5:2) — STACKDUMP bits. These bits control the enabling/disabling of the mechanism by which the stack is dumped in the event of an abort:

00 — Enables only if enabled at father level.

01 — Enables unconditionally.

10 — Same as 00.

11 — Disables unconditionally for new process.

*Default: 00.*

Bit (4:1) — Reserved for MPE. Should be set to zero.

## NOTE

The following bits (0:4) are used only when the bit pair (5:2) is 01. Otherwise, these bits are ignored.

Bit (3:1) — DL to QI bit. If *on*, the portion of the stack from DL to QI is dumped. If *off*, this portion of the stack is not dumped.

*Default: Off.*

Bit (2:1) — QI to S bit. If *on*, the portion of the stack from QI to S is dumped. If *off*, this portion of the stack is not dumped.  
*Default: Off.*

Bit (1:1) — Q-63 to S bit. If *on*, the portion of the stack from Q-63 to S is dumped. If *off*, this portion of the stack is not dumped.  
*Default: Off.*

Bit (0:1) — ASCII DUMP bit. If *on*, the dump is interpreted in ASCII, in addition to the octal dump. If *off*, ASCII interpreting is not given.  
*Default: Off.*

*Default: Default values as noted are taken.*

*stacksize*                    *integer by value (optional)*  
An integer (Z — Q) denoting the number of words assigned to the local stack area bounded by the initial Q and Z registers.  
*Default: The same as that specified in the program file.*

*dlsiz*e                        *integer by value (optional)*  
An integer (DB — DL) denoting the number of words in the user-managed stack area bounded by the DL and DB registers.  
*Default: The same as that specified in the program file.*

*maxdata*                    *integer by value (optional)*  
The maximum size allowed for the process' stack (Z — DL) area in words. When specified, this value overrides the one established at program-preparation time.  
*Default: If not specified, and not specified in program file either, MPE assumes stack will remain same size.*

*priorityclass*              *logical by value (optional)*  
A string of two ASCII characters describing the priority class in which the new process is scheduled. This may be all, as discussed under *Rescheduling Processes* (see Section VII, page 7-13) for users with Process-Handling Capability, or CS, DS, and ES for users without the Process-Handling Capability.  
*Default: The same as the priority of the calling process.*

*rank*                         *integer by value (optional)*  
This parameter is used only for compatibility with previous versions of the MPE Operating system. It is ignored for all users.

## NOTE

For the *stacksize*, *dlsiz*e, and *maxdata* parameters, a value of -1 indicates that the MPE Segmenter is to assign default values. Specifying -1 is equivalent to omitting the parameter.



# CREATE

## CONDITION CODES

CCE	Request granted. The new process is created.
CCG	Request granted. The <i>maxdata</i> and/or <i>dlsiz</i> e parameters given were illegal, but other values were used, as follows: <ol style="list-style-type: none"><li>1. If the <i>maxdata</i> specified exceeds that maximum Z — DL allowed by the configuration, the configuration maximum value is assigned.</li><li>2. If <math>(dlsiz</math>e + 100) modulo 128 is <i>not</i> zero, then <i>dlsiz</i>e is rounded upward so that <math>(dlsiz</math>e + 100) modulo 128 = 0.</li></ol>
CCL	Request denied because the <i>progname</i> or <i>entryname</i> specified does not exist.

## SPECIAL CONSIDERATIONS

Process-Handling Capability required.

## TEXT DISCUSSION

Page 7-3.

Converts a string of characters from EBCDIC to ASCII or from ASCII to EBCDIC.

INTRINSIC NUMBER 61

```
IV BA BA IV BA 0-V  
CTRANSULATE(code,instring,outstring,stringlength,table);
```

The CTRANSLATE intrinsic converts a string of characters represented in EBCDIC code to its ASCII equivalent, or a string of characters represented in ASCII code to its EBCDIC equivalent. The CTRANSLATE intrinsic also can be used to translate ASCII and EBCDIC strings to a user-defined code.

## PARAMETERS

- code* *integer by value ( required)*  
An integer identifying a specific translation to be used as follows:
- 0 = Use the user-supplied translation table specified in the parameter *table*.
  - 1 = Use the EBCDIC-to-ASCII table. Those EBCDIC characters which have no ASCII equivalent will translate into a byte of zero.
  - 2 = Use the ASCII-to-EBCDIC table. The ASCII parity bit is ignored.
- instring* *byte array (required)*  
The string of characters to be translated.
- outstring* *byte array (optional)*  
A byte array to which is returned the translated character string. If *outstring* is not specified, all translation will occur within *instring*. The parameters *instring* and *outstring* may specify the same array.
- stringlength* *integer by value (required)*  
A positive integer specifying the length (in bytes) of *instring*.
- table* *byte array (required when code = 0)*  
A byte array to be used as the translation table. The contents of *table*, and the order of these contents, define the translation process. The length of *table* may be as large as 256 bytes, but it needs to be only as large as the largest numeric value of any source byte in *instring*. The table is constructed such that each byte in the table corresponds to a byte in the source string to be translated; for example, the fifth byte in the table corresponds to the fifth character (byte) in the string to be translated.

## CONDITION CODES

- CCE Request granted. Translation performed successfully.

# CTRANSLATE

CCG

Not returned by this intrinsic.

CCL

Request denied because an error occurred.

## TEXT DISCUSSION

Page 4-13.

Converts a two-word binary number (double word) to a numeric ASCII string.

INTRINSIC NUMBER 75

```
I      DV IV  BA
numchar:=DASCII(dword,base,string);
```

A 32-bit double-word binary number can be converted to a different base and represented as a numeric character ASCII string by issuing the DASCII intrinsic call.

## FUNCTIONAL RETURN

This intrinsic returns the number of characters in the resulting string.



## PARAMETERS

**dword** *double by value (required)*  
A double-word value indicating the number to be converted to ASCII code.

**base** *integer by value (required)*  
An integer indicating octal or decimal conversion.  
8 = octal  
10 = decimal (left justified)  
If any other number is entered in this parameter, the intrinsic causes the user process to abort.

**string** *byte array (required)*  
The byte array into which the converted value is placed. This array must be long enough to contain the result. No result, however, exceeds 11 characters.

For octal conversion (base = 8), 11 characters, including leading zeros, are always returned in *string*, showing the octal representation of *dword*. The length (*numchar*) returned by DASCII is the number of significant (right justified) characters in *string*, excluding leading zeros. If *dword* = 0, the length returned by DASCII is 1.

For decimal conversions (base = 10), *dword* is considered as a 32-bit, 2's complement integer ranging from -2,147,483,648 to +2,147,483,647. Leading zeros are removed and the numeric DASCII result is left justified in *string*. If the value of *dword* is negative, the first byte of the string returned contains a minus sign. If *dword* = 0, only one zero character is returned to *string*. *String* can contain up to 11 characters, including the sign. If *dword* = 0, the length returned by DASCII is 1.

# **DASCII**

## **CONDITION CODES**

The condition code remains unchanged.

## **TEXT DISCUSSION**

Page 4-10.

Converts a number from an ASCII string to a double-word binary value.   INTRINSIC NUMBER 74

```
      D      BA      IV  
dval:=DBINARY(string,length);
```

The DBINARY intrinsic performs double-integer ASCII to binary conversion.

## FUNCTIONAL RETURN

This intrinsic returns the converted double-word binary value to *dval*.

## PARAMETERS

*string*                                    *byte array (required)*  
Contains the octal or signed decimal number (in ASCII characters) to be converted. If the character string in this array begins with a percent sign (%), it is treated as an octal representation. If the string begins with a plus sign, minus sign, or number, it is treated as a decimal representation.

*length*                                   *integer by value (required)*  
An integer representing the length (number of bytes) in the string containing the ASCII-coded value. If the value of *length* is 0, the intrinsic returns 0 to the calling process. If the value of *length* is less than 0, the intrinsic causes the user process to abort.

## CONDITION CODES

CCE                                       Successful conversion. A double-word binary value is returned to the program.

CCG                                       A word overflow, possibly resulting from too many characters (*string* number too large), occurred in the word returned.

CCL                                       An illegal character was encountered in *string*. For example, the digits 8 or 9 specified in an octal value.

## TEXT DISCUSSION

Page 4-13.

# DLSIZE

INTRINSIC NUMBER 135

Expands or contracts the area between DL and DB in multiples of 128 words.

```
I      IV
dldbsize:=DLSIZE(size);
```

This intrinsic causes the area between DL and DB to be expanded or contracted within the stack segment. All current information within the area is saved on expansion. If contracting, data in the area which is to be contracted is lost. A request for contraction less than the initial DL size of the area causes the initial DL size to be granted and an error condition (CCL) to be returned. If the size requested causes the stack to exceed the maximum size permitted by the stack area ( $Z - DL$ ), only this maximum is granted.

All addressing within the DL to DB area is DB relative *negative* indexing. Therefore, SPL is the only language, at present, which can access this area for you. If you wish to access this area in SPL, please note that the original data is *not* moved relative to DB on expansion or contraction of the area. For example, if a variable is located at DB - 10 before an expansion, it will be at DB - 10 *after* the expansion.

## FUNCTIONAL RETURN

This intrinsic returns the size actually granted. This value is a negative quantity except on error condition CCL when it is possible to have a positive value returned.

## PARAMETERS

*size*                    *integer by value (required)*  
A negative integer. A size of 0 is permitted and resets the DL to DB area to the original value assigned when the process was created (initial DL). The size granted will be an *absolute value* which is rounded up so that the distance between the beginning of the segment to DB is a multiple of 128 words.

## CONDITION CODES

CCE                    Request granted. The value returned is at least as large as the value requested.

CCG                    Requested size exceeded maximum limit allowed. The maximum limit allowable is granted and its size is returned.

CCL                    1. An illegal *size* parameter was specified. The *size* parameter was a *positive* integer or the negative size requested was smaller than the original DL to DB area. The original area size assigned when the stack segment was created is granted and this size is returned as a negative value.

2. The data segment is a FROZEN stack segment which cannot be changed until the system UNFREEZES it. The area remains *unchanged*. The value returned is a *positive* integer size of the area and denotes this special error condition.

## TEXT DISCUSSION

Page 4-22.



# DMOVIN

INTRINSIC NUMBER 132

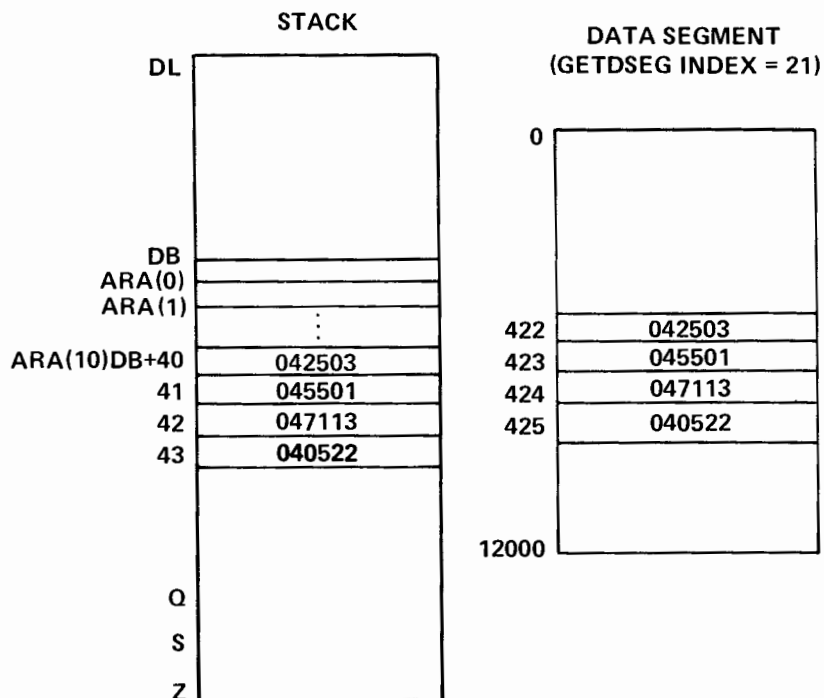
Copies data from data segment to stack.

```
LV IV IV LA  
DMOVIN(index,disp,number,location);
```

A process can copy data from an extra data segment into the stack by issuing the DMOVIN intrinsic call. A bounds check is performed by the intrinsic on both the extra data segment and the stack to insure that the data is taken from within the data segment boundaries and moved to an area within the stack boundaries. For example, in the diagram shown below, if you wish to move 4 words from locations 422 through 425 of the data segment whose index is 21 to DB + 40 through DB + 43 of your stack, the intrinsic call would be

```
DMOVIN(21,422,4,ARA(10));
```

The *index* is 21 (from GETDSEG, see page 2-94); displacement (*disp*) within the data segment is 422; the *number* of words to move into the stack is 4; and the DB relative *location* to begin transferring the data is the address of ARA(10). If ARA(10) is at DB + 40, the end result will be the 4 words moved to DB + 40 through DB + 43 within the stack, as shown below.



## PARAMETERS

<i>index</i>	<i>logical by value (required)</i> A word containing the logical index of the extra data segment, obtained from a GETDSEG intrinsic call.
<i>disp</i>	<i>integer by value (required)</i> The displacement of the first word in the string to be transferred, from the first word in the data segment. This must be an integer value greater than or equal to zero.
<i>number</i>	<i>integer by value (required)</i> The size of the data string to be transferred, in words. This must be an integer value greater than or equal to zero.
<i>location</i>	<i>logical array (required)</i> The array (buffer) in the stack where the data string is to be moved.

## CONDITION CODES

CCE	Request granted.
CCG	Request denied because of bounds-check failure.
CCL	Request denied because of illegal <i>index</i> or <i>number</i> parameter.

## SPECIAL CONSIDERATIONS

Data-Segment Management Capability required.

## TEXT DISCUSSION

Page 8-15.

# DMOVOUT

INTRINSIC NUMBER 133

Copies data from stack to extra data segment.

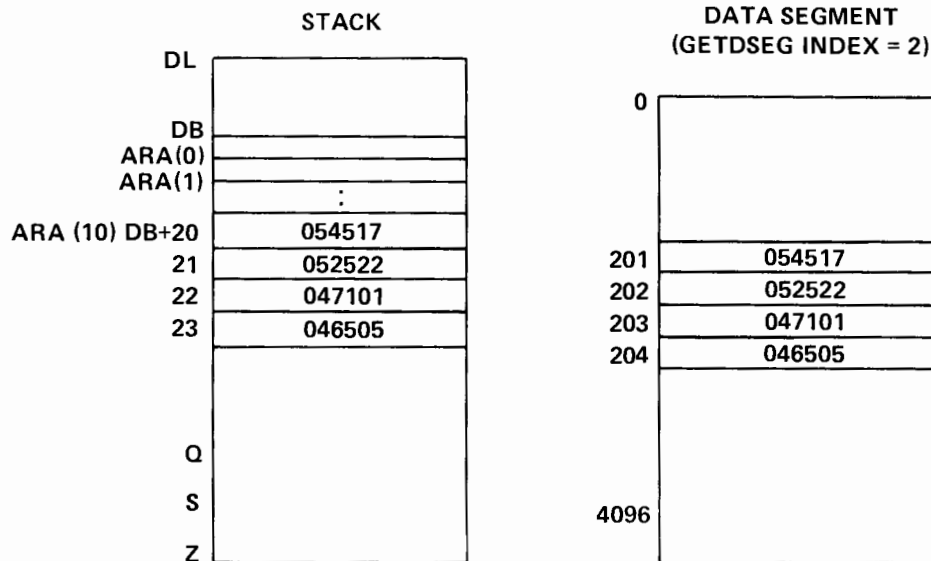
```
LV IV IV LA  
DMOVOUT(index,disp,number,location);
```

The DMOVOUT intrinsic copies data from the stack to an extra data segment. A bounds check is initiated to insure that the data is taken from an area within the stack boundaries and moved to an area with the extra data segment boundaries.

In the example shown below, if you wish to move 4 words from DB + 20 within your stack to the data segment whose index is 2 (from a GETDSEG call, see page 2-94), starting at location 201 within the segment, the intrinsic call could be

```
DMOVOUT(2,201,4,ARA(10));
```

The *index* is 2; the displacement (*disp*) within the data segment is 201; the *number* of words to be moved to the data segment is 4; and the *location* of the data within the stack is the address of ARA(10). If ARA(10) is at DB + 20, the end result is that the 4 words within the stack will be moved to words 201 through 204 of the data segment, as shown below.



## PARAMETERS

<i>index</i>	<i>logical by value (required)</i> A word containing the logical index of the extra data segment, obtained through a GETDSEG call.
<i>disp</i>	<i>integer by value (required)</i> The displacement, in the extra data segment, of the first word of the receiving buffer from the first word in the data segment. This value must be an integer greater than or equal to zero.
<i>number</i>	<i>integer by value (required)</i> The size of the data string to be transferred, in words. This must be an integer value greater than or equal to zero.
<i>location</i>	<i>logical array (required)</i> The array (buffer) in the stack containing the data to be moved.

## CONDITION CODES

CCE	Request granted.
CCG	Request denied because of bounds-check failure.
CCL	Request denied because of illegal <i>index</i> or <i>number</i> parameter.

## SPECIAL CONSIDERATIONS

Data-Segment Management Capability required.

## TEXT DISCUSSION

Page 8-15.

# EXPANDUSLF

INTRINSIC NUMBER 84

Changes length of a USL file.

```
I           IV   IV  
filenum:=EXPANDUSLF(uslfnm,records);
```

You can increase or decrease the length of a USL file by calling the EXPANDUSLF intrinsic.

When this intrinsic is executed, a new USL file is created whose length is *records* longer or shorter than the USL file specified by *uslfnm*. The old USL file is copied to the new file with the same file name, and the old USL file then is deleted.

## FUNCTIONAL RETURN

This intrinsic returns the new file number. If an error occurs, the error number is returned instead of the new file number. The condition code therefore *must* be tested immediately on return from this intrinsic. If an error number were to be used as a file number, unpredictable results would occur.

## PARAMETERS

*uslfnm*

*integer by value (required)*

A word identifier supplying the file number of the file.

*records*

*integer by value (required)*

A signed integer specifying the number of records by which the length of the USL file is to be changed. If *records* is positive, the new USL file is longer than the old USL. If *records* is negative, the new USL file is shorter than the old USL.

## CONDITION CODES

CCE

Request granted. The new file number is returned.

CCG

Not returned by this intrinsic.

CCL

Request denied. One of the following error numbers is returned.

Error Number

Meaning

0

The file specified by *uslfnm* was empty, or an unexpected end-of-file was encountered when reading the old *uslfnm*, or an unexpected end-of-file was encountered when writing on the new *uslfnm*.

1

Unexpected input/output error occurred. This can occur on the old *uslfnm* or the new *uslfnm* to which the intrinsic is copying the information.

# EXPANDUSLF

Error Number	Meaning
3	Your request attempted to exceed the maximum file directory size (32,768 words).
6	Insufficient space was available in the USL file information block.
7	The intrinsic was unable to open the new USL file.
8	The intrinsic was unable to close (purge) the old USL file.
9	The intrinsic was unable to close (purge) the new USL file.
10	The intrinsic was unable to close \$NEWPASS.
11	The intrinsic was unable to open \$OLDPASS.

## TEXT DISCUSSION

See the *MPE Segmenter Reference Manual*.



# FATHER

INTRINSIC NUMBER 109

Requests PIN of father process.

```
I  
pin:=FATHER;
```

A process can determine the Process Identification Number (PIN) of its father by issuing the FATHER intrinsic call.

## FUNCTIONAL RETURN

This intrinsic returns the PIN of the process' father.

## CONDITION CODES

CCE	Request granted. The father is a user process.
CCG	Request granted. The father is a job or session main process.
CCL	Request granted. The father is a system process.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 7-14.

Drives the HP 7260A Optical Mark Reader (OMR)

```

      I      I   IA   I   I
FCARD(recode,filenum,bufadr,count,status);

```

The FCARD intrinsic allows you to control the operation of the 7260A OMR programmatically. This is achieved through passing a parameter (*recode*), corresponding to the function of FCARD desired, from your program to FCARD. FCARD returns to the program parameter values which indicate the success or the cause of failure of execution, the status of the 7260A, the file number of the 7260A/terminal file for which the function has been performed and the number of columns read at the completion of a read request.

## PARAMETERS

*recode*

*integer (required)*

A positive integer represented as an input or output parameter. As an input parameter, *recode* requests one of the following twelve options (functions):

- 0 = Open the reader and the terminal as a file and return to the program the *filenum* through SPL/3000 conventions.
- 1 = Read a single card whether in ASCII or in column image format. See Section V for descriptions of ASCII and column image reading formats.
- 2 = Select the previously read card by routing the card into the select output hopper (providing option 002 of the 7260A is installed).
- 3 = Retransmit data from the previously read card. This transmission may be performed in ASCII or column image reading formats, depending on latest issued FCARD call specifying *recode* equal to 11 or 12.
- 4 = Temporarily suspend the program awaiting an operator action (depress the 7260A "READY" switch). This particular call to FCARD will maintain control and will not be completed until the operator presses the 7260A "READY" switch.
- 10 = Cause the 7260A motor to come to a stop and de-activate MUTE for the associated terminal, if muted. When MUTE is activated and the 7260A is in its "READY" state, data transmission from the computer and from the 7260A to the terminal is disabled.
- 11 = Cause the output format of the subsequent read (*recode*=1) and retransmit (*recode*=3) requests to be performed in the image reading format.

In image mode reading, *count* is returned to the program with the number of columns which have been transmitted.



# FCARD

12 = Cause the output format of the subsequent read (*recode*=1) and retransmit (*recode*=3) requests to be performed in the ASCII reading format.

In ASCII mode reading, *count* is returned to the program with the number of characters (columns) transmitted.

13 = Cause the 7260A optional bell to ring (providing option 004 is installed).

17 = Enable the "echo-on" function of the computer.

18 = Disable the "echo-on" function of the computer.

20 = Close the reader/terminal file opened with *recode*=0. This effectively completes the program.

As an output parameter, *recode* indicates to the program whether a call to FCARD has been properly executed. The indication given by the value of *recode* is as described below:

0 = Indicates that the request, i.e., the call to FCARD, has been successfully performed. For the following conditions, when output *recode*=0, the specified parameters are significant to the program:

- a. If the request was to open a file (*recode*=0), then *filenum* is significant.
- b. If the request was either to read (*recode*=1) or to retransmit (*recode*=3), then *bufadr* (the first byte may contain status information identical to that contained in the parameter *status*), *count*, *filenum* and *status* are significant.
- c. If the request was to select the previously read card (*recode*=2), then *status* is significant.
- d. If the request was to perform a temporary suspension of the program (*recode*=4), then *status* is significant.
- e. For all other requests (*recode*=10,11,12,17,18 and 20), none of the other parameters are significant.

1 = Indicates that *recode* specified in the request was not one of the following legal values: 0,1,2,3,4,10,11,12,13,17,18 or 20.

2 = Indicates that FCARD was unable to open the 7260A/terminal pair as a file. This error is not recoverable, thus the program should indicate an error and terminate itself.

4 = Indicates that FCARD has encountered a file read or write error while accessing the 7260A. This error is not recoverable, thus the program should indicate an error and terminate itself.

5 = Indicates that FCARD was unable to close the 7260A/terminal file. This error is not recoverable, thus the program should indicate an error and proceed to a normal termination.

6 = Indicates that a logical end-of-data (:JOB, :EOJ, :EOD and :DATA) was encountered while reading data in response to either a read or retransmit request.

7 = Indicates that FCARD has encountered a file error on requests for either enabling or disabling the echo function.

8 = Indicates that FCARD has detected a data dropout condition while the 7260A was transmitting. You should request a retransmission of the data or status (see *recode=3*).

***filenum***

*integer (required)*

A word identifier supplying the file number of the file associated with the reader/terminal file. This file number is returned to the program from FCARD with output *recode=0*. It must be provided to FCARD on all requests.

***bufadr***

*integer array (required)*

The array to which the record is to be transferred. This parameter should be set to 120 words.

***count***

*integer (required)*

A positive integer which is returned to the program upon completion of a read (*recode=1*) or a retransmit (*recode=3*) request indicating the number of columns which have been transferred from the 7260A OMR.

***status***

*integer (required)*

An integer indicating whether the OMR has successfully performed the requested task. If *status* is equal to zero, then the request has been successfully performed. If *status* is not equal to zero, then it contains an octal value specifying the OMR condition. The options are:

OCTAL 22     READY status. Indicates that the OMR READY push button has been pressed (*recode=4*). Would also indicate that the OMR is ready but there is no data to be retransmitted (*recode=3*).

OCTAL 07     Input hopper empty or hopper full status. Can either be returned upon a read request (*recode=1*) or upon a retransmit request, if there is no data to retransmit (*recode=3*).

OCTAL 11     Pick fail status. Can either be returned upon a read request (*recode=1*) or upon a retransmit request, if there is no data to retransmit (*recode=3*).

# FCARD

- OCTAL 37    Not ready status. Can either be returned upon a read request (*recode=1*) or upon a retransmit request (*recode=3*). This status is provided by the OMR if the operator has pushed the OMR STOP push button or if a lamp has burned out in the OMR read head.
- OCTAL 14    Select successful status. Indicates that the OMR has successfully selected the card upon the select request (*recode=2*).
- OCTAL 13    Select hopper full status. Indicates that the OMR's select hopper was full when the select request (*recode=2*) was issued.

FCARD derives the parameter *status* by assigning the contents of the first byte of *bufadr* to *status*, if this byte equals one of the values of status given above after a read (*recode=1*), select (*recode=2*) or retransmit (*recode=3*) request, or if this byte equals octal 22 after a request for a temporary suspension of the program (*recode=4*).

For more details on the OMR status, refer to the HP 7260A Operating and Service Manual (HP Part No. 07260-90001).

## SPECIAL CONSIDERATIONS

The condition code remains unchanged.

## TEXT DISCUSSION

Page 5-28.

Requests details about file input/output errors.

INTRINSIC NUMBER 10

```

IV      I  I  D      I  0-V
FCHECK(filename,errorcode,tlog,blknum,numrecs);

```

When a file intrinsic returns a condition code indicating a physical input/output error, additional details may be obtained by using the FCHECK intrinsic call. This intrinsic applies to files on any device.

FCHECK accepts zero as a legal *filename* parameter value. When zero is specified, the information returned in *errorcode* reflects the status of the last call to FOPEN. When an FOPEN fails, there is obviously no file number which can be referenced in *filename*. Therefore, when an FOPEN fails, a *filename* of zero can be used in the FCHECK intrinsic call to obtain the *errorcode* only. If the *tlog*, *blknum*, or *numrecs* parameters are specified, a zero value will be returned to these parameters. If a *filename* of zero is used for a file which has been previously FOPENed, but not yet FCLOSEd, the returned *errorcode* will be meaningless.

## PARAMETERS

<i>filename</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file for which error information is to be returned.
<i>errorcode</i>	<i>integer (optional)</i> A word to which is returned a 16-bit code, specifying the type of error that occurred. If the previous operation was successful, all 16 bits are set to zero. <i>Default: The error code is not returned.</i>
<i>tlog</i>	<i>integer (optional)</i> A word to which is returned the transmission log value recorded when an erroneous data transfer occurs. This word specifies the number of words not read or written (those left over) as the result of the input/output error. <i>Default: The transmission log value is not returned.</i>
<i>blknum</i>	<i>double (optional)</i> A double word to which is returned the relative number of the block involved in the error. <i>Default: The block number is not returned.</i>
<i>numrecs</i>	<i>integer (optional)</i> A word to which is returned the number of logical records in the bad block. <i>Default: The number of logical records is not returned.</i>

# FCHECK

In the 16 bits returned to the word specified by the *errorcode* parameter, the low-order eight bits contain the error-type code that shows what kind of error occurred.

The following codes are returned in *errorcode* by FCHECK:

Code (Decimal)	Meaning
0	End of file.
1	Illegal DB register setting (typically, a request in split-stack mode when it is illegal).
2	Illegal capability
8	Illegal parameter value.
20	Invalid operation.
21	Data parity error.
22	Software time-out.
23	End of tape.
24	Unit not ready.
25	No write ring on tape.
26	Transmission error.
27	Input/output time-out.
28	Timing error or data overrun.
29	Start input/output (SIO) failure.
30	Unit failure.
31	End of line (special character terminator).
32	Software abort of input/output operation.
33	Data lost.
34	Unit not on line.
35	Data set not ready.
36	Invalid disc address.
37	Invalid memory address.
38	Tape parity error.
39	Recovered tape error.
40	Operation inconsistent with access type.
41	Operation inconsistent with record type.
42	Operation inconsistent with device type.
43	The <i>tcount</i> parameter value exceeded the <i>reclsize</i> parameter, but the <i>multirecord access aoption</i> was not specified when the file was opened.
44	The FUPDATE intrinsic was called, but the file was positioned at record zero. (FUPDATE must reference the last record read, but no previous record was read.)
45	Privileged file violation.
46	File space on all discs in the device class specified is insufficient to satisfy this request.
47	Input/output error on a file label.
48	Invalid operation due to multiple file access.
49	Unimplemented function.
50	The account referenced does not exist.
51	The group referenced does not exist.
52	The referenced file does not exist in the system (permanent) file domain.

Code (Decimal)	Meaning
53	The referenced file does not exist in the job temporary file domain.
54	The file reference is invalid.
55	The referenced device is not available.
56	The device specification is invalid or undefined.
57	Virtual memory is not sufficient for the file specified.
58	The file was not passed (typically, a request for \$OLDPASS when there is no \$OLDPASS).
59	Standard label violation.
60	Global RIN not available.
61	Group disc file space exceeded.
62	Account disc file space exceeded.
63	Non-sharable device (ND) capability required but not assigned.
64	Multiple RIN (MR) capability required but not assigned.
66	Plotter limit switch reached.
67	Paper tape error.
68	System internal error.
69	Miscellaneous (ATTACHIO) input/output error.
71	Too many files opened for process.
72	Invalid file number.
73	Bounds check violation.
77	NO-WAIT input/output operation is pending.
78	There is no NO-WAIT input/output for any file.
79	There is no NO-WAIT input/output for file specified.
80	Configured maximum number of spoolfile sectors would be exceeded by this output request.
81	No SPOOL class defined in system.
82	Insufficient space in SPOOL class to honor this input/output request.
83	Extent size exceeds maximum allowable.
84	The next extent in this spoolfile resides on a device which is unavailable to the system (i.e., the device is =DOWN).
85	Operation inconsistent with spooling; e.g., attempt to read hardware status.
86	Spool process internal error.
87	Offset to data is greater than 255 sectors.
89	Power failure.
90	The calling process requested exclusive access to a file to which another process has access.
91	The calling process requested access to a file to which another process has exclusive access.
92	Lockword violation.
93	Security violation.
94	Creator conflict in use of FRENAME intrinsic (user is not the creator).
95	"BROKEN" terminal read.
96	Miscellaneous disc input/output error (device may require HP Customer Engineer attention).
97	CONTROL Y processing requested but no CONTROL Y PIN exists.
98	Input/output read time has overflowed.
99	Magnetic tape error. Beginning of tape (BOT) found while requesting a backspace record (BSR) or a backspace file (BSF).

# FCHECK

Code (Decimal)	Meaning
100	Duplicate file name in the system file directory.
101	Duplicate file name in the job temporary file directory.
102	Directory input/output error.
103	System directory overflow.
104	Job temporary directory overflow.
105	Illegal variable block structure.
106	Extent size exceeds maximum allowable.
107	Offset to data is greater than 255 sectors.
108	Inaccessible file due to a bad file label.
109	Illegal carriage control option.
110	The intrinsic attempted to save a system file in the job temporary file directory.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because <i>filenum</i> was invalid or a bounds violation occurred while processing this request and <i>errorcode</i> is 73.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-65.

Closes a file.

INTRINSIC NUMBER 9

```
IV      IV      IV
FCLOSE(filenum,disposition,seccode);
```

The FCLOSE intrinsic terminates access to a file. This intrinsic applies to files on all devices. FCLOSE deletes the buffers and control blocks through which the user process accessed the file. It also deallocates the device on which the file resides and it may change the disposition of the file. If you do not issue FCLOSE calls for all files opened by your process, such calls are issued automatically by MPE when the process terminates. When a file on magnetic tape is saved, the tape is rewound. All magnetic tape files are left offline after an FCLOSE to indicate to the operator that they may be removed (i.e., the magnetic tape drive has been deallocated).

## PARAMETERS

*filenum*

*integer by value (required)*

A word identifier supplying the file number of the file to be closed.

*disposition*

*integer by value (required)*

Indicates the disposition of the file, significant only for files on disc and magnetic tape. This disposition can be overridden by a corresponding parameter in a :FILE command entered prior to program execution. The disposition options are defined by two-bit fields, as follows:

(13:3) Domain Disposition.

### NOTE

Bit groups are denoted using the standard SPL notation.  
Thus, bits (13:3) indicates bits 13, 14, and 15.

0 = No change. The disposition code remains as it was before the file was opened. Thus, if the file is new, it is deleted by FCLOSE; otherwise, the file is assigned to the domain to which it belonged previously.

1 = Permanent file. If a disc file, it is saved in the system file domain. If the file is a new or old temporary file on disc, an entry is created for it in the system file directory. An error code is returned if a file of the same name already exists in the directory. If the file is an old permanent file on disc, this disposition value has no effect. If the file is stored on magnetic tape, that tape is rewound and unloaded and no directory entries are made.

2 = Temporary job file (rewound). The file is retained in the user's temporary (job/session) file domain and can thus be requested by any process within the job/session. The uniqueness of the file name is checked. If a file of the same name exists already, an error code is returned. If the file resides on magnetic tape, the tape is rewound but not unloaded.



# FCLOSE

3 = Temporary job file (not rewind). This option has the same effect as disposition code 2, except that tape files are *not* rewind.

4 = Released file. The file is deleted from the system.

## NOTE

Although the basic functions covering magnetic tape files are covered above in dispositions 0 through 4, it is recommended that you read the discussion of magnetic tape files in Section III for special considerations not here.

*Default value for this field is code 0 (no change).*

(12:1) Disc Space Disposition.

1 = Returns to the system any disc space allocated beyond the end-of-file indicator.

0 = Does not return any disc space allocated beyond the end-of-file indicator.

*The default value for this field is code 0 (no return).*

When a file is opened by the FOPEN intrinsic, a file count (maintained by the system) is incremented by one. When the file is FCLOSEd, the file count is decremented by one. If more than one FOPEN is in effect for a particular file, its disposition is saved but not affected by the FCLOSE call until the file count is decremented to zero. Then the effective (saved) disposition is the smallest non-zero disposition parameter specified among all FCLOSE calls issued against the file. For example, a file XYZ is opened three successive times by a process. The first FCLOSE disposition is 1, the second FCLOSE disposition is %14, and the third (and last) FCLOSE disposition is %12. The final disposition on the file XYZ will be disposition 1 (permanent file and no return of disc space).

Bits (0:12) are reserved for MPE and should be set to zero.

## *seccode*

*integer by value (required)*

Denotes the type of security initially applied to the file, significant only for new permanent files. The options are:

0 — Unrestricted access — the file can be accessed by any user, unless prohibited by current MPE provisions.

1 — Private file creator security — the file can be accessed only by its creator.

*The default value is 0.*

## CONDITION CODES

CCE	The file was closed successfully.
CCG	Not returned by this intrinsic.
CCL	The file was not closed, perhaps because an incorrect <i>filenum</i> was specified, or because another file with the same name and disposition exists in the system.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-35.



# FCONTROL

INTRINSIC NUMBER 13

Performs control operations on a file or device.

```
IV      IV      L  
FCONTROL(filename,controlcode,param);
```

The FCONTROL intrinsic performs various control operations on a file or on the device on which the file resides.

These operations include:

- Supplying a printer or terminal carriage-control directive.
- Verifying input/output.
- Reading the hardware status word pertaining to the device on which the file resides.
- Setting a terminal's time-out interval.
- Re-winding the file.
- Writing an end-of-file indicator.
- Skipping forward or backward to a tape mark.

The FCONTROL intrinsic applies to files on disc, tape, terminal, or line printer.

## PARAMETERS

*filename*

*integer by value (required)*

A word identifier supplying the file number of the file for which the control operation is to be performed.

*controlcode*

*integer by value (required)*

An integer identifying the operation to be performed:

0 = General Device Control. The *param* parameter is transmitted to the appropriate device driver, and the value returned is transmitted to the user through the *param* parameter.

1 = Line Control. A request to send the value specified in the *param* parameter to the terminal or line printer driver as a carriage-control directive. Use line controls provided by FWRITE when directing to a disc or a spooled file.

# FCONTROL

2 = Complete Input/Output. This insures that requested input/output has been physically completed. Valid only for buffered files. Posts the block being written whether full or not.

3 = Read Hardware Status Word. This operation will return in *param* the status word from the device on which the file resides. The returned value is the status of the device from the previous input/output operation, including FOPEN of the file.

4 = Set Time-Out Interval. This code indicates that a time-out interval is to be applied to input from the terminal. If input is requested from the terminal but is not received in this interval, the FREAD request terminates prematurely with condition code CCL. The interval itself is specified, in seconds, in a word in the user's stack, indicated by *param*. If this interval is *zero*, any previously established interval is cancelled, not being read from the terminal. Note that this only affects the next read.

5 = Rewind File. This repositions the file at its beginning, so that the next record read or written is the first record in the file. This code is valid only for files on disc and magnetic tape.

6 = Write End-Of-File. This operation is used to denote the end of a file on disc or magnetic tape, and is effective only for those devices. If applied to a disc file, the operation writes a logical end-of-data indicator at the point where the file was last accessed. The file label also is updated and written to disc. If the file is an unlabeled magnetic tape file, a tape mark is written at the current position of the tape.

7 = Space Forward to Tape Mark. This moves the tape forward until a tape mark is encountered.

8 = Space Backward to Tape Mark. This moves the tape backward until a tape mark is encountered.

9 = Rewind and Unload Tape File. This repositions the tape file at its beginning and places the tape offline.

## NOTE

Control codes 0, 1, and 3 will be rejected for spooled devicefiles. Control codes 5 through 9 (magnetic tape control) will be rejected for spooled :DATA tapes.

Although the basic functions covering magnetic tape files are covered above, it is recommended that you read the discussion of magnetic tape files in Section III for special considerations not covered here.

# FCONTROL

The following values for *controlcode* are used for changing terminal characteristics. See Section V.

- 10 = Change terminal input speed.
- 11 = Change terminal output speed.
- 12 = Turn echo facility on.
- 13 = Turn echo facility off.
- 14 = Disable the system break function.
- 15 = Enable the system break function.
- 16 = Disable the subsystem break function.
- 17 = Enable the subsystem break function.
- 18 = Disable tape mode option.
- 19 = Enable tape mode option.
- 20 = Disable the terminal input timer.
- 21 = Enable the terminal input timer.
- 22 = Read the terminal input timer.
- 23 = Disable parity checking.
- 24 = Enable parity checking.
- 25 = Define line-termination characters for terminal input.
- 26 = Disable binary transfers.
- 27 = Enable binary transfers.
- 28 = Disable user block mode transfers.
- 29 = Enable user block mode transfers.
- 34 = Disable line deletion echo suppression.
- 35 = Enable line deletion echo suppression.
- 36 = Set parity.
- 37 = Allocate a terminal.
- 38 = Set terminal type.
- 39 = Obtain terminal type information.
- 40 = Obtain terminal output speed.
- 41 = Set unedited terminal mode.
- 43 = Aborts pending NO-WAIT I/O request.

## *param*

### *logical (required)*

If *controlcode* is 1, *param* denotes a word containing the value to be transmitted to the terminal or line printer driver as a carriage control or mode control directive. The carriage control directive is selected from figure 2-3, following FWRITE.

The mode control determines whether any carriage control directive transmitted through the FWRITE intrinsic takes effect before printing (pre-space movement) or after printing (post-space movement). The mode control directive is selected from the octal codes %400 or %401 in figure 2-3.

# FCONTROL

If *param* contains a mode control directive, then a value is returned to *param* that shows the mode setting of the device as it was before the call to FCONTROL, as follows:

Value	Meaning
0	Post-spacing
1	Pre-spacing

If *controlcode* is 4, *param* denotes a word in the user's stack that contains the time-out interval, in seconds, to be applied to input from the terminal.

If *controlcode* is 2, 5, 6, 7, 8, or 9, *param* is any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of the intrinsic. It serves no other purpose, however, and is not modified by the intrinsic.

See Section V for *param* requirements when *controlcode* is 10 or greater.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an error occurred.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Pages 3-76 and 5-1.

# FGETINFO

INTRINSIC NUMBER 11

Requests access and status information about a file.

```
      IV   BA   L   L   I   I   L   L
FGETINFO(filenum, filename, foptions, aoptions, reysize, devtype, ldnum, hdaddr,
      I   D   D   D   D   D   I   L
filecode, recpt, eof, flimit, logcount, physcount, blksize, extsize,
      I   I   BA   D   0-V
numextents, userlabels, creatorid, labaddr);
```

Once a file is opened on any device, the FGETINFO intrinsic can be used to request access and status information about that file.

## PARAMETERS

- filenum*                    *integer by value (required)*  
A word identifier supplying the file number of the file about which information is requested.
- filename*                   *byte array (optional)*  
A byte array to which is returned the actual designator of the file being referenced, in this format:  
f.g.a  
where  
f = the local file name.  
g = the group name (supplied or implicit).  
a = the account name (supplied or implicit).  
The byte array must be 28 bytes long. When the actual designator is returned, unused bytes in the array are filled with blanks on the right. A nameless file will return an empty string.  
*Default: The actual designator is not returned.*
- foptions*                   *logical (optional)*  
The *foptions* parameter returns six different file characteristics by setting corresponding bit groupings in a 16-bit word. Correspondence is from right to left. The file characteristics returned are as follows. The bit settings are summarized in figure 2-1.

### NOTE

Bit groups are denoted using the standard SPL notation. Thus bits (14:2) indicates bits 14 and 15; bits (10:3) indicates bits 10, 11, and 12.

Bits (14:2) — Domain *Foption*.

The file domain that was searched by MPE to locate the file, indicated by these bit settings:

00 - The file is a new file.

BITS	(0:5)	(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	(RESERVED)	DISALLOW :FILE	(RESERVED)	CARRIAGE CONTROL	RECORD FORMAT	DEFAULT DESIGNATOR	ASCII/ BINARY	DOMAIN
MEANING		1 = No :FILE 0 = :FILE		0 = NOCCTL 1 = CCTL	00 = Fixed 01 = Variable 10 = Undefined	000 = filename 001 = \$STDLIST 010 = \$NEWPASS 011 = \$OLDPASS 100 = \$STDIN 101 = \$STDINX 110 = \$NULL	0 = Binary 1 = ASCII	00 = New file 01 = Old System File 10 = Temporary File 11 = Old User File

Figure 2-1. Foptions Bit Summary



01 = The file is an old permanent file.  
10 = The file is an old temporary file.  
11 = The file is an old file.

Bit (13:1) — ASCII/Binary *Foption*.  
For ASCII this bit is 1. For binary, it is 0.

Bits (10:3) — Default File Designator *Foption*.

The bit settings are:

000 = The actual file designator is the same as the formal file designator.

001 = The actual file designator is \$STDLIST.

010 = The actual file designator is \$NEWPASS.

011 = The actual file designator is \$OLDPASS.

100 = The actual file designator is \$STDIN.

101 = The actual file designator is \$STDINX.

110 = The actual file designator is \$NULL.

Bits (8:2) — Record Format *Foption*.

The format in which the records in the file are recorded, indicated by these bit settings:

00 = Fixed-length records.

01 = Variable-length records.

10 = Undefined-length records.

Bit (7:1) — Carriage Control *Foption*.

0 = No carriage-control character expected.

1 = Carriage-control character expected.

Bit (6:1) — Reserved for MPE.

Bit (5:1) — Disallow File Equation *Foption*.

This option ignores any corresponding :FILE command, so that the specifications in the FOPEN call take effect (unless overridden by those in the file label). For disallowing :FILE, this bit is set to 1; for allowing :FILE, the bit is 0.

*Default: Foptions are not returned.*

*aoptions*

*logical (optional)*

The *aoptions* parameter returns up to seven different access options represented by bit groupings in a 16-bit word, as described below. The bit settings are summarized in figure 2-2.

Bits (12:4) — Access Type *Aoptions*.

The type of access allowed users of this file, as follows:

0000 = Read access only.

0001 = Write access only.

0010 = Write access only, but previous data in the file is not deleted.

BITS	(0:3)	(4:1)	(5:1)	(6:1)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)
FIELD	(RESERVED)	NO WAIT I/O	(RESERVED)	MULTI ACCESS	INHIBIT BUFFERING	EXCLUSIVE ACCESS	DYNAMIC LOCKING	MULTI-RECORD ACCESS	ACCESS TYPE
MEANING		1 = No-Wait 0 = Non No-Wait		1 = Multi access 0 = Non-Multi access	1 = NOBUF 0 = BUF	01 = Exclusive 10 = Semi-exclusive 11 = Share 00 = Default	0 = No Dynamic Lock 1 = Dynamic Lock	1 = Multi-record 0 = No multi-record	0000 = Read only 0001 = Write only 0010 = Write (save) only 0011 = Append only 0100 = Read/write 0101 = Update 0110 = Execute

Figure 2-2. Aoptions Bit Summary

# FGETINFO

0011 = Append access only.  
0100 = Input/output access.  
0101 = Update access.  
0110 = Execute access.

Bit (11:1) — *Multirecord Aoption.*

For multirecord mode, this bit is set to 1; for non-multirecord mode, it is 0.

Bit (10:1) — *Dynamic Locking Aoption.*

The bit settings are:

1 = Allow dynamic locking/unlocking.  
0 = Disallow dynamic locking/unlocking.

Bits (8:2) — *Exclusive Aoption.*

This *aoption* specifies whether a user has continuous exclusive access to this file, from the time it is opened to the time it is closed. The bit settings are:

01 = Exclusive access.  
10 = Semi-exclusive access.  
11 = Share access.

Bit (7:1) — *Inhibit Buffering Aoption.*

This option inhibits automatic buffering by MPE and allows input/output to take place directly between the user's stack or extra data segment and the applicable hardware device.

1 = Inhibit buffering.  
0 = Normal buffering

Bit (6:1) — *Multi-Access Mode Aoption.*

This field provides the accessor with a means of sharing access to the file.

1 = Multi access.  
0 = Non-multi access.

Bit (5:1) — Reserved for MPE.

Bit (4:1) — *No-Wait I/O Aoption.*

This bit allows the accessor to initiate an I/O request and to have control returned before the completion of the I/O.

1 = No-wait I/O in effect.  
0 = No-wait I/O not in effect.

Bits (0:3) — Reserved for MPE.

*Default: Aoptions are not returned.*

- recsize*                    *integer (optional)*  
A word to which is returned the logical record size associated with the file. If the file was created as a binary type, this value is positive and expresses the size in words. If the file was created as an ASCII type, this value is negative and expresses the size in bytes.  
*Default: The logical record size is not returned.*
- devtype*                    *integer (optional)*  
A word to which is returned the type and subtype of device being used for the file, where  
bits (0:8) = device subtype, and  
bits (8:8) = device type.  
If the file is not spooled, which can be determined from *hdaddr* (0:8), the returned *devtype* is actual. The same is true if the file is spooled and was opened via logical device number. However, if an output file is spooled and was opened by device class name, *devtype* contains the type and subtype of the first device in its class, which may be different from the device actually used.  
*Default: The device type and subtype are not returned.*
- ldnum*                      *logical (optional)*  
A word to which is returned the logical device number associated with the device on which the file resides.  
  
If the file is a disc file, then the logical device number will be that of the first extent. If the file is spooled, then *ldnum* will be a virtual device number which does not correspond to the system configuration I/O device list.  
*Default: The logical device number is not returned.*
- hdaddr*                      *logical (optional)*  
A word to which the hardware address of the device is returned, where  
bits (0:8) = the Device Reference Table (DRT) number, and  
bits (8:8) = the unit number.  
If the device is spooled, the DRT number will be zero and the unit number is undefined.  
*Default: The hardware address is not returned.*
- filecode*                    *integer (optional)*  
A word to which is returned the value recorded with the file as its file code (for disc files only).  
*Default: The file code is not returned.*
- recpt*                        *double (optional)*  
A double word to which is returned a double integer representing the current logical record pointer setting. This is the displacement in logical records from record number 0 in the file. It identifies the record that would next be accessed by an FREAD or FWRITE call.  
*Default: The logical record pointer setting is not returned.*

<i>eof</i>	<i>double (optional)</i> A double word to which is returned a double positive integer equal to the number of logical records currently in the file. If the file does not reside on disc, this value will be zero. <i>Default: The number of logical records in the file is not returned.</i>
<i>flimit</i>	<i>double (optional)</i> A double word to which is returned a double positive integer representing the number of the last logical record that could ever exist in the file, because of the physical limits of the file. If the file does not reside on disc, this value will be zero. <i>Default: The file limit information is not returned.</i>
<i>logcount</i>	<i>double (optional)</i> A double word to which is returned a double positive integer representing the total number of logical records passed to and from the user during the current access of the file. <i>Default: The logical record count is not returned.</i>
<i>physcount</i>	<i>double (optional)</i> A double word to which is returned a double positive integer representing the total number of physical input/output operations performed within this process against the file since the last FOPEN call. <i>Default: The number of I/O operations is not returned.</i>
<i>blksize</i>	<i>integer (optional)</i> A word to which is returned the block size associated with the file. If the file was created as a binary type, this value is positive and expresses the size in words. If the file was created as an ASCII type, this value is negative and shows the size in bytes. <i>Default: The block size is not returned.</i>
<i>extsize</i>	<i>logical (optional)</i> A word to which is returned the disc extent size associated with the file (in sectors). <i>Default: The disc extent size is not returned.</i>
<i>numextent</i>	<i>integer (optional)</i> A word to which is returned the maximum number of disc extents allowable for the file. <i>Default: The maximum allowable number of extents is not returned.</i>
<i>userlabels</i>	<i>integer (optional)</i> A word to which is returned the number of user header labels defined for the file when it was created. If the file is not a disc file, this number is zero. When an old file is opened for overwrite output, the value of <i>userlabels</i> is not reset and old user labels are not destroyed. <i>Default: The number of user labels is not returned.</i>

<i>creatorid</i>	<i>byte array (optional)</i> A type array to which is returned the eight-byte name of the user who created the file. If the file is not a disc file, blanks are returned. <i>Default: The user name is not returned.</i>
<i>labaddr</i>	<i>double (optional)</i> A double word to which is returned the sector address of the label of the file. If the file is not a disc file, this value is zero. The high-order eight bits show the logical device number. The remaining 24 bits show the absolute disc address. <i>Default: The label address is not returned.</i>

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an error occurred.

## TEXT DISCUSSION

Page 3-63.

# FLOCK

INTRINSIC NUMBER 15

Dynamically locks a file.

```
IV    LV  
FLOCK(filenum,lockcond);
```

The FLOCK intrinsic dynamically locks a file.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file to be locked.
<i>lockcond</i>	<i>logical by value (required)</i> A word specifying conditional or unconditional locking:  TRUE — Locking will take place unconditionally. If the file cannot be locked immediately, the calling process suspends until the file can be locked. Bit 15 = 1  FALSE — Locking will take place only if the file's RIN is not currently locked. If the RIN is locked, control returns immediately to the calling process, with condition code CCG. Bit 15 = 0.

## CONDITION CODES

The condition codes possible when *lockcond* = TRUE are

CCE	Request granted.
CCG	Not returned when <i>lockcond</i> = TRUE.
CCL	Request denied because this file was not opened with the <i>dynamic locking aoption</i> specified in the FOPEN intrinsic, or the request was to lock more than one file and the calling process does not possess the Multiple RIN Capability.

The condition codes possible if *lockcond* = FALSE are

CCE	Request granted.
CCG	Request denied because the file was locked by another process.
CCL	Request denied because this file was not opened with the <i>dynamic locking aoption</i> specified in the FOPEN intrinsic, or the request was to lock more than one file and the calling process does not possess the Multiple RIN Capability.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

Standard Capability sufficient if only one file is to be locked dynamically.

If more than one file is to be locked dynamically, the Multiple RIN Capability is required.

## TEXT DISCUSSION

Page 3-52.





# FOPEN

INTRINSIC NUMBER 1

Opens a file.

```
      I          BA   LV   LV   IV   BA   BA
filenum:=FOPEN(formaldesignator,foptions,aoptions,resize,device,formmsg,
              IV     IV     IV  DV     IV
userlabels,blockfactor,numbuffers,filesize,numextents,
              IV   IV  0-V
initialloc,filecode);
```

The FOPEN intrinsic makes it possible to access a file. In the FOPEN intrinsic call, a particular file may be referenced by its *formal file designator*, described in Section III. When the FOPEN intrinsic is executed, it returns to the user's process a *file number* by which the system uniquely identifies the file. This file number, rather than the file designator, then is used by subsequent intrinsics in referencing the file.

## FUNCTIONAL RETURN

This intrinsic returns an integer file number used to identify the opened file in other intrinsic calls.

## PARAMETERS

*formaldesignator* . *byte array (optional)*  
Contains a string of ASCII characters interpreted as a formal file designator, as defined in Section III. This string must begin with a letter, contain alphanumeric characters, slashes, or periods, and terminate with any non-alphanumeric character except a slash or a period. If the string names a system-defined file, it can begin with a dollar sign (\$); if it names a user-predefined file, it can begin with an asterisk (\*).  
*Default: A temporary nameless file that can be read from or written to, but not saved, is assigned.*

*foptions* *logical by value (optional)*  
The *foptions* parameter allows you to specify six different file characteristics, by setting corresponding bit groupings in a 16-bit word. The correspondence is from right to left, beginning with bit 15. These characteristics are as follows, proceeding from the rightmost bit groups to the leftmost bit groups in the word. The bit settings are summarized in figure 2-1.

### NOTE

Bit groups are denoted using the standard SPL notation. Thus bits (14:2) indicates bits 14 and 15; bits (10:3) indicates bits 10, 11, and 12.

## Bits (14:2) — Domain *Foption*.

The file domain to be searched by MPE to locate the file, indicated by these bit settings:

00 = The file is a *new* file, created at this point. No search is necessary.

01 = The file is an old permanent file, and the system file domain should be searched.

10 = The file is an old temporary file, and the job file domain should be searched.

11 = The file is an old file that is to be located by first searching the job file domain and then, if the file is not found, by searching the system file domain.

## Bit (13:1) — ASCII/Binary *Foption*.

The code (ASCII or binary) in which a *new* file is to be recorded when it is written to a device that supports both codes. In the case of disc files, this also affects padding that can occur when a direct-write intrinsic call (FWRITEDIR) is issued to a record that lies beyond the current logical end-of-file indicator. In ASCII files, any dummy records between the previous end-of-file and the newly-written record are padded with blanks. In binary files, such records are padded with binary zeros. All files not on disc are treated as ASCII files.

For ASCII files, this bit is 1.

For binary files, this bit is 0.

## Bits (10:3) — Default File Designator *Foption*.

The *actual* file designator is equated with the formal file designator specified in FOPEN, if

1. No explicit or implicit :FILE command equating the formal file designator to a different actual file designator occurs in the job or session; *or*
2. The Disallow File Equation *Foption* (bit 5) is specified.

The bit settings are

000 = The actual file designator is the same as the formal file designator.

001 = The actual file designator is \$STDLIST.

010 = The actual file designator is \$NEWPASS.

011 = The actual file designator is \$OLDPASS.

100 = The actual file designator is \$STDIN.

101 = The actual file designator is \$STDINX.

110 = The actual file designator is \$NULL.

## Bits (8:2) — Record Format *Foption*.

The format in which the records in the file are recorded, indicated by these bit settings:

00 = Fixed-length records. The file is composed of logical records of uniform length.

01 = Variable-length records. The file contains logical records of varying length. This format is restricted to records that are written in sequential order. The size of each record is recorded internally. Specifically, undefined-length records are supported by all devices; fixed- and variable-length records are supported by disc and

# FOPEN

magnetic tape devices only. To state this another way: disc and magnetic tape devices support *all* record formats, whereas all other devices support *only* undefined-length records. The actual record size used is determined by multiplying the *recsize* (specified or default) by the *blockfactor*, and adding two words reserved for system use. This option is not allowed when NOBUF is specified. In such a case, the record format used is undefined-length records, discussed below.

10 = Undefined-length records. The file contains records of varying length that were not written using the variable-length *foption* (01). All files not on disc or magnetic tape are treated as containing undefined-length records by default.

Bit (7:1) — Carriage Control *Foption*.

If selected, this specifies that you will supply a carriage control directive in the calling sequence of each FWRITE call that writes records onto the file.

0 = No carriage control directive expected.

1 = Carriage control directive expected.

Carriage control is defined only for character oriented, i.e., ASCII, files. This option and binary are mutually exclusive and attempts to open new files with both binary and this option results in an access violation.

This option is a physical attribute of the file and its state cannot be modified when opening an old disc file.

A carriage control character passed through the *control* parameter of FWRITE is recognized and acted upon only for files for which carriage control is specified in FOPEN. Embedded control is treated strictly as data on files for which no carriage control is specified, and does not invoke spacing for such files. You may specify spacing action on files for which carriage control has been specified, either by embedding the control in the record, indicated with a *control* parameter of one in the call to FWRITE, or by sending the control code directly via the *control* parameter of FWRITE.

A carriage control character sent to a file on which the control cannot be executed directly, for example, line spacing to a disc or tape file, will result in having the control character embedded as the first byte of the record. Thus, the first byte of each record in a disc file having a carriage control character contains control information. Control sent to other types of files results in transmission of the control to the driver.

The control codes %400 through %403 are remapped to %100 through %103 so that they fit into one byte and thus can be embedded. Records written to the line printer with one of the above controls should not contain information other than control information.

For the purpose of computing record size, carriage control information is considered by the file system to be part of the data record. As such, specifying the carriage control option adds one byte to the record size

at the time the file is created. For example, a specification of REC = -132, 1,F,ASCII;CCTL results in a *recsize* of 133 characters.

You always may read up to and including the *recsize* as returned by FGETINFO. On writes of files for which carriage control is specified, however, the data transferred is limited to *recsize* - 1 unless a control of one is passed indicating the data record is prefixed with embedded control.

Bit (6:1) — Reserved for MPE. Should be set to zero.

Bit (5:1) Disallow File Equation *Foption*.

This option ignores any corresponding :FILE command, so that the specifications in the FOPEN call take effect (unless preempted by those in the file label, for disc files).

0 = Allow :FILE.

1 = Disallow :FILE.

Bits (0:5) — Reserved for MPE. Should be set to zero.

*Default: All bits are set to zero.*

## *aoptions*

*logical by value (optional)*

The *aoptions* parameter permits you to specify up to seven different access options established by bit groupings in a 16-bit word. These access options are described below. The bit settings are summarized in figure 2-2.

Bits (12:4) — Access Type *Aoptions*.

The type of access allowed for this access of this file:

0000 = Read access only. The FWRITE, FUPDATE, and FWRITEDIR intrinsic calls cannot reference this file.

0001 = Write access only. Any data written in the file prior to the current FOPEN request is deleted. The FREAD, FREADSEEK, FUPDATE and FREADDIR intrinsic calls cannot reference this file.

0010 = Write access only, but previous data in the file is *not* deleted. The FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file.

0011 = Append access only. The FREAD, FREADDIR, FREADSEEK, FUPDATE, FSPACE, FPOINT, and FWRITEDIR intrinsic calls cannot reference this file. This option is not valid for files containing variable-length records.

0100 = Input/output access. Any file intrinsic except FUPDATE can be issued for this file.

0101 = Update access. All file intrinsics, including FUPDATE, can be issued for this file.

0110 = Execute access. Allows users with Privileged Mode Capability input/output access to any loaded file.

## Bit (11:1) — Multirecord *Aoption*.

Signifies that individual read or write requests are not confined to record boundaries. Thus, if the number of words or bytes to be transferred (specified in the *tcount* parameter of the read or write request) exceeds the size of the physical record (i.e., block) referenced, the remaining words or bytes are taken from subsequent successive records until the number specified by *tcount* have been transferred. This option is available only if the inhibit buffering *aoption*, described below, is selected also.

0 = Non-multirecord mode.

1 = Multirecord mode.

## Bit (10:1) — Dynamic Locking *Aoption*.

Indicates that you want to use the FLOCK and FUNLOCK intrinsics to dynamically permit or restrict concurrent access to the file by other processes at certain times. The user process can continue this temporary locking/unlocking until it closes the file. Dynamic locking/unlocking is made possible through a Resource Identification Number (RIN) assigned to the file and temporarily acquired by the FOPEN intrinsic. The calling process and other processes must use the RIN in cooperation to guarantee the integrity of the file, as discussed in Section III. Non-cooperating processes are allowed concurrent access at all times, unless other provisions prohibit this.

0 = Disallow dynamic locking/unlocking.

1 = Allow dynamic locking/unlocking. A file may be multiple accessed only if all FOPEN requests for the file specify dynamic locking, or if none of them do. An FOPEN request that disagrees with the current access, if any, will fail.

## Bits (8:2) — Exclusive *Aoption*.

This *aoption* specifies whether you have continuous exclusive access to this file, from the time it is opened to the time it is closed. This option often is used when performing some critical operation, such as updating the file.

01 = Exclusive access. After this file is opened, prohibits another FOPEN request, whether issued by this or another process, until this process issued the FCLOSE request or terminates. If any process already is accessing this file when this FOPEN call is issued, a CCL error code is returned to the calling process. If another FOPEN call is issued for this file while the exclusive *aoption* is in effect, an error code is returned to that calling process. The exclusive access *aoption* can be requested only by users allowed the file locking access mode by the security provisions for the file.

10 = Semi-exclusive access. After the file is opened, prohibits concurrent output access to this file through another FOPEN request, whether issued by this or another process, until this process issues the FCLOSE request or terminates. A subsequent request for the input/output or update *aoption* access type will obtain read only access. Other types of read access, however, are allowed. If any process already has output access to the file when this FOPEN call

is issued, a CCL error code is returned to the calling process. If another FOPEN call that violates the read-only restriction is issued while the semi-exclusive *aoption* is in effect, that call fails and an error code is returned to the calling process. The semi-exclusive access can be requested only by users allowed the file-locking access mode by the security provisions for the file.

- 11 = Share access. After the file is opened, permits concurrent access to this file by any process, in any access mode, subject to other basic MPE security provisions in effect.
- 00 = Default value. If the read access only *aoption* is selected, share access (11) takes effect. Otherwise, exclusive access (01) takes effect.

#### Bit (7:1) — Inhibit Buffering *Aoption*.

When selected, this *aoption* inhibits automatic buffering by MPE and allows input/output to take place directly between the user's data area and the applicable hardware device.

0 = Allow normal buffering.

1 = Inhibit buffering (NOBUF).

NOBUF access is oriented to the transfer of physical blocks rather than logical records.

With NOBUF access, you have responsibility for blocking and de-blocking of records in the file (see Section III). To be consistent with files built using buffered I/O, records should begin on word boundaries, and when the information content of the record is less than the defined record length, the record should be padded with blanks by you if the file is ASCII or with zeros if the file is binary.

The *resize* and block size for files manipulated under NOBUF access follow the same rules as those files that are created using buffering. The default *blockfactor* for a file created under NOBUF is one.

When a NOBUF file is opened without multirecord access, the amount of data transferred per read or write is limited to a maximum of one block.

The end-of-file, next record pointer, and record transfer count are maintained in terms of logical records for all files. The number of logical records affected by each transfer is determined from the size of the transfer.

Transfers always begin on a block boundary. Those transfers which do not transfer whole blocks leave the next record pointer set to the first record in the next block. The end-of-file pointer always points at the last record in the file.

For files opened with NOBUF access, the FREADDIR, FWRITEDIR, and FPOINT intrinsics treat the *recnum* parameter as a block number.

Bit (6:1) — Multi-Access Mode *Aoption*.

When selected, this *aoption* provides the accessor with a means of sharing access to the file, including file system buffers. Thus, the accessor can “sequentially” access records within the file in conjunction with other such accessors in the same job. This option is not allowed if the file is accessed exclusively, if NOBUF is selected, or if the multirecord option is requested.

Bit (5:1) — Reserved for MPE. Should be set to zero.

Bit (4:1) — No-Wait I/O *Aoption*.

The selection of this *aoption* allows you to initiate an I/O request and to have control returned before the completion of the I/O. The IOWAIT intrinsic must be called after each I/O request to confirm the completion of the I/O. Also, multirecord access is not available.

## NOTE

You must be running in Privileged Mode to select No-Wait I/O and NOBUF.

Bits (0:3) — Reserved for MPE. Should be set to zero.

*Default: All bits are set to zero.*

*resize*

*integer by value (optional)*

An integer indicating the size of the logical records in the file. If a positive number, this represents words; bytes are represented by a negative number. If the file is a newly-created file, this value is recorded permanently in the file label. If the records in the file are of variable length, this value indicates the maximum logical record length allowed.

Binary files are word oriented. A record size specifying an odd byte count for a binary file is rounded up by FOPEN to the next highest even number.

ASCII files may be created with logical records which are an odd number of bytes in length. Within each block, however, records begin on word boundaries.

For either ASCII or binary files with fixed or undefined length records, the record size is rounded up to the nearest word boundary. For example, a *resize* specified as -106 for an ASCII file is 106 characters (53 words) in length. A *resize* of -113 for a binary file is 114 characters (57 words) in length. The rounded sizes should be used in computations for *blockfactor* or block size.

*Default: The default value is the configured physical record width of the associated device.*

*device*

*byte array (optional)*

Contains a string of ASCII characters terminating with any non-alphanumeric character except a slash or period, designating the device

on which the file is to reside. The string may represent a device class name up to eight alphanumeric characters beginning with a letter or a logical device number consisting of a three-byte numeric string. Device class names and logical device numbers are defined and assigned to devices during system configuration. See the MPE General Information Manual for a discussion of device class names and logical device numbers.

If the file is a newly-created disc file and the device specification is a device class, then all extents of the file are restricted to members of the class. Similarly, if the device specification is a logical device number, then all extents are restricted to the specified logical device.

*Default: Disc.*

*formmsg*

*byte array (optional)*

Contains a forms message that can be used for such purposes as telling the console operator what type of paper to use in the line printer. This message must be displayed to the operator and verified before this file can be printed on a line printer. The message itself is a string of ASCII characters terminated by a period. The maximum number of characters allowed in the array is 49, including any terminating period. Arrays with more than 49 characters are truncated by MPE.

*Default: No forms message is available.*

*userlabels*

*integer by value (optional)*

An integer specifying the number of user-label records to be written for this file.

*Default: The default number of user-label records is zero.*

*blockfactor*

*integer by value (optional)*

An integer containing the size of each buffer to be established for the file, specified as a number equal to the number of logical records per block. For fixed-length records, *blockfactor* is the actual number of records in a block. For variable-length records, *blockfactor* is interpreted as a multiplier used to compute the block size (maximum *resize* x *blockfactor*). For undefined-length records, *blockfactor* is always one logical record per block. The *blockfactor* value specified by you may be overridden by MPE. The valid range for *blockfactor* is from 1 through 255. Specification of a negative or zero value results in the default *blockfactor* setting. Values greater than 255 are defaulted to *blockfactor* modulo 256. *Blockfactor* establishes the *physical* record size on disc and magnetic tape files.

*Default: 1.*

*numbuffers*

*integer by value (optional)*

A 16-bit word whose bits specify the following:

Bits (11:5) — Number of Buffers.

Specifies the number of buffers to be allocated to the file. This parameter is not used for files representing interactive terminals, since a system-managed buffering method is always used in such cases. If



omitted, set to zero, or set to a negative number, the default value of 2 is set by MPE.

Bits (4:7) — Number of Copies.

For spooled output devices, specifies the number of copies of the entire file to be produced by the spooling facility. This can be specified for a file already FOPENed (for example, \$STDLIST), in which case the highest value supplied before the last FCLOSE will take effect. The copies do not appear contiguously if the console operator intervenes or if a file of higher *outputpriority* becomes READY before the last copy is complete. This parameter is ignored for non-spooled output devices. The default value is 1.

Bits (0:4) — Output Priority.

Specifies the *outputpriority* to be attached to this file. This priority is used to determine the order in which files are produced when several are waiting for the same device. This parameter must be a number between 1 (lowest priority) and 13 (highest priority), inclusive. If this value is less than the current output fence set by the console operator, file printing/punching is deferred until the operator raises the *outputpriority* of the file or lowers the output fence. This parameter can be specified for a file already FOPENed (for example, \$STDLIST), in which case the highest value supplied before the last FCLOSE takes effect. This parameter is ignored for non-spooled devices. The default value is 8.

*Default: The default values of all bit groupings are taken.*

*filesize*

*double by value (optional)*

A double-word integer (as defined in SPL) specifying the maximum file capacity in terms of logical records for files containing variable-length and undefined-length records, and logical records for files containing fixed-length records. A zero or negative value results in the default *filesize* setting. The maximum capacity allowed is over two million ( $2^{21}$ ) sectors. The number of sectors in a file is found by the formula shown under FILE CHARACTERISTICS in Section III.

*Default: 1023 logical records.*

*numextents*

*integer by value (optional)*

An integer specifying the number of extents (integral number of contiguously-located disc sectors) that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *filesize* parameter value divided by the *numextents* parameter value. If specified, *numextents* must be an integer from 1 to 32. A zero or negative value results in the default setting.

*Default: 8 extents.*

## NOTE

Extents are allocated on any disc in the device class specified in the *device* parameter when the file was created. If it is necessary to insure that all extents of a file are on a particular disc, a single disc device class or a logical device number must be used in the *device* parameter.

*initialloc*

*integer by value (optional)*

An integer specifying the number of extents to be allocated to the file when it is opened. This must be an integer from 1 to 32. If an attempt to allocate the requested disc space fails, the FOPEN intrinsic returns an error condition code to the calling program.

*Default: 1 extent.*

*filecode*

*integer by value (optional)*

An integer recorded in the file label and made available for general use to anyone accessing the file through the FGETINFO intrinsic. This parameter is used for new files only. For this parameter, any user can specify a positive integer ranging from 0 to 1023. If your process is running in privileged mode, you can specify a negative integer for *filecode* when initially opening a file. Then, any future accesses of the file must be requested in privileged mode and also must specify the correct *filecode*. Certain positive integers beyond 1023 have particular HP-defined meanings, as follows:

Integer	Meaning
-400	An IMAGE root file.
-401	An IMAGE data set.
1024	A USL file.
1025	A BASIC data file.
1026	A BASIC program file.
1027	A BASIC fast program file.
1028	A relocatable library (RL) file.
1029	A program file.
1030	A STAR file.
1031	A segmented library (SL) file.
1040	A Cross Loader ASCII file (SAVE).
1041	A Cross Loader relocated binary file.
1042	A Cross Loader ASCII file (DISPLAY).
1050	An EDIT KEEPQ file (non-COBOL).
1051	An EDIT KEEPQ file (COBOL).
1052	An EDIT TEXT file (COBOL).
1060	An RJE punch file.
1069	Reserved.
1070	A QUERY procedure file.
1071 }	QUERY work files.
1072 }	
1080	Reserved.

*Default: 0.*

# FOPEN

## CONDITION CODES

CCE	Request granted. The file is open.
CCG	Not returned by this intrinsic.
CCL	Request denied. This may be because another process already has exclusive or semi-exclusive access for this file, or an initial allocation of disc space cannot be made due to lack of disc space. The file number value returned by FOPEN if the file is not opened successfully is zero. The FCHECK intrinsic should be called for more details.

## TEXT DISCUSSION

Page 3-24.

Sets the logical record pointer for a disc file.

INTRINSIC NUMBER 6

```
IV DV
FPOINT(filenum,recnum);
```



The FPOINT intrinsic sets the logical record pointer for a disc file, containing only fixed-length records, to any logical record in the file. When the next FREAD or FWRITE request is issued for the file, this record will be the one read or written.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file on which the pointer is to be set.
<i>recnum</i>	<i>double by value (required)</i> A positive double integer representing the relative logical record (or block number of NOBUF files) at which the logical record pointer is to be positioned. The number of the first logical record is zero.

## CONDITION CODES

CCE	Request granted.
CCG	Request denied. The logical record pointer position is unchanged. Positioning was requested at a point beyond the physical end-of-file.
CCL	Request denied. The logical record pointer position is unchanged because of one of the following: Invalid <i>filenum</i> parameter. Input/output is pending on a no-wait I/O request. The file is spooled or is not on disc. The file does not contain fixed-length records. The NOBUF <i>option</i> of FOPEN is in effect and the FPOINT intrinsic cannot perform the requested repositioning of the logical record pointer.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-77.

# FREAD

INTRINSIC NUMBER 2

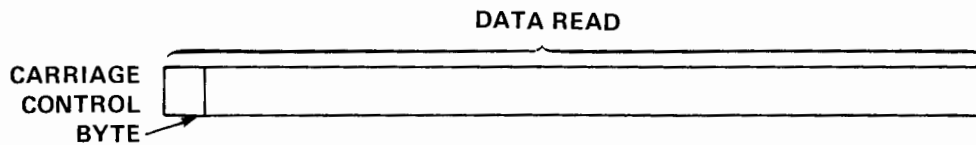
Reads a logical record from a file on any device to the user's stack.

```
I          IV LA IV  
lgth:=FREAD(filenum,target,tcount);
```

The FREAD intrinsic reads a logical record, or a portion of such a record, from a file on any device to the user's stack. The record read is determined by the current position of the record pointer.

When the logical end-of-data is encountered during reading, the CCG condition code is returned to the user process. On magnetic tape, the end-of-data can be denoted by a physical indicator such as a tape mark. On disc, the end-of-data occurs when the last logical record of the file is passed. In this case, the CCG condition code is returned *and* no record is read. If the file is embedded in an input source containing MPE commands, the end-of-data is indicated when an :EOD command is encountered, but the :EOD command itself is not returned to the user. The end-of-data is indicated by a hardware end-of-file, including :EOF:, or on \$STDIN by any record beginning with a colon, or on \$STDINX by :EOD. In addition, on the standard input device for a job, as opposed to a session, :JOB, :EOJ, or :DATA indicate end-of-data.

When an old file containing carriage-control characters, supplied through the *control* parameter of the FWRITE intrinsic, is read, and the carriage-control *foption* parameter of the FOPEN intrinsic, or the *CCTL* parameter of the :FILE command is specified, the carriage-control byte is read as follows:



(If file has carriage control specified)

## FUNCTIONAL RETURN

The FREAD intrinsic returns a positive integer value showing the length of the information transferred. If the *tcount* parameter in the FREAD call was positive, the positive value returned represents a *word* count; if the *tcount* parameter was negative, the positive value returned represents a *byte* count. FREAD always returns zero if no-wait I/O is specified. In this case, the actual record length is returned in the *tcount* parameter of the IOWAIT intrinsic.

## PARAMETERS

*filenum*

*integer by value (required)*

A word identifier supplying the file number of the file to be read.

*target*

*logical array (required)*

An array to which the record is to be transferred. This array should be large enough to hold all of the information to be transferred.

*tcount*

*integer by value (required)*

An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies the length in *words*; if it is negative, it signifies the length in *bytes*; if it is zero, no transfer occurs. If *tcount* is less than the size of the record, only the first *tcount* words or bytes are read from the record.

If *tcount* is larger than the size of the logical record, and the *multirecord aoption* was not specified in FOPEN, transfer is limited to the length of the logical record. If the *multirecord aoption* was specified in FOPEN, transfer continues until either *tcount* is satisfied or the end-of-data is encountered, and each transfer will begin at the start of the next physical record (i.e., block). Any data remaining in the last physical record read will be inaccessible.

## CONDITION CODES

CCE	The information was read.
CCG	The logical end-of-data was encountered during reading.
CCL	The information was not read because an error occurred, a terminal read was terminated by a special character as specified in the FCONTROL intrinsic, or a tape error was recovered and the FSETMODE option was enabled.

### NOTE

The condition codes should be checked both in normal I/O and in no-wait I/O.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-43.

# FREADDIR

INTRINSIC NUMBER 7      Reads a specific logical record from a disc file to the user's data stack.

```
IV LA IV DV  
FREADDIR(filename,target,tcount,recnum);
```

The FREADDIR intrinsic reads a specific logical record, or a portion of such a record, from a disc file to the user's data stack. This intrinsic differs from the FREAD intrinsic in that the FREAD intrinsic reads only the record pointed to by the logical record pointer. The FREADDIR intrinsic may be issued only for disc files composed of fixed-length or undefined-length records.

After the FREADDIR intrinsic is executed, the logical record pointer is set to the beginning of the next logical record, or first logical record of the next block for NOBUF files.

It is possible to skip portions of records inadvertently if the *multirecord aoption* of FOPEN is set and the *tcount* parameter specified is greater than one logical record. For example, if you read all of record 11 and half of record 12 in a file, the logical record pointer is set to the beginning of record 13 after the FREADDIR intrinsic executes. Thus the second half of record 12 may be skipped.

## PARAMETERS

<i>filename</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file to be read.
<i>target</i>	<i>logical array (required)</i> An array to which the record is to be transferred. This array should be large enough to hold all of the information to be transferred.
<i>tcount</i>	<i>integer by value (required)</i> An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies <i>words</i> ; if negative, it signifies <i>bytes</i> ; and if it is zero, no transfer occurs. If <i>tcount</i> is less than the size of the record, only the first <i>tcount</i> words or bytes are read from the record.  If <i>tcount</i> is larger than the size of the logical record and the <i>multirecord aoption</i> was not specified in FOPEN, the transfer is limited to the length of the logical record. If the <i>multirecord aoption</i> was specified in FOPEN, the remaining words or bytes specified in <i>tcount</i> are read from succeeding records.
<i>recnum</i>	<i>double by value (required)</i> A double-word integer indicating the relative number, in the file, of the logical record to be read. The first record is indicated by OD (double word zero in SPL notation).

## CONDITION CODES

CCE	The specified information was read.
CCG	The logical end-of-data was encountered during reading.

CCL

The information was not read because an error occurred.

## **SPECIAL CONSIDERATIONS**

Split stack calls permitted.

## **TEXT DISCUSSION**

Page 3-47.



# FREADLABEL

INTRINSIC NUMBER 19

Reads a user file label.

```
IV LA IV IV 0-V  
FREADLABEL(filenum,target,tcount,labelid);
```

The FREADLABEL intrinsic reads a user-defined label from a disc file. Before reading occurs, the user's read-access capability is verified. Note that MPE automatically skips over any unread user labels when the first FREAD intrinsic call is issued for a file; therefore, the FREADLABEL intrinsic should be called immediately after the FOPEN intrinsic has opened the file.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file whose label is to be read.
<i>target</i>	<i>logical array (required)</i> An array in the stack to which the label is to be transferred. This array should be large enough to hold the number of words specified by <i>tcount</i> .
<i>tcount</i>	<i>integer by value (optional)</i> An integer specifying the number of words to be transferred from the label. <i>Tcount</i> is limited to 128 words. <i>Default: 128 words.</i>
<i>labelid</i>	<i>integer by value (optional)</i> An integer specifying the label number. <i>Default: A default value of 0 is assigned.</i>

## CONDITION CODES

CCE	The label was read.
CCG	The intrinsic referenced a label beyond the last label written on the file.
CCL	The label was not read because an error occurred.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-63.

Moves a record from a disc file to a buffer in anticipation of a FREADDIR intrinsic call.

INTRINSIC NUMBER 12

IV    DV  
FREADSEEK(*filenum,recnum*);

Direct access of disc files can be enhanced by issuing the FREADSEEK intrinsic call. This call is used when the need for a certain record is known before its transfer to the user's stack, by a FREADDIR call, is actually required. The FREADSEEK intrinsic directs MPE to move the record from disc into a buffer in anticipation of the FREADDIR call, which subsequently moves the record directly to the stack.

## NOTE

The FREADSEEK intrinsic call can be issued only for files for which input/output buffering and fixed or undefined-length records are in effect.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file to be read.
<i>recnum</i>	<i>double by value (required)</i> A double-word integer in SPL notation indicating the relative number of the logical record to be read. The first record is indicated by OD.

## CONDITION CODES

CCE	Request granted.
CCG	A logical end-of-file indication was encountered.
CCL	Request denied because an error occurred.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-49.

# FREEDSEG

INTRINSIC NUMBER 131

Releases an extra data segment.

```
LV LV  
FREEDSEG(index,id);
```

A process can release an extra data segment assigned to it by using the FREEDSEG intrinsic. If this is a private data segment, or if it is a sharable segment not currently assigned to any other process in the job/session, the segment is deleted from the entire job/session. Otherwise, it is deleted from the calling process but continues to exist in the job/session.

## PARAMETERS

*index* *logical by value (required)*  
A word containing the logical index assigned to the data segment, obtained from the GETDSEG intrinsic call.

*id* *logical by value (required)*  
The identity, if any, assigned to the segment. If none is assigned, zero should be entered.

## CONDITION CODES

CCE Request granted. The data segment is deleted from the job/session.

CCG Request granted. The data segment is deleted from the calling process but continues to exist in the job/session because it is being shared by another process.

CCL Request denied. Either the *index* is invalid or *index* and *id* do not specify the same extra data segment.

## SPECIAL CONSIDERATIONS

Data-Segment Management Capability required.

## TEXT DISCUSSION

Page 8-15.

# **FREELOCRIN**

Frees all local RIN's from allocation to a job.

INTRINSIC NUMBER 31

## **FREELOCRIN;**

The FREELOCRIN intrinsic frees all local Resource Identification Numbers (RIN's) currently reserved for your job.

If the GETLOCRIN intrinsic has been called by a process, the FREELOCRIN intrinsic must be called before GETLOCRIN can be called successfully a second time.

## **CONDITION CODES**

CCE	Request granted.
CCG	Request denied because no RIN's are currently reserved for the job.
CCL	Request denied because at least one RIN is currently locked by a process.

## **TEXT DISCUSSION**

Page 6-9.

# FRELATE

INTRINSIC NUMBER 18

Determines whether a file pair is interactive, duplicative, or both interactive and duplicative.

```
      L           IV       IV
intordup:=FRELATE(infilenum,listfilenum);
```

A device file is *interactive* if it requires human intervention for all input operations. This quality is necessary to establish the person/machine dialog required to support a session. A device file is *duplicative* if all input operations are echoed to a corresponding display without intervention by the operating system software.

You can determine whether a pair of files is interactive, duplicative, or both interactive and duplicative through the FRELATE intrinsic call. The interactive/duplicative attributes of a file pair do not change between the time they are opened and the time they are closed.

The FRELATE intrinsic applies to files on all devices.

## NOTE

A condition code of CCG is returned when either *infilenum* or *listfilenum* corresponds to \$NULL. \$NULL is considered to be a logical file which contains no data. No data can be read from this file and all data written to it are discarded. The *infilenum* and *listfilenum* functions, therefore, are illogical for the \$NULL file.

## FUNCTIONAL RETURN

FRELATE returns a word to the calling process showing whether the two files referenced are interactive and/or duplicative. The word returned contains two significant bits, bit 15 and bit 1.

If bit 15 = 1, *infilenum* and *listfilenum* form an interactive pair.

If bit 15 = 0, *infilenum* and *listfilenum* do not form an interactive pair.

If bit 1 = 1, *infilenum* and *listfilenum* form a duplicative pair.

If bit 1 = 0, *infilenum* and *listfilenum* do not form a duplicative pair.

## PARAMETERS

*infilenum*                    *integer by value (required)*  
A word identifier supplying the file number of the input file.

*listfilenum*                *integer by value (required)*  
A word identifier supplying the file number of the list file.

## CONDITION CODES

CCE                            Request granted.

CCG Request denied because *infilenum* and/or *listfilenum* corresponds to \$NULL. Interactive or duplicative functions do not apply.

CCL Request denied because an error occurred.

## **SPECIAL CONSIDERATIONS**

Split stack calls permitted.

## **TEXT DISCUSSION**

Page 3-78.

# FRENAME

INTRINSIC NUMBER 17

Renames a disc file.

```
IV      BA
FRENAME(filenum,newfilereference);
```

The FRENAME intrinsic changes the actual designator (including *lockword*, if any) of an open disc file.

The file to be renamed must be either:

1. A new file, or
2. An existing file, opened for fully-exclusive access, for which you have write access (specified by the security provisions of the file).

## PARAMETERS

*filenum*                         *integer by value (required)*  
A word identifier supplying the file number of the file to be renamed.

*newfilereference*             *byte array (required)*  
Contains an ASCII string specifying the new name of the file. The maximum number of characters allowed in the string is 36. The format of *newfilereference* is

*filename/lockword.group.account*

where

*filename* = the new file name for the file. (Required in *newfilereference*.)

*lockword* = a lockword for the new file name. (Optional parameter of *newfilereference*.) If you wish to keep, or add a lockword to the file, you must enter the *lockword* parameter in the ASCII string. If this part of *newfilereference* is not specified, the new file named will not have a lockword associated with it.

*group* = the group name where the file is to reside. (Optional parameter of *newfilereference*.) If no group is specified, the file will reside in the group to which it was assigned before the FRENAME intrinsic call.

*account* = the account name where the file is to reside. (Optional parameter of *newfilereference*.) The account to which the file is currently assigned must be used. If other than the current account name is specified, the CCL error condition is returned and the file retains its old name.

The ASCII string contained in *newfilereference* must begin with a letter; can contain up to eight alphanumeric characters for each of the *filename*, *lockword*, *group*, and *account* fields; and must end with any non-alphanumeric character, including a blank, other than a slash (/) or a period (since period and slash are used as field delimiters within the string).

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an error occurred.

## TEXT DISCUSSION

Page 3-40.



# FSETMODE

INTRINSIC NUMBER 14

Activates or deactivates file access modes.

```
IV      LV  
FSETMODE(filename,modeflags);
```

The FSETMODE intrinsic activates or deactivates the following access mode options: automatic error recovery, critical output verification, and terminal control by the user.

The access mode established by the FSETMODE intrinsic remains in effect until another FSETMODE call is issued or until the file is closed. The FSETMODE intrinsic applies to files on all devices.

## PARAMETERS

*filename*

*integer by value (required)*

A word identifier supplying the file number of the file to which the call applies.

*modeflags*

*logical by value (required)*

A 16-bit value that denotes the access mode options in effect, as described below.

Bit (14:1) — Critical Output Verification

1 = All physical (block) output to the file is to be verified as physically complete before control returns from a write intrinsic to the user's program. The user waits while the system is posting a full block to the file. Note that this bit is effective only in buffered mode.

0 = Output is not verified.

Bit (13:1) — Terminal Control by the User

1 = Inhibit normal terminal control by the system. Thus, MPE will *not* issue an automatic carriage return and line feed at the completion of each terminal input line.

0 = MPE will automatically issue the carriage return and line feed for the terminal. This parameter is ignored if the device is not a terminal.

Bit (12:1) — Tape Error Recovery

1 = Report recovered tape error by FREAD or FWRITE with CCL condition code and error number.

0 = Report recovered tape error with CCE condition code.

The remaining 12 bits are reserved for MPE and must always be set to zero.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an error occurred.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-77.



# FSPACE

INTRINSIC NUMBER 5

Spaces forward or backward on a file.

IV IV  
**FSPACE**(*filenum*,*displacement*);

You can space forward or backward on a fixed-length or undefined-length file by using the FSPACE intrinsic. This results in resetting the logical record pointer. The FSPACE intrinsic applies to files on disc and magnetic tape devices only. On magnetic tape devices, however, FSPACE spaces *physical* rather than *logical* records.

The FSPACE intrinsic *cannot* be used with variable-length record files or with spooled files on disc. An attempt to use this intrinsic on such files results in a CCL error condition code and the logical record pointer is left at its current position.

See Section III for special considerations on magnetic tape files.

## PARAMETERS

*filenum* *integer by value (required)*  
A word identifier supplying the file number of the file on which spacing is to be done.

*displacement* *integer by value (required)*  
A signed integer indicating the number of logical records for buffered disc files, or blocks for NOBUF files and all tape files, to be spaced over, relative to the current position of the logical record pointer. A positive value signifies forward spacing, a negative value signifies backward spacing. The maximum positive value is 32767, the maximum negative value is -32768. For positive values, the sign is optional

## CONDITION CODES

CCE Request granted.

CCG A logical end-of-file indicator was encountered during spacing. For disc files, the logical record pointer is not changed. For magnetic tape files, the logical record pointer points to the logical end-of-file. The *magnetic tape*, however, is positioned to one record past the file mark on the tape.

CCL Request denied because an error occurred; the file resides on a device that prohibits spacing.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-75.

2-84



# FUPDATE

INTRINSIC NUMBER 4

Updates (writes) a logical record in a disc file.

```
IV LA IV  
FUPDATE(filenum,target,tcount);
```

The FUPDATE intrinsic updates a logical record in a disc file. This intrinsic affects the logical record (or block for NOBUF files) last referenced by any intrinsic call for the file named. FUPDATE moves the specified information from the user's stack into this record. The file containing this record must have been opened with the update *aoption* specified in the FOPEN call, and must not have variable-length records.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file to be updated.
<i>target</i>	<i>logical array (required)</i> Contains the record to be written in the updating.
<i>tcount</i>	<i>integer by value (required)</i> An integer specifying the number of words or bytes to be written from the record. If this value is positive, it signifies words; if it is negative, it signifies bytes; if it is zero, no transfer occurs. If <i>tcount</i> is less than the <i>resize</i> parameter associated with the record, only the first <i>tcount</i> bytes or words are written. For buffered file, <i>tcount</i> is limited to the block size. FUPDATE cannot perform multirecord updates.

## CONDITION CODES

CCE	Request granted.
CCG	An end-of-file condition was encountered during updating.
CCL	Request denied because of an error, such as the file not residing on disc, or <i>tcount</i> exceeding the size of the record when multirecord mode is not in effect.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-54.

Writes a logical record from the user's stack to a file on any device.

INTRINSIC NUMBER 3

```
IV LA IV LV  
FWRITE(filenum,target,tcount,control);
```

The FWRITE intrinsic writes a logical record, or a portion of such a record, from the user's stack to a file on any device.

When information is written to a fixed-length record, and the NOBUF *option* was not specified in FOPEN, any unused portion of the record will be padded with binary zeros for a binary file or ASCII blanks for an ASCII file.

When the FWRITE intrinsic is executed, the logical record pointer is set to the record immediately following the record just written, or the first logical record in the next block for NOBUF files.

When an FWRITE call writes a record beyond the current logical end-of-file indicator, this indicator is advanced to a farther location; however, this is only noted in the file label when the file is actually closed. If the physical bounds of the file are reached, the CCL condition code is returned.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file to be written on.
<i>target</i>	<i>logical array (required)</i> Contains the record to be written.
<i>tcount</i>	<i>integer by value (required)</i> An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; if it is negative, it signifies bytes; if it is zero, no transfer occurs. If <i>tcount</i> is less than the <i>reclsize</i> parameter associated with the record, only the first <i>tcount</i> words or bytes are written.  If <i>tcount</i> is larger than the <i>reclsize</i> value, and the <i>multirecord option</i> was not specified in FOPEN, the FWRITE request is refused and condition code CCL is returned. If the <i>multirecord option</i> was specified in FOPEN, the excess words or bytes are written to succeeding records. For files for which carriage control is specified, the actual data transferred is limited to <i>reclsize</i> minus one byte.
<i>control</i>	<i>logical by value (required)</i> A logical value representing a carriage control code, effective if the file is transferred to a line printer or terminal (indicating a spooled file whose ultimate destination is a line printer or a terminal). This parameter is effective only for files opened with carriage control specified.

# FWRITE

The options are

- 0 = Print the full record transferred, using single spacing. This results in a maximum of 132 characters per printed line.
- 1 = Use the first character of the data written to signify space control, and suppress this character on the printed output. This results in a maximum of 132 characters of data per printed line. Permissible control characters are shown in figure 2-3.

Any octal code from figure 2-3 can be used to determine space control and print the full record transferred. This results in a maximum of 132 characters per printed line.

If the *control* parameter is not 0 or 1, and *tcount* is 0, only the space control is executed — no data are transferred.

The effect of the FWRITE *control* parameter in combination with the FOPEN carriage control *foption* (or overriding :FILE command CCTL/NOCTL parameter) upon the data written is summarized in figure 2-4.

You determine whether the carriage control directive takes effect before printing (pre-space movement) or after printing (post-space movement), through the FCONTROL intrinsic.

All of the carriage control codes listed in figure 2-3 may be used as the value of the *param* parameter in FCONTROL (when *controlcode* = 1), regardless of whether the file is opened with CCTL or NOCTL. When the file is opened with CCTL, these carriage control codes may be used in either of the following ways via FWRITE:

- a. As the value of the *control* parameter.
- b. When *control* = 1, as the first byte of the *target* array.

The default carriage control code is post spacing with automatic page eject. This applies to all HP-supported subsystems except FORTRAN which is prespacing with automatic page eject.

## CONDITION CODES

CCE	Request granted.
CCG	The physical bounds of the file prevented further writing; all disc extents are filled.
CCL	Request denied because an error occurred, such as <i>tcount</i> exceeding the size of the record in non-multirecord mode; or the FSETMODE option is enabled to signify recovered tape errors; or the end-of-tape marker was sensed.

Octal Code	ASCII Symbol	Carriage Action
%40	" "	*Single-space.
%60	" 0 "	*Double-space.
%61	" 1 "	Page-eject (form-feed).
%53	" + "	No space, return (next printing at column 1).
%2nn		Space nn lines. (No automatic page eject.)
(where n is any digit from 0 through 7)		
%300		Page-eject (Tape Channel 1).
%301		Skip to bottom of form (Tape Channel 2).
%302		Single-spacing (with automatic page eject). (Tape Channel 3.)
%303		Single-space on next odd-numbered line (with automatic page eject). (Tape Channel 4.)
%304		Triple-space (with automatic page eject). (Tape Channel 5.)
%305		Space 1/2 page (with automatic page eject). (Tape Channel 6.)
%306		Space 1/4 page (with automatic page eject). (Tape Channel 7.)
%307		Space 1/6 page (with automatic page eject). (Tape Channel 8.)
%310		Space to bottom of form. (Tape Channel 9.)
%311		Skip to Tape Channel 10. (User option.)
%312		Skip to Tape Channel 11. (User option.)
%313		Skip to Tape Channel 12. (User option.)
%320		No space, no return. (Next printing physically follows this.)
%0 – %37	}	Same as %40
%41 – %52		
%54 – %57		
%62 – %77		
%314 – %317		
%321 – %377		
%400 or %100		Set post-space movement option; this first prints, then spaces. If previous option set was pre-space movement option, the driver outputs a line (and suppresses spacing) to clear the buffer.
%401 or %101		Set pre-space movement option; this first spaces, then prints.
%402 or %102		Set single-space option, with automatic page eject (60 lines per page).
%403 or %103		Set single-space option <i>without</i> automatic page eject (66 lines per page).
*Spacing with or without automatic page eject can be selected.		

Figure 2-3. Carriage-Control Directives



# FWRITE

FOPEN OR :FILE	FWRITE Control Parameter		
	= 0	= 1	= Greater than 1
Carriage Control Foption Specified or CCTL	<p>Byte 1      resize 133</p> <p>Data output contains 132 characters; the prefix byte is added and contains 0</p>	<p>resize 132</p> <p>Data output contains 132 characters; the carriage control character in the first byte is not printed if output is to a list device.</p>	<p>Byte 1      resize 133</p> <p>Data output contains 132 characters; the prefix character added is a carriage-control character specified by the FWRITE <i>control</i> parameter.</p>
Carriage Control Foption <i>not</i> specified or NOCCTL	<p>132</p> <p>Data output contains 132 characters.</p>	<p>132</p> <p>Data output contains 132 characters</p>	<p>132</p> <p>Data output contains 132 characters.</p>

EFFECT ON DATA OUTPUT

Figure 2-4. Carriage-Control Summary

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-46.

Writes a specific logical record from the user's stack to a disc file.

INTRINSIC NUMBER 8

```
IV LA IV DV
FWRITEDIR(filenum,target,tcount,recnum);
```

The FWRITEDIR intrinsic writes a specific logical record, or a portion of such a record, from the user's stack to a disc file. This intrinsic differs from the FWRITE intrinsic in that the FWRITE intrinsic writes only the record pointed to by the logical record pointer. The FWRITEDIR intrinsic may be used only for disc files composed of fixed or undefined-length records.

When information is written to a fixed-length record and NOBUF was not specified in the FOPEN call that opened the file, any unused portion of the record will be padded with binary zeros for a binary file, or ASCII blanks for an ASCII file.

When the FWRITEDIR intrinsic is executed, the logical record pointer is set to the record immediately following the record just written, or the first logical record of the next block for NOBUF files.

When an FWRITEDIR call writes a record beyond the current logical end-of-file indicator, the indicator is advanced to a farther location. This can result in the creation of dummy records to pad the records between the previous end-of-file and the newly-written record. These dummy records are filled with binary zeros for a binary file, or with ASCII blanks for an ASCII file.

When the physical bounds of the file prevent further writing, because all allowable extents are filled, the end-of-file condition (CCG) is returned to the user's program.

## PARAMETERS

*filenum*                                    *integer by value (required)*  
A word identifier specifying the file number of the file to be written on.

*target*                                    *logical array ( required)*  
Contains the record to be written. This array should be large enough to hold all of the information to be transferred.

*tcount*                                    *integer by value (required)*  
An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; if it is negative, it signifies bytes; if it is zero, no transfer occurs. If *tcount* is less than the *resize* parameter associated with the record, only the first *tcount* words or bytes are written.

If *tcount* is larger than the size of the logical record and the *multirecord aoption* was not specified in FOPEN, the transfer is limited to the length of the logical record. If the *multirecord aoption* was specified in FOPEN, the remaining words or bytes are written to succeeding records.

# **FWRITEDIR**

*recnum*

*double by value (required)*

A double integer indicating the relative number of the logical record, or block number for NOBUF files, to be written. The first record is indicated by OD.

## **CONDITION CODES**

CCE

Request granted.

CCG

The physical end-of-file was encountered.

CCL

Request denied because an error occurred.

## **SPECIAL CONSIDERATIONS**

Split stack calls permitted.

## **TEXT DISCUSSION**

Page 3-49.

# FWRITELABEL

Writes a user file label.

INTRINSIC NUMBER 20

```
IV LA IV IV 0-V  
FWRITELABEL(filenum,target,tcount,labelid);
```

The FWRITELABEL intrinsic writes a user-defined label onto a disc file. This intrinsic overwrites old user labels.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier specifying the file number of the file to which the label is to be written.
<i>target</i>	<i>logical array (required)</i> Contains the label to be written to the disc file.
<i>tcount</i>	<i>integer by value (optional)</i> An integer specifying the number of words to be transferred from the array. <i>Default: 128 words.</i>
<i>labelid</i>	<i>integer by value (optional)</i> An integer specifying the number of the label to be written. The first label is 0. <i>Default: A default value of 0 is assigned.</i>

## CONDITION CODES

CCE	Request granted.
CCG	Request denied because the calling process attempted to write a label beyond the limit specified in FOPEN when the file was opened.
CCL	Request denied because an error occurred.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 3-59.

# GETDSEG

INTRINSIC NUMBER 150

Creates an extra data segment.

```
      L      ILV  
GETDSEG(index,length,id);
```

The GETDSEG intrinsic creates or acquires an extra data segment. The number of extra data segments that can be requested, and the maximum size allowed these segments, are limited by parameters specified when the system is configured. When an extra data segment is created, the GETDSEG intrinsic returns a *logical index number* to the calling process. This index number is assigned by MPE and allows this process to reference the segment in later intrinsic calls. The GETDSEG intrinsic also is used to assign the segment the *identity* that either allows other processes in the job or session to share the segment, or that declares it private to the calling process. If the segment is sharable, other processes can obtain its logical index (through GETDSEG) and use this index to reference the segment. Thus, the logical index is a local name that identifies the segment throughout any process that obtained the index with the GETDSEG call. The logical index need not be the same value in all processes sharing the data segment. The *identity*, on the other hand, is a job-wide or session-wide name that permits any process to determine the logical index of the segment.

## PARAMETERS

*index*

*logical (required)*

A word to which the logical index of the data segment, assigned by MPE, is returned.

*length*

*integer (required)*

The maximum size of the data segment requested, if the segment is not yet created, or the word to which the maximum size of the segment is returned, if the segment already exists.

*id*

*logical by value (required)*

A word containing the identity that declares the data segment sharable between other processes in the job/session, or private to the calling process. For a sharable segment, *id* is specified as a non-zero value. If a data segment with the same *id* exists already, it is made available to the calling process. Otherwise, a new data segment, sharable within the job/session, is created with this *id*. For a private data segment, an *id* of zero must be specified.

## CONDITION CODES

CCE

Request granted. A new segment was created.

CCG

Request granted. An extra data segment with this identity exists already.

CCL

Request denied. An illegal length was specified (*index* is set to %2000), or the process requested more than the maximum allowable number of data segments (*index* is set to %2001), or sufficient storage was not available for the data segment (*index* is set to %2002).

## SPECIAL CONSIDERATIONS

Data Segment Management Capability required.

## TEXT DISCUSSION

Page 8-6.



# GETJCW

INTRINSIC NUMBER 73

Fetches content of job control word.

```
L  
jcw:=GETJCW;
```

The GETJCW intrinsic returns the complete job control word to the calling process.

## FUNCTIONAL RETURN

This intrinsic returns the job control word. This word is structured for a desired purpose by the calling program through the SETJCW intrinsic.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-45.

Acquires local RIN's. The number acquired = *rincount*.

INTRINSIC NUMBER 30

```
LV  
GETLOCRIIN(rincount);
```

Just as *global* Resource Identification Numbers (RIN's) must be acquired by users before they can be used in jobs/sessions, *local* RIN's must be acquired by a job/session before they can be used within the job/session. This is done by using the GETLOCRIIN intrinsic.

## PARAMETERS

*rincount*

*logical by value (required)*

The number of local RIN's to be acquired by the job/session. The maximum number of RIN's available is defined when the system is configured.

## CONDITION CODES

CCE

Request granted.

CCG

Request denied. RIN's already are allocated to this job. Additional RIN's cannot be allocated until these RIN's are released.

CCL

Request denied. Not enough RIN's are available to satisfy this call. None are allocated to this job.

## TEXT DISCUSSION

Page 6-8.



# GETORIGIN

INTRINSIC NUMBER 105

Determines source of activation call.

```
I  
source:=GETORIGIN;
```

After a suspended process is reactivated, it can determine whether the source of the activation request was its father process, one of its son processes, or whether the reactivation was by an interrupt or the timer.

## FUNCTIONAL RETURN

This intrinsic returns one of the following codes:

- 1 = Activated by father.
- 2 = Activated by a son.
- 3 = Activated by interrupt.
- 4 = Activated from timer.

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-14.

Reschedules a process.

INTRINSIC NUMBER 120

```
IV      LV IV 0-V
GETPRIORITY(pin, priorityclass, rank);
```

When a process is created, it is scheduled on the basis of a priority class assigned by its father. After this point, the priority class of the created process can be changed at any time by using the GETPRIORITY intrinsic.

## NOTE

A process can change its own priority or that of its son but it cannot reschedule its father.

## PARAMETERS

- pin* *integer by value (required)*  
An integer specifying the process whose priority is to be changed. If this is a son process, the integer is the process Process Identification Number (PIN). If this is the calling process, the integer is zero.
- priorityclass* *logical by value (required)*  
A 16-bit word that contains two ASCII characters describing the priority class in which the process is rescheduled. This may be "AS", "BS", "CS", "DS", or "ES". For users with optional capabilities (other than Process Handling), this class may be any priority permitted by those optional capabilities. The *priorityclass* parameter may then be specified by *x*A, where *x* is an 8-bit priority number and A is the ASCII character "A". For example, a request for a *priorityclass* of 31 in the master queue would be requested as %017501.
- rank* *integer by value (optional)*  
This parameter is used only for compatibility with previous versions of the MPE Operating System. It is ignored for all users.

## CONDITION CODES

- CCE Request granted.
- CCG Request denied because the process specified is not alive.
- CCL Request denied because an illegal PIN was specified.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.  
Process Handling Capability required.

## TEXT DISCUSSION

Page 7-13.

# GETPRIVMODE

INTRINSIC NUMBER 200

Dynamically enters privileged mode.

GETPRIVMODE; O-P

The GETPRIVMODE intrinsic switches a temporarily-privileged program from the non-privileged mode to the privileged mode. This intrinsic turns the privileged mode bit in the status register *on*, but leaves the privileged mode bit in the Code Segment Table (CST) entry for the executing segment unchanged. The status register, rather than the CST, determines a mode change when running in privileged mode. Thus, if additional segments are to be run as part of the program, they will be run in privileged mode unless an intrinsic is called specifically to return to the non-privileged mode.

The calling process is aborted if the program file does not possess the Privileged Mode Capability, and the CST indicates non-privileged mode.

## CONDITION CODES

CCE	Request granted. The program was in non-privileged mode when the intrinsic call was issued.
CCG	Request granted. The program was already in privileged mode when the intrinsic call was issued.
CCL	Not returned by this intrinsic.

## SPECIAL CONSIDERATIONS

Privileged Mode Capability required.

## TEXT DISCUSSION

Page 9-3.

### IMPORTANT NOTE

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard will investigate and attempt to resolve problems resulting from the use of privileged mode code. This service, which is not provided under the standard Service Contract, is available on a time and materials billing basis. However, Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.

Requests PIN of a son process.

INTRINSIC NUMBER 112

```
I          LV
pin:=GETPROCID(numson);
```

A process can determine the Process Identification Number (PIN) assigned to any of its sons by using the GETPROCID intrinsic.

## FUNCTIONAL RETURN

This intrinsic returns the PIN of the specified son process.

## PARAMETERS

*numson*

*logical by value (required)*

A number from 1 to  $n$  which specifies the chronological son's PIN desired. The value  $n$  cannot exceed the number of sons in existence. For example, a father process has three sons and it is desired to know the PIN of the second son. The value of *numson* then would be 2.

If  $n$  exceeds the number of sons currently attached to this calling process, a zero is returned. If  $n$  is less than 1, the PIN of the first son (or zero if no sons exist) is returned.

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-15.

# GETPROCINFO

INTRINSIC NUMBER 110

Requests status information about a father or son process.

```
D          IV  
statinfo:=GETPROCINFO(pin);
```

Information about a father or son process can be obtained with the GETPROCINFO intrinsic.

## FUNCTIONAL RETURN

This intrinsic returns a double-word message denoting the following information about a father or son process:

Word 1:

Bits (8:8) — The process' priority number in the master queue.

Bits (0:8) — Reserved for MPE. These bits are set to zero by the system.

Word 2:

Bit (15:1) — Activity state.

1 = The process is active.

0 = The process is suspended.

Bit (13:2) — Suspension condition. Set *only* if bit 15 = 0.

If bit 14 = 1, the source of the expected activation is the father.

If bit 13 = 1, the source of the expected activation is a son.

Bits (9:4) — Reserved for MPE. These bits are set to zero by the system.

Bits (7:2) — Origin of the last ACTIVATE intrinsic call.

00 = The process was activated by MPE.

01 = The process was activated by the father.

10 = The process was activated by a son.

Bits (4:3) — Queue Characteristics.

001 = DS or ES priority class.

010 = CS priority class.

100 = Linearly scheduled (AS or BS or Master queue).

Bits (0:4) — Reserved for MPE. These bits are set to zero by the system.

## PARAMETERS

*pin*

*integer by value (required)*

The process to which the returned message pertains. If this is a request for a father process, *pin* must be zero. If it is a request for a son process, *pin* is the PIN of that process.

## CONDITION CODES

CCE

Request granted.

CCG

Request denied because the process is being terminated.

CCL

Request denied because an illegal PIN was specified.

## **SPECIAL CONSIDERATIONS**

Split stack calls permitted.  
Process Handling Capability required.

## **TEXT DISCUSSION**

Page 7-15.

# GETUSERMODE

INTRINSIC NUMBER 201

Dynamically returns to non-privileged mode.

```
GETUSERMODE;
```

The GETUSERMODE intrinsic changes a temporarily-privileged program from the privileged to the non-privileged mode.

This intrinsic changes the privileged mode bit in the status register to *off*, and is the complement of the GETPRIVMODE intrinsic.

## CONDITION CODES

CCE	Request granted. The process was in privileged mode when the intrinsic call was issued.
CCG	Request granted. The program was in non-privileged mode when the intrinsic call was issued.
CCL	Not returned by this intrinsic.

## SPECIAL CONSIDERATIONS

Privileged Mode Capability required.

## TEXT DISCUSSION

Page 9-5.

Initializes buffer for a USL file to the empty state.

INTRINSIC NUMBER 82

```

      I          IV IA
errnum:=INITUSLF(uslfnm,rec0);

```

The INITUSLF intrinsic initializes the first record (record 0) of a USL file to the empty state.

## FUNCTIONAL RETURN

This intrinsic returns an error number if an error occurs. If no error occurs, no value is returned.

## PARAMETERS

<i>uslfnm</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the USL file.
<i>rec0</i>	<i>integer array (required)</i> A 128-word buffer, corresponding to the first record of the USL file (record 0), to be initialized to the empty state. This buffer should be set to all zeros. The intrinsic will set certain values in record 0 before returning to the calling program. See the <i>MPE Segmenter Reference Manual</i> for record 0 format.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied. One of the error numbers listed below is returned.

Error Number	Meaning
0	The file specified by <i>uslfnm</i> was empty, or an unexpected end-of-file was encountered when reading the old <i>uslfnm</i> , or an unexpected end-of-file was encountered when writing on the <i>new uslfnm</i> .
1	Unexpected input/output error occurred. This can occur on the old <i>uslfnm</i> or the new <i>uslfnm</i> to which the intrinsic is copying the information.
3	Your request attempted to exceed the maximum file directory size (32,768 words).
6	Insufficient space was available in the USL file information block.



# INITUSLF

Error Number	Meaning
7	The intrinsic was unable to open the new USL file.
8	The intrinsic was unable to close (purge) the old USL file.
9	The intrinsic was unable to close (purge) the new USL file.
10	The intrinsic was unable to close \$NEWPASS.
11	The intrinsic was unable to open \$OLDPASS.

## TEXT DISCUSSION

See the *MPE Segmenter Reference Manual*



# IODONTWAIT

*cstation*

*logical (optional)*

This parameter must be included in the intrinsic declaration statement but is otherwise ignored by MPE.

## CONDITION CODES

CCE

Request granted. If the functional return is non-zero then I/O completion occurred with no errors. If the return is zero then no I/O has completed.

CCG

An end-of-file condition was encountered.

CCL

Request denied. Normal I/O completion did not occur because there were no I/O requests pending, a parameter error occurred, or an abnormal I/O completion occurred.

## SPECIAL CONSIDERATIONS

You must be running in Privileged Mode to specify FOPEN *options* No-Wait I/O and NOBUF.

## TEXT DISCUSSION

Page 3-59.

Initiates completion operations for an I/O request.

INTRINSIC NUMBER 22

```

I      IV LA  I  L O-V
fnum := IOWAIT(filenum, target, tcount, cstation);

```

If a file has been opened with the no-wait I/O mode *aoption* of the FOPEN intrinsic (*aoptions* bit (4:1) = 1), all read and write requests must be followed by the IOWAIT intrinsic call. This intrinsic initiates completion operations for the associated I/O request, including data transfer into the user's buffer area if necessary.

The IOWAIT intrinsic call must precede any subsequent I/O request against the file. Within this restriction, the IOWAIT intrinsic call can be delayed as long as desired to allow effective I/O and processing overlap.

### FUNCTIONAL RETURN

This intrinsic returns an integer representing the file number for which the completion occurred. If no completion occurred, zero is returned.

### PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier specifying the file number for which there is a pending I/O request. If zero is specified, the IOWAIT intrinsic will wait for the first I/O completion.
<i>target</i>	<i>logical array (optional)</i> A word pointer specifying the DB-relative address of the user's input buffer. This buffer must be large enough to contain the input record. Note that this parameter is required if the original I/O request was a read and the file was opened with buffering specified.
<i>tcount</i>	<i>integer (optional)</i> A word to which is returned a positive integer representing the length of the received or transmitted record. If the original request specified a byte count, the integer represents bytes; if the request specified words, the integer represents words. Note that this parameter is pertinent only if the original request was a read. The FREAD intrinsic always returns zero as its functional return if no-wait I/O is specified. In this case, the actual record length is returned in the <i>tcount</i> parameter of IOWAIT. <i>Default: The length of the record is not returned.</i>
<i>cstation</i>	<i>logical (optional)</i> This parameter must be included in the intrinsic declaration statement but is otherwise ignored by MPE.

# IOWAIT

## CONDITION CODES

CCE	Request granted. I/O completion occurred with no errors.
CCG	An end-of-file condition was encountered.
CCL	Request denied. Normal I/O completion did not occur because there were no I/O requests pending, a parameter error occurred, or an abnormal I/O completion occurred.

## SPECIAL CONSIDERATIONS

You must be running in Privileged Mode to specify FOPEN *options* No-Wait I/O and NOBUF.

## TEXT DISCUSSION

Page 3-57.



# LOADPROC

INTRINSIC NUMBER 80

Dynamically loads a library procedure.

```
I          BA IV    I  
identnum:=LOADPROC(procname,lib,plabel);
```

The LOADPROC intrinsic dynamically loads a library procedure, together with external procedures referenced by it.

## FUNCTIONAL RETURN

This intrinsic returns an identity number required for use in unloading the procedure. If an error occurs, an error code number is returned instead of the identity number.

## PARAMETERS

<i>procname</i>	<i>byte array (required)</i> Contains the name of the procedure to be loaded. The name must be terminated by a blank.
<i>lib</i>	<i>integer by value (required)</i> An integer value of 0, 1, or 2, to request library searching for the procedure, as follows: 0 = Search the system library only. 1 = Search libraries in this order: Account public library. System library. 2 = Search libraries in this order: Group library. Account public library. System library.
<i>plabel</i>	<i>integer (required)</i> The word to which the procedure's label (P-label) is returned. This is the external P-label so that the SPL construct ASSEMBLE (PCAL 0) may be used.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied. The value returned to the calling process is a loader error code. See Section X for a description of error codes.

## TEXT DISCUSSION

Page 4-2.

Locks a global RIN.

INTRINSIC NUMBER 34

```
LV      L      BA
LOCKGLORIN(rinum,lockcond,rinpassword);
```



Any global Resource Identification Number (RIN) assigned to a group of users can be locked, by one job at a time, by using the LOCKGLORIN intrinsic. When this is done, any other jobs that attempt to lock this RIN are suspended.

To use the LOCKGLORIN intrinsic, you must know both the RIN number and the RIN password. Users with only Standard MPE Capability cannot lock more than one global RIN simultaneously. An attempt by such a user to lock more than one RIN simultaneously aborts the process.

## PARAMETERS

- rinum*** *logical by value (required)*  
A word identifier specifying the RIN number of the resource to be locked. This is the RIN number furnished in the :GETRIN command. See the *MPE Commands Reference Manual* for a description of the :GETRIN command.
- lockcond*** *logical (required)*  
A word identifier specifying conditional or unconditional RIN locking, as follows:  
TRUE = Locking will take place unconditionally. If the RIN is not available, the calling process suspends until it becomes available. The TRUE condition is passed as a word in which bit 15 is 1. All other bits are ignored.  
FALSE = Locking takes place only if the RIN is immediately available. If the RIN is not immediately available, control returns to the calling process immediately with the condition code CCG. The FALSE condition is passed as a word in which bit 15 is 0. All other bits are ignored.
- rinpassword*** *byte array (required)*  
Contains the RIN password assigned through the :GETRIN command. This array must be a minimum of 10 bytes in length and must be terminated by a non-alphanumeric ASCII character (a blank is recommended).

## CONDITION CODES

The condition codes possible if *lockcond* = TRUE are

- CCE Request granted. If the calling process had already locked the RIN, FALSE is returned to the word *lockcond*. If the RIN was free, TRUE is returned to *lockcond*.
- CCG Not returned.



# LOCKGLORIN

CCL Request denied because of invalid RIN. *Rinum* is not a global RIN or the value is out of bounds for the RIN table.

The condition codes possible if *lockcond* = FALSE are

CCE Request granted. If the calling process had already locked the RIN, FALSE is returned to the word *lockcond*. If the RIN was free, TRUE is returned to *lockcond*.

CCG Request denied because the RIN was locked by another job.

CCL Not returned.

## SPECIAL CONSIDERATIONS

Multiple RIN Capability is required if you are going to lock more than one global RIN at a time within a process.

## TEXT DISCUSSION

Page 6-3.

Locks a local RIN.

INTRINSIC NUMBER 32

```
LV L  
LOCKLOCRIN(rinnum,lockcond);
```

Any local Resource Identification Number (RIN) assigned to a job can be locked, by one process at a time, by using the LOCKLOCRIN intrinsic. When this is done, other processes within the job that attempt to lock that RIN are suspended until the locked RIN is released.

## PARAMETERS

<i>rinnum</i>	<i>logical by value (required)</i> An identifier specifying one of the previously-allocated local RIN's, designated by an integer from 1 to the value specified in the <i>rincount</i> parameter of the GETLOCKRIN intrinsic.
<i>lockcond</i>	<i>logical (required)</i> A word identifier specifying conditional or unconditional locking, as follows: TRUE = Locking takes place unconditionally. If the RIN is not available, the calling process suspends until the RIN becomes available. The TRUE condition is passed as a word in which bit 15 is 1. All other bits are ignored. FALSE = Locking takes place only if the RIN is immediately available. If it is not, control returns to the calling process immediately with the condition code CCG. The FALSE condition is passed as a word in which bit 15 is 0. All other bits are ignored.

## CONDITION CODES

The condition codes possible if *lockcond* = TRUE are

CCE	Request granted. If the calling process had already locked the RIN, TRUE is returned to the word <i>lockcond</i> . If the RIN was free, FALSE is returned to <i>lockcond</i> .
CCG	Not returned.
CCL	Request denied because the RIN was invalid. Possibly due to: <i>rinnum</i> too large, no local RIN allocated, or <i>rinnum</i> specified a number less than or equal to zero.

The condition codes possible if *lockcond* = FALSE are

CCE	Request granted. If the calling process had already locked the RIN, TRUE is returned to the word <i>lockcond</i> . If the RIN was free, FALSE is returned to <i>lockcond</i> .
-----	--

# LOCKLOCRIN

CCG Request denied because the RIN was locked by another process.

CCL Request denied because the RIN was invalid. Possibly due to: *rinum* too large, no local RIN allocated, or *rinum* specified a number less than or equal to zero.

## TEXT DISCUSSION

Page 6-8.

Tests mailbox status.

INTRINSIC NUMBER 106

```

L      IV      I
status:=MAIL(pin,count);

```

A process can determine the status of the mailbox used by its father or son with the MAIL intrinsic. If the mailbox contains mail that is awaiting collection by this process, the length of this message (in words) is returned to the calling process in the *count* parameter. This enables the calling process to initialize its stack in preparation for receipt of the message.

## FUNCTIONAL RETURN

This intrinsic returns the status of the mailbox, as follows:

Status Returned	Meaning
0	The mailbox is empty.
1	The mailbox contains previous <i>outgoing</i> mail from this calling process that has not yet been collected by the destination process.
2	The mailbox contains <i>incoming</i> mail awaiting collection by this calling process. The length of the mail is returned in <i>count</i> .
3	An error occurred because an invalid <i>pin</i> was specified or a bounds check failed.
4	The mailbox is temporarily inaccessible because other intrinsics are using it in the preparation or analysis of mail.

## PARAMETERS

*pin*                                    *integer by value (required)*  
 An integer specifying the mailbox tested. If this integer specifies the mailbox of a son process, it must be the Process Identification Number (PIN) of that son. Zero specifies the mailbox of a father process.

*count*                                    *integer (required)*  
 A word to which an integer denoting the length, in words, of any incoming mail in the mailbox is to be returned.

## CONDITION CODES

CCE                                    Request granted. The mailbox status was tested.

# MAIL

CCG Request denied because an illegal *pin* parameter was specified. The value of 3 is returned to the calling process.

CCL Not returned by this intrinsic.

## SPECIAL CONSIDERATIONS

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-10.

Parses (delineates and defines parameters) user-supplied command image. INTRINSIC NUMBER 71

```
I          BA    BA    IV    I  DA BA BP 0-V  
entryno:=MYCOMMAND(comimage,delimiters,maxparms,numparms,parms,dict,defn);
```

Within your program, you can extract and format for execution the parameters of a command (that is *not* an MPE command) by using the MYCOMMAND intrinsic. This intrinsic also allows you, at your option, to request the searching of a byte array, serving as a command dictionary, for a specified command.

The MYCOMMAND intrinsic aborts the calling process if the number of characters in *comimage* exceeds 255 characters *and* no delimiter is present.

## FUNCTIONAL RETURN

If the *dict* parameter is specified in the intrinsic call, the command entry number is returned.

## PARAMETERS

*comimage*

*byte array (required)*

Contains either:

- A command name (expected if the *dict* parameter is specified), followed by parameters, followed by a carriage-return character. The command name is delimited by the first non-alphanumeric character, and cannot be preceded by any leading blanks. The parameters are formatted and referenced in *parms* array. Also, *comimage* is converted to upper case and the byte array specified by *dict* is searched for a name matching the command.
- Only command parameters (expected if the *dict* parameter is not specified), followed by a carriage-return character. These parameters will be formatted.

In the byte array named for the *comimage* parameter, the first character of the command or parameter list may be a leading blank.

*delimiters*

*byte array (optional)*

A byte array containing a string of up to 32 legal delimiters, each of which is an ASCII special character. The last character must be a carriage return. Each delimiter is identified later by its position in this string.

*Default: If this parameter is omitted, the delimiter array "comma, equal, semicolon, carriage return" is used.*

*maxparms*

*integer by value (required)*

An integer specifying the maximum number of parameters expected in *comimage*.

*numparms*

*integer (required)*

A word to which is returned the actual number of parameters found in *comimage*.

# MYCOMMAND

*parms*

*double array (required)*

A double array of *maxparms* double words that, on return, delineates the parameters. When the intrinsic is executed, the first *numparms* double words are returned to the user's process in this array, with the first double word corresponding to the first parameter, the second double word corresponding to the next parameter, and so forth. The parameter fields of *comimage* are delimited by the delimiters specified in *delimiters*. In formatting, the byte pointer in the first word of *parms* points to the parameter in *comimage*. The string in *comimage* is upshifted. The second word of *parms* contains the delimiter number and parameter information. Each double word in the array named by *parms* contains the following information:

Word 1 — Contains the byte pointer to the first character of the parameter.

Word 2 — Contains bits that describe the parameter:

Bits (11:5) = The delimiter number in *delimiters*, starting at zero.

Bit (10:1) = If *on*, indicates that the parameter contains special characters other than those in *delimiters*.

Bit (9:1) = If *on*, indicates that the parameter contains numeric characters.

Bit (8:1) = If *on*, indicates that the parameter contains alphabetic characters.

Bits (0:8) = The length of the parameter, in bytes. This value is zero if the parameter is omitted.

*dict*

*byte array (optional)*

A byte array that will be searched for the command name in *comimage*. The format must be identical to that of the *dict* parameter in the SEARCH intrinsic. Actually, the command, delimited by a blank, is extracted from *comimage*, and the SEARCH intrinsic is called with the command name used as the *target* parameter in SEARCH. If the command name is found in *dict*, its entry number is returned to the user's program. If the command is *not* found, or if the *dict* parameter is not specified, zero is returned. If *dict* is specified but the command name is not found in *dict*, the parameters specified in *comimage* are not formatted.

*Default: 0 is returned.*

*defn*

*byte pointer (optional)*

A word to which is returned the relative address of the definition portion of the command entry in *dict*.

*Default: The corresponding information is not returned.*

## CONDITION CODES

CCE

The parameters were formatted, without exception. If *dict* was specified, the command entry number was returned to the user's program.

CCG More parameters were found in *comimage* than were allowed by *maxparms*. Only the first *maxparms* of these parameters were formatted in *parms* and returned to the user.

CCL The *dict* parameter was specified, but the command name was not located in the array *dict*. The parameters in *comimage* were not formatted.

## TEXT DISCUSSION

Page 4-4.



# PAUSE

INTRINSIC NUMBER 45

Suspends the calling process for a specified number of seconds.

**R**  
**PAUSE(*interval*);**

## PARAMETERS

*interval*

*real (required)*

A positive real value specifying the amount of time, in seconds, that the process will pause. The maximum time allowed is approximately 2,147,484 seconds.

## CONDITION CODES

CCE

Request granted.

CCG

Not returned by this intrinsic.

CCL

Request denied because a negative value was specified for *interval* or the value is too large.

## TEXT DISCUSSION

Page 4-19.

Prints character string on job/session listing device.

INTRINSIC NUMBER 65

```

    LA    IV    IV
    PRINT(message,length,control);

```

You can write a string of ASCII characters from an array to the job/session listing device by using the PRINT intrinsic. This intrinsic is similar to issuing an FWRITE intrinsic call against the file \$STDLIST. The PRINT intrinsic is limited in its usefulness, however, in that the full capability of the file system is not available to a user of this intrinsic. For example, :FILE commands are not allowed and certain file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not available to normal users of the PRINT intrinsic.

## PARAMETERS

*message*                      *logical array (required)*  
 Contains the character string to be output.

### NOTE

SPL programmers can avoid annoying warning messages in the compiled output by equivalencing a byte array to a logical array for the *message* parameter.

*length*                      *integer by value (required)*  
 An integer denoting the length of the character string to be transmitted. If *length* is positive, it specifies the length in words; if *length* is negative, it specifies the length in bytes. Note that if *length* exceeds the configured record length of the device, successive records will be written.

*control*                      *integer by value (required)*  
 An integer representing a carriage-control code as shown in figure 2-3.

## CONDITION CODES

CCE	Request granted.
CCG	End-of-data was encountered.
CCL	Request denied because of input/output error. Further error analysis through the FCHECK intrinsic is not possible.

## TEXT DISCUSSION

Page 4-16.

# PRINTFILEINFO

INTRINSIC NUMBER 21

Prints a file information display on the job/session list device.

```
IV  
PRINTFILEINFO(fnum);
```

From SPL (only), a secondary entry point is provided that allows the PRINTFILEINFO intrinsic to be called in the following format:

```
IV  
PRINT'FILE'INFO(fnum);
```

The PRINTFILEINFO intrinsic causes MPE to print a file information display on the standard list device in one of two formats (See Section X).

## PARAMETERS

*fnum*                                      *integer by value (required)*  
A word containing the file number.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 3-43.

Prints a character string on the Operator's Console.

INTRINSIC NUMBER 66

```
LA IV IV  
PRINTOP(message,length,control);
```

The PRINTOP intrinsic transmits a string of ASCII characters from an array in your program to the Operator's Console.

## PARAMETERS

<i>message</i>	<i>logical array (required)</i> The array from which the character string is output. The character string contained in <i>message</i> is limited to 56 characters.
<i>length</i>	<i>integer by value (required)</i> An integer denoting the length of the output string to be transmitted. If <i>length</i> is positive, it specifies the length in words; if <i>length</i> is negative, it specifies the length in bytes.
<i>control</i>	<i>integer by value (required)</i> An integer representing a carriage-control code as described in figure 2-3.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because of a physical input/output error. Further error analysis through the FCHECK intrinsic is not possible.

## TEXT DISCUSSION

Page 4-18.

# PRINTOPREPLY

INTRINSIC NUMBER 67 Prints a character string on the Operator's Console and solicits a reply.

```
I LA IV IV LA IV  
lgth:=PRINTOPREPLY(message,length,control,reply,expectedl);
```

The PRINTOPREPLY intrinsic transmits a string of ASCII characters from an array in your program to the Operator's Console and solicits a reply.

## FUNCTIONAL RETURN

This intrinsic returns a positive integer indicating the length of the reply from the console operator. This length represents a word count if *expectedl* is positive or a byte count if *expectedl* is negative.

If *expectedl* is zero, then the PRINTOPREPLY intrinsic behaves like PRINTOP and does not solicit a reply. In this case, the value returned by PRINTOPREPLY is zero.

If an error occurs, the value returned is zero.

The parameter *length* may be zero, in which case only the standard message prefix is written on the Operator's Console. If both *length* and *expectedl* are zero, then a CCL condition code is returned.

## PARAMETERS

<i>message</i>	<i>logical array (required)</i> The array from which the characters are output to the Operator's Console.
<i>length</i>	<i>integer by value (required)</i> An integer denoting the length of the output string to be transmitted. If <i>length</i> is positive, it specifies the length in words; if <i>length</i> is negative, it specifies the length in bytes. This parameter should never specify a message length of more than 50 bytes.
<i>control</i>	<i>integer by value (required)</i> An integer representing a carriage-control code as described in figure 2-3.
<i>reply</i>	<i>logical array (required)</i> The array into which the input characters are read from the Operator's Console.
<i>expectedl</i>	<i>integer by value (required)</i> An integer specifying the maximum length of the message to be read into the array <i>reply</i> . If <i>expectedl</i> is positive, it signifies a word count; if it is negative, it signifies a byte count. This parameter should never specify a reply length of more than 31 bytes.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because a physical input/output error occurred. Further error analysis through the FCHECK intrinsic is not possible.

## TEXT DISCUSSION

Page 4-18.

# PROCTIME

INTRINSIC NUMBER 42

Returns a process' accumulated central processor time.

```
D  
time:=PROCTIME;
```

The PROCTIME intrinsic is used to obtain the amount of CPU time, in milliseconds, that a process has accumulated. This is the basis on which CPU time is charged. (See the :REPORT command in the *MPE Commands Reference Manual*.)

## FUNCTIONAL RETURN

This intrinsic returns a double integer value which shows the number of milliseconds that the process has been running.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-44.

Accepts input from paper tapes which do not contain X-OFF control characters

INTRINSIC NUMBER 191

```
IV IV  
PTAPE(filenum1,filenum2);
```

You can read data programmatically from paper tapes not containing the X-OFF control character, or from tapes being read through terminals not recognizing this character, by using the PTAPE intrinsic. PTAPE deletes the characters as the tape is read through a terminal which does not recognize these characters.

Tape input terminates when a CONTROL Y (Y<sup>C</sup>) character is encountered, returning control to you at the terminal.

Prior to calling this intrinsic, you must be sure to position the end-of-file pointer in the disc file (*filenum2*) to the proper position in the file. If you are reading more than one tape, you should specify, in the FOPEN intrinsic call that opens the disc file, the append-only *aooption* and a variable-length record format, before the first PTAPE call. In addition, you should set the end-of-file pointer to zero, if necessary, before issuing the first PTAPE intrinsic call.

Lines will be folded at 256-character intervals until a carriage-control character indicates the end of a line or until the input is terminated by the Y<sup>C</sup> character.

## PARAMETERS

*filenum1*                      *integer by value (required)*  
A word identifier specifying the file number of the user's terminal. This is the value returned by FOPEN when the terminal file was opened.

*filenum2*                      *integer by value (required)*  
A word identifier specifying the file number of the disc file to which the data is to be written.

## CONDITION CODES

CCE                              Request granted.

CCG                              Request denied because an error occurred while writing to the specified disc file.

CCL                              Request denied because the input file specified is not a terminal or does not belong to the calling process, or because insufficient resources, such as disc space or main memory, are available to satisfy the request.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION



# QUIT

INTRINSIC NUMBER 76

Aborts a process.

IV  
QUIT(*num*);

From within any process in a user program structure, you can abort the process by using the QUIT intrinsic. The QUIT intrinsic also transmits a Type 2 abort message (see Section X) to the calling process' output device, and sets the job/session in an error state. In batch jobs not containing the :CONTINUE command (see the *MPE Commands Reference Manual*), this results in job termination when the entire program finishes.

## PARAMETERS

*num*

*integer by value (required)*

Any arbitrary number. When the QUIT intrinsic is executed, *num* is transmitted as part of the resulting abort message, as follows:

```
ABORT: PROG.GROUP.ACCT. %SEG. %LOC  
PROGRAM ERROR: PROCESS QUIT. PARAM= num
```

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-20.

Aborts a process.

INTRINSIC NUMBER 61

IV  
QUITPROG(*num*);

You may abort the entire user-process structure by using the QUITPROG intrinsic. This intrinsic destroys all sons of the job/session main process. The job/session main process is set in the error state. In batch jobs not containing the :CONTINUE command (see the *MPE Commands Reference Manual*), this terminates the job.

An abort message (see Section X) is transmitted to the job/session list device.

## PARAMETERS

*num*

*integer by value (required)*

Any arbitrary number. When the QUITPROG intrinsic is executed, *num* is output as part of the abort message, as follows:

ABORT: PROG.GROUP.ACCT. %SEG. %LOC  
PROGRAM ERROR: PROCESS QUIT. PARAM=*num*

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-20.

# READ

INTRINSIC NUMBER 64

Reads an ASCII string from an input device.

```
I      LA      IV  
lgth:=READ(message,expectedl);
```

The READ intrinsic reads a string of ASCII characters from a job/session input device into an array in your program. This intrinsic is similar to issuing an FREAD intrinsic call against the file \$STDIN. The READ intrinsic is limited in its usefulness, however, in that the full capability of the file system is not available to a user of this intrinsic. For example, :FILE commands are not allowed and certain file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not available to normal users of the READ intrinsic.

Basic line editing such as cancellation of lines and backspacing are performed automatically by the input/output driver. If the input device is a terminal and it is in full-duplex mode and the echo facility is on, or if the terminal is in half-duplex mode, the characters read are printed.

## FUNCTIONAL RETURN

This intrinsic returns a positive value representing the length of the ASCII string which was read. If *expectedl* is positive, this length specifies words; if *expectedl* is negative, length specifies bytes.

## PARAMETERS

<i>message</i>	<i>logical array (required)</i> The array into which the ASCII characters are read.
<i>expectedl</i>	<i>integer by value (required)</i> An integer specifying the maximum length of the array <i>message</i> . If <i>expectedl</i> is positive, this specifies the length in words; if <i>expectedl</i> is negative, this specifies the length in bytes. When the record is read, the first <i>expectedl</i> characters are input. If <i>expectedl</i> equals or exceeds the size of the physical record, the entire record is transmitted.

## CONDITION CODES

CCE	Request granted.
CCG	A record with a colon in the first column, signalling the end of data, or a hardware end-of-file was encountered.
CCL	Request denied because a physical input/output error occurred. Further error analysis through the FCHECK intrinsic is not possible.

## TEXT DISCUSSION

Page 4-16.

Reads an ASCII string from an input device.

INTRINSIC NUMBER 64

```
I      LA      IV
lgth:=READX(message,expectedl);
```



The READX intrinsic reads a string of ASCII characters from a job/session input device into an array into your program. This intrinsic is similar to issuing an FREAD intrinsic call against the file \$STDINX. The READX intrinsic is limited in its usefulness, however, in that the full capability of the file system is not available to a user of this intrinsic. For example, :FILE commands are not allowed and certain file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not available to normal users of the READX intrinsic.

Basic line editing such as cancellation of lines and backspacing are performed automatically by the input/output driver. If the input device is a terminal and it is in full-duplex mode and the echo facility is on, or if the terminal is in half-duplex mode, the characters read are printed.

## FUNCTIONAL RETURN

This intrinsic returns a positive value representing the length of the ASCII string which was read. If *expectedl* is positive, this length specifies words; if *expectedl* is negative, length specifies bytes.

## PARAMETERS

<i>message</i>	<i>logical array (required)</i> The array into which the ASCII characters are read.
<i>expectedl</i>	<i>integer by value (required)</i> An integer specifying the maximum length of the array <i>message</i> . If <i>expectedl</i> is positive, this specifies the length in words; if <i>expectedl</i> is negative, this specifies the length in bytes. When the record is read, the first <i>expedtedl</i> characters are input. If <i>expectedl</i> equals or exceeds the size of the physical record, the entire record is transmitted.

## CONDITION CODES

CCE	Request granted.
CCG	An :EOD, :EOF:, or in a job, :EOJ, :JOB, or :DATA command was encountered.
CCL	Request denied because a physical input/output error occurred. Further error analysis through the FCHECK intrinsic is not possible.

## TEXT DISCUSSION

Page 4-16.

# RECEIVEMAIL

INTRINSIC NUMBER 108

Receives mail from another process.

```
L          IV  LA  LV
status:=RECEIVEMAIL(pin,location,waitflag);
```

A process collects mail transmitted to it by its father or a son by using the RECEIVEMAIL intrinsic. If the mailbox for the receiving process is empty, the action taken depends on the *waitflag* parameter specified in the RECEIVEMAIL intrinsic call. If the mailbox currently is being used by other intrinsics, the RECEIVEMAIL waits until the mailbox is free before accessing it.

## FUNCTIONAL RETURN

This intrinsic returns one of the following mailbox status codes:

Status Returned	Meaning
0	The mailbox was empty and the <i>waitflag</i> parameter was FALSE.
1	No message was collected because the mailbox contained outgoing mail from the receiving process.
2	The message was collected successfully.
3	An error occurred because an invalid <i>pin</i> was specified or a bounds check failed.
4	The request was denied because <i>waitflag</i> specified that the receiving process wait for mail if the mailbox is empty, but the other process sharing the mailbox is already suspended, waiting for mail. If <i>both</i> processes were suspended, neither could activate the other, and they may be deadlocked.

## PARAMETERS

<i>pin</i>	<i>integer by value (required)</i> An integer specifying the process sending the mail. If a son process is specified, the integer is the Process Identification Number (PIN) of that process. If a father process is specified, the integer is zero.
<i>location</i>	<i>logical array (required)</i> The array (buffer) in the stack where the message is to be written.
<i>waitflag</i>	<i>logical by value (required)</i> A word specifying the action to be taken if the mailbox is empty: TRUE = Wait until incoming mail is ready for collection. (Bit 15 = 1) FALSE = Return immediately to the calling process. (Bit 15 = 0)

## CONDITION CODES

CCE	Request granted. The mail was collected.
CCG	Request denied because of an illegal <i>pin</i> parameter. The value 3 is returned to RECEIVEMAIL.
CCL	Request denied because the bounds check revealed that the <i>location</i> parameter did not define a legal stack address (the value 3 is returned to RECEIVEMAIL) or because both sending and receiving processes would be awaiting incoming mail (deadlock). The value 4 is returned to RECEIVEMAIL.

## SPECIAL CONSIDERATIONS

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-12.

# RESETCONTROL

INTRINSIC NUMBER 55

Resets terminal to accept CONTROL-Y signal.

```
RESETCONTROL;
```

To reset the terminal so that a CONTROL-Y signal can be accepted, the RESETCONTROL intrinsic is used. To take effect, this intrinsic must be called after the trap procedure is entered.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because the trap procedure was not invoked.

## TEXT DISCUSSION

Page 4-40.

Searches an array for a specified entry or name.

INTRINSIC NUMBER 70

```
I      BA  IV BP BP  0-V  
entryno:=SEARCH(target,length,dict,defn);
```

The SEARCH intrinsic searches a specially-formatted array, consisting of sequential entries, for a specified name. A simple linear search is performed, with the specified name compared against each entry of the specially-formatted array. Because the search is linear, the most frequently used name byte arrays should appear at the beginning of the array to insure efficient searching.

## FUNCTIONAL RETURN

This intrinsic searches *dict* for a word matching *target*, and returns the corresponding entry number to the user's program. If the name specified in *target* is not found, a zero is returned.

## PARAMETERS

<i>target</i>	<i>byte array (required)</i> Contains the name for which the search is to be performed.
<i>length</i>	<i>integer by value (required)</i> An integer specifying the length, in bytes, of the byte array <i>target</i> .
<i>dict</i>	<i>byte array (required)</i> The specially-formatted array which is to be searched for <i>target</i> .
<i>defn</i>	<i>byte pointer (optional)</i> A word to which is returned the address of the definition portion of the entry sought in the array. <i>Default: If defn is omitted, the address is not returned.</i>

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-3.



# SENDMAIL

INTRINSIC NUMBER 107

Sends mail to another process.

```
L      IV  IV  LA  LV  
status:=SENDMAIL(pin,count,location,waitflag);
```

A process sends mail to its father or sons by using the SENDMAIL intrinsic. If the mailbox for the receiving process contains a message sent previously by the calling process but not collected by the receiving process, the action taken depends on the *waitflag* parameter specified in the SENDMAIL intrinsic call. If the mailbox currently is being used by other intrinsics, the SENDMAIL intrinsic waits until the mailbox is free and then sends the mail.

## FUNCTIONAL RETURN

This intrinsic returns one of the following status codes:

Status Returned	Meaning
0	The mail was transmitted successfully. The mailbox contained no previous mail.
1	The mail was transmitted successfully. The mailbox contained mail sent previously that was overwritten by the new mail, or contained previous incoming/outgoing mail that was cleared.
2	The mail was not transmitted successfully because the mailbox contained incoming mail to be collected by the sending process (regardless of the <i>waitflag</i> setting).
3	An error occurred because an illegal <i>pin</i> parameter was specified, or a bounds check failed.
4	An illegal wait request would have produced a deadlock.
5	The request was denied because <i>count</i> exceeded the maximum mailbox size allowed by the system.
6	The request was denied because storage resources for the mail data segment were not available.

## PARAMETERS

*pin*

*integer by value (required)*

An integer specifying the process to receive the mail. If a son process is specified, the integer is the Process Identification Number (PIN) of that process. If a father process is specified, the integer is zero.

<i>count</i>	<i>integer by value (required)</i> An integer specifying the length of the message, in words, transmitted from the sending process' stack. If zero is specified, SENDMAIL empties the mailbox of any incoming or outgoing mail.
<i>location</i>	<i>logical array (required)</i> The array (buffer) in the stack containing the message.
<i>waitflag</i>	<i>logical by value (required)</i> A word specifying (in bit 15) the action to be taken if the mailbox contains mail sent previously: TRUE = Wait until the receiving process collects the previous mail before sending current mail. (Bit 15 = 1) FALSE = Cancel (overwrite) any mail sent previously with the current mail. (Bit 15 = 0).

## CONDITION CODES

CCE	Request granted. The mail was sent.
CCG	Request denied because of an illegal <i>count</i> parameter (the value 5 is returned to SENDMAIL), or an illegal <i>pin</i> parameter was specified (the value 3 is returned to SENDMAIL), or storage for the mail data segment was not available (the value 6 is returned to SENDMAIL).
CCL	Request denied because the bounds check revealed that the <i>count</i> or <i>location</i> parameters did not define a legal stack area (the value 3 is returned to SENDMAIL), or both processes are waiting to send mail (the value 4 is returned to SENDMAIL).

## SPECIAL CONSIDERATIONS

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-11.

# SETJCW

INTRINSIC NUMBER 72

Sets bits in the job control word.

LV  
SETJCW(*word*);

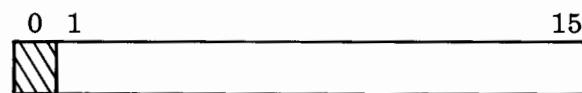
You can establish the bit contents of the job control word with the SETJCW intrinsic.

## PARAMETERS

*word*

*logical by value (required)*

A 16-bit word whose contents are established by the user for inter-process communication. The form is



Bit zero is reserved for MPE and should be set to 0. If you set this bit to 1, the system will output the following message when the user program terminates, either normally or due to an error:

ERR 2 (ABNORMAL PROGRAM TERMINATION)

A batch job is terminated unless the :CONTINUE command is used. (See the *MPE Commands Reference Manual*). Bits 1 through 15 may be used for any purpose.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-45.

Suspends a process.

INTRINSIC NUMBER 103

```
LV IV 0-V  
SUSPEND(susp,rin);
```

A process can suspend itself with the SUSPEND intrinsic. When this intrinsic is executed, the process relinquishes its access to the central processor unit until reactivated by an ACTIVATE intrinsic call. When a process suspends itself, it must specify the anticipated source of this ACTIVATE call (its father or a son process). When the process is reactivated, it begins execution with the instruction immediately following the SUSPEND call.

## PARAMETERS

*susp*

*logical by value (required)*

A word whose 14th and 15th bits specify the anticipated source of the call that later will reactivate the process. For processes run by users with only the Process Handling Capability, at least one of these bits must be set to 1.

If bit 15 = 1, the process expects to be activated by its father.

If bit 14 = 1, the process expects to be activated by one of its sons.

If both of these bits = 1, the suspended process can be activated by either father or sons.

*rin*

*integer by value (optional)*

An integer specifying a Resource Identification Number (RIN). If *rin* is specified, it represents a local RIN that is locked by the process but that will be released when this process is suspended. This facility can be used to synchronize processes within the same job.

*Default: If omitted, no RIN is unlocked when the process suspends.*

## CONDITION CODES

CCE

Request granted.

CCL

Request denied because the *susp* parameter is not valid, the specified RIN is not owned by this process, or the specified RIN was not locked.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-8.

# SWITCHDB

INTRINSIC NUMBER 139

Switches DB register pointer.

```
      L          LV  0-P  
logindex := SWITCHDB(index);
```

The SWITCHDB intrinsic changes the DB register so that it points to the base of an extra data segment instead of the base of the stack.

## FUNCTIONAL RETURN

This intrinsic returns the logical index of the data segment indicated by the previous DB register setting, thus allowing you to restore this setting later. If the previous DB setting indicated the stack, zero is returned.

## PARAMETERS

*index* *logical by value (required)*  
Specifies the logical index of the data segment to which the DB register is to be switched, as obtained through the GETDSEG intrinsic call. MPE checks the value specified for *index* to insure that the process has acquired access to this segment previously. For an extra data segment, *index* must be a positive, non-zero integer. To switch back to the stack segment, *index* must be zero.

## CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an illegal data segment was specified.

## SPECIAL CONSIDERATIONS

Must be running in Privileged Mode.

## TEXT DISCUSSION

Page 9-5.

# TERMINATE

Terminates a process.

INTRINSIC NUMBER 60

**TERMINATE;**

A process and all of its descendants, including any extra data segments belonging to them, can be deleted by using the **TERMINATE** intrinsic.

All files still open by the process are closed and assigned the same disposition they had when opened.

## CONDITION CODES

The process that calls this intrinsic is terminated and *no* return is made.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-20.



# TIMER

INTRINSIC NUMBER 40

Returns system timer.

```
D  
count:=TIMER;
```

A 31-bit logical quantity representing the current system timer and overflow count can be returned to your program with the `TIMER` intrinsic.

The resolution of the system timer is one millisecond. Thus, readings taken within a one-millisecond period may be identical.

## FUNCTIONAL RETURN

This intrinsic returns a 31-bit logical quantity representing the actual millisecond count since the last system cold load.

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-42.







Unlocks a local RIN.

INTRINSIC NUMBER 33

LV  
UNLOCKLOCIN(*rinum*):

The UNLOCKLOCIN intrinsic unlocks a local Resource Identification Number (RIN) that has been locked by the LOCKLOCIN intrinsic.

## PARAMETERS

<i>rinum</i>	<i>logical by value (required)</i> A word supplying the locked RIN, designated by an integer from 1 to the value specified in the <i>rincount</i> parameter of the GETLOCKRIN intrinsic call.
--------------	--

## CONDITION CODES

CCE	Request granted.
CCG	Request denied because the RIN specified is not locked by the calling process.
CCL	Request denied because the specified RIN is not allocated to this process.

## TEXT DISCUSSION

Page 6-8.

# WHO

INTRINSIC NUMBER 69

Returns user attributes.

```
L      D  D  BA  BA  BA  BA  L  0-V  
WHO(mode,capability,lattr,usern,groupn,acctn,homen,termn);
```

The WHO intrinsic supplies the access mode and attributes of the user running the program.

## PARAMETERS

*mode*

*logical (optional)*

A word to which the current user's access mode is returned. In this word, the bits will have the following meanings:

Bit (15:1)

1 = The job/session input file and job/session list file form an *interactive* pair. A dialog can be established between a program, displaying information on the list device, and a person responding through the input device.

0 = The job/session input file and job/session list file are *not* interactive.

Bit (14:1)

1 = The job/session input file and job/session list file form a *duplicative* pair. Images on the input device are duplicated automatically on the list device.

0 = The job/session input file and job/session list file are *not* duplicative.

Bits (12:2)

01 = The user is accessing the system through a session.

10 = The user is accessing the system through a job.

Bits (0:12) — Reserved for MPE. The WHO intrinsic sets these bits to zero.

*Default: The user's access mode is not returned.*

*capability*

*double (optional)*

A double word to which the user's file access, user, and capability class attributes are returned. In the first word, possession of the following file access and user attributes is indicated by the corresponding bit being *on* (equal to 1).

File access  
attributes

{ Bit (15:1) = Ability to save files (declare them permanent) (SF).

{ Bit (14:1) = Ability to acquire non-sharable devices (ND).

Bits (6:8) = Reserved for MPE. The WHO intrinsic sets these bits to zero.

User

Attributes

{ Bit (5:1) = System supervisor (OP).

{ Bit (4:1) = Diagnostician (DI).

{ Bit (3:1) = Group librarian (GL).

{ Bit (2:1) = Account librarian (AL).

{ Bit (1:1) = Account manager (AM).

{ Bit (0:1) = System manager (SM).

In the second word, possession of the user's capability-class attributes is indicated by the corresponding bit being *on* (equal to 1).

Bit (15:1) = Process handling (PH).

Bit (14:1) = Extra data segments (DS).

Bit (13:1) = Reserved for MPE. The WHO intrinsic sets this bit to zero.

Bit (12:1) = Exclusive simultaneous use of more than one system resource (Multiple RIN's) (MR).

Bits (10:2) = Reserved for MPE. The WHO intrinsic sets these bits to zero.

Bit (9:1) = Privileged mode operation (PM).

Bit (8:1) = Interactive (session) access (IA).

Bit (7:1) = Batch (job) access (BA).

Bits (0:7) = Reserved for MPE. The WHO intrinsic sets these bits to zero.

*Default: The user's file access, user, and capability-class attributes are not returned.*

<i>lattr</i>	<p><i>double (optional)</i></p> <p>A double word to which is returned the local attributes of the user, as defined by a user with the Account Manager attribute.</p> <p><i>Default: The user's local attributes are not returned.</i></p>
<i>usern</i>	<p><i>byte array (optional)</i></p> <p>An 8-byte array to which the user's name is returned.</p> <p><i>Default: The user's name is not returned.</i></p>
<i>groupn</i>	<p><i>byte array (optional)</i></p> <p>An 8-byte array to which the name of the user's log-on group is returned.</p> <p><i>Default: The user's log-on group is not returned.</i></p>
<i>acctn</i>	<p><i>byte array (optional)</i></p> <p>An 8-byte array to which the name of the user's log-on account is returned.</p> <p><i>Default: The user's log-on account is not returned.</i></p>
<i>homen</i>	<p><i>byte array (optional)</i></p> <p>An 8-byte array to which the name of the user's home group is returned. If a home group was not assigned, this array is filled with blanks.</p> <p><i>Default: This information is not returned.</i></p>
<i>termn</i>	<p><i>logical (optional)</i></p> <p>A word to which the logical device number of the job/session input device is returned.</p> <p><i>Default: The logical device number is not returned.</i></p>

# WHO

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-10.

Enables the user-written software arithmetic trap.

INTRINSIC NUMBER 50

```

          IV   IV   I   I
XARITRAP(mask,plabel,oldmask,oldplabel);
    
```

The XARITRAP intrinsic enables you to replace the trap handler in MPE with your own trap handler routine.

## PARAMETERS

*mask*

*integer by value (required)*

A word mask that selects which hardware traps will invoke the software trap, and which will not. Only the 14 rightmost bits of the word forming the mask are used. The setting of the other bits is not significant, but it is recommended that they be set to zero. Thus, octal values up to %37777 are allowed for this parameter.

If a bit is on (= 1), the corresponding hardware trap activates the software trap. If a bit is off (= 0), the corresponding hardware trap does not activate the software trap. If all bits are set to zero, the software trap is disabled.

Bit	Hardware Error Trap
15	Floating point divide by zero.
14	Integer divide by zero.
13	Floating point underflow.
12	Floating point overflow.
11	Integer overflow.
10	Extended precision overflow.
9	Extended precision underflow.
8	Extended precision divide by zero.
7	Decimal overflow.
6	Invalid ASCII digit.
5	Invalid decimal digit.
4	Invalid source word count.
3	Invalid decimal operand length.
2	Decimal divide by zero.



*plabel*

*integer by value (required)*

The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disabled. The external-type label of the procedure, which resides in the segment transfer table of the procedure's code segment, is passed as a parameter (in SPL) by placing a @ before the procedure name.

*oldmask*

*integer by value (required)*

A word in which the value of the previous *mask* is returned to the user's program.

# XARITRAP

*oldplabel*

*integer (required)*

A word in which the previous *plabel* is returned to the user's program. If no *plabel* existed previously, zero is returned.

## CONDITION CODES

CCE	Request granted. Software trap enabled.
CCG	Request granted. Software trap disabled.
CCL	Request denied because of an invalid <i>plabel</i> .

### NOTE

The validity of a trap procedure, specified by the external-type label of the user's trap procedure (*plabel*), depends on the code domain of the caller's code and executing mode (privileged or non-privileged), and on the code domain of the *plabel* and the mode (privileged or non-privileged). The code domains are:

PROG	(User Program)
GSL	(Group SL)
PSL	(Public SL)
SSL	(System SL, non-MPE segments)
MPSSL	(System SL, MPE segments)

If, when a trap procedure is being enabled, the code of the caller is

1. Non-privileged in PROG, GSL, or PSL, *plabel* must be non-privileged in PROG, GSL, or PSL.
2. Privileged in PROG, GSL, or PSL, *plabel* may be privileged or non-privileged in PROG, GSL, or PSL.
3. Privileged or non-privileged in SSL, *plabel* may be in any non-MPSSL segment.

## TEXT DISCUSSION

Page 4-32.

Enables or disables the CONTROL-Y trap.

INTRINSIC NUMBER 54

```

      IV      I
XCONTRAP(plabel,oldplabel);
```

When a session is initiated, the CONTROL-Y trap is disabled. The XCONTRAP intrinsic enables this trap. This intrinsic takes effect on the file \$STDINX, and also on \$STDIN (when \$STDIN is defined as a terminal).

## PARAMETERS

<i>plabel</i>	<i>integer by value (required)</i> The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disabled.
<i>oldplabel</i>	A word in which the previous <i>plabel</i> is returned to the user's program. If no <i>plabel</i> existed previously, zero is returned.

## CONDITION CODES

CCE	Request granted. Trap enabled.
CCG	Request granted. Trap disabled.
CCL	Request denied because of illegal <i>plabel</i> , or because \$STDIN is not defined as a terminal.

### NOTE

The validity of a trap procedure, specified by the external-type label of the user's trap procedure (*plabel*), depends on the code domain of the caller's code and executing mode (privileged or non-privileged), and on the code domain of the *plabel* and the mode (privileged or non-privileged). The code domains are:

PROG	(User program)
GSL	(Group SL)
PSL	(Public SL)
SSL	(System SL, non-MPE segments)
MPSSL	(System SL, MPE segments)

If, when a trap procedure is being enabled, the code of the caller is

1. Non-privileged in PROG, GSL, or PSL, *plabel* must be non-privileged in PROG, GSL, or PSL.
2. Privileged in PROG, GSL, or PSL, *plabel* may be privileged or non-privileged in PROG, GSL, or PSL.
3. Privileged or non-privileged in SSL, *plabel* may be in any non-MPSSL segment.

## TEXT DISCUSSION



# XLIBTRAP

INTRINSIC NUMBER 52

Enables or disables the software library trap.

```
IV      I  
XLIBTRAP(plabel,oldplabel);
```

When a program begins execution, the software library trap is disabled automatically. You can enable (or subsequently disable) this trap with the XLIBTRAP intrinsic call.

## PARAMETERS

<i>plabel</i>	<i>integer by value (required)</i> The external-type label of the user's trap procedure. If the value of this entry is 0, the trap is disabled.
<i>oldplabel</i>	<i>integer (required)</i> A word in which the previous <i>plabel</i> is returned to the user's program. If no <i>plabel</i> existed previously, zero is returned.

## CONDITION CODES

CCE	Request granted. Trap enabled.
CCG	Request granted. Trap disabled.
CCL	Request denied because of an illegal <i>plabel</i> .

## TEXT DISCUSSION

Page 4-35.

Enables or disables the system trap.

INTRINSIC NUMBER 53

```
IV      I  
XSYSTRAP(plabel,oldplabel);
```

When a program begins execution, the system trap is disabled automatically. When enabled by the XSYSTRAP intrinsic, and subsequently activated by an error, the trap transfers control to a trap procedure.

You can enable (or subsequently disable) the system trap by using the XSYSTRAP intrinsic.

## PARAMETERS

<i>plabel</i>	<i>integer by value (required)</i> The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disabled.
<i>oldplabel</i>	<i>integer (required)</i> A word in which the previous <i>plabel</i> is returned to the user's program. If no <i>plabel</i> existed previously, zero is returned.

## CONDITION CODES

CCE	Request granted. Trap enabled.
CCG	Request granted. Trap disabled.
CCL	Request denied because of an illegal <i>plabel</i> .

## TEXT DISCUSSION

Page 4-36.

# ZSIZE

INTRINSIC NUMBER 136

Changes size of Z to DB area.

I	IV
<code>actsize:=ZSIZE(size);</code>	

The ZSIZE intrinsic alters the size of the current Z to DB area by adjusting the register offset of the Z address from the DB address (Z to DB).

The ZSIZE intrinsic moves the Z address forward (expansion) or backward (contraction). If the Z to DB area size requested exceeds the maximum size permitted for the Z to DL (stack) area, only the maximum size allowed is granted.

All changes to the Z to DB area are made in increments or decrements of 128 words, and hence the size actually granted may differ from the size requested.

## FUNCTIONAL RETURN

This intrinsic returns the size actually granted.

## PARAMETERS

<i>size</i>	<i>integer by value (required)</i> An integer value greater than or equal to zero that specifies the desired register offset (in words) for Z to DB.
-------------	---

## CONDITION CODES

CCE	Request granted.
CCG	The requested size exceeded the maximum limits of the Z to DL (stack) area. The maximum limit is granted, and this value is returned to the calling process as the value of ZSIZE.
CCL	An illegal <i>size</i> parameter, less than $(S - DB) + 64$ words, was specified. This minimum value is assigned by default.

## TEXT DISCUSSION

Page 4-27.

# ACCESSING AND ALTERING FILES

SECTION

III

By using MPE file system intrinsics, you can perform the following operations on files:

- Open files with FOPEN. See page 3-24.
- Request access and status information about a file with FGETINFO. See page 3-63.
- Request file error information with FCHECK. See page 3-65.
- Read records (or a portion of a record) from a file with FREAD (see page 3-43) and FREADDIR (see page 3-47).
- Write a record (or a portion of a record) to a file with FWRITE (see page 3-46) and FWRITEDIR (see page 3-49).
- Move a specific record from a file into a buffer preparatory to reading the record to the stack with FREADSEEK. See page 3-49.
- Initiate completion operations for an I/O operation with the IOWAIT intrinsic. See page 3-57).
- Read a user-defined label from a disc file with FREADLABEL. See page 3-63.
- Write a user-defined label onto a disc file with FWRITELABEL. See page 3-59.
- Update a record on a disc file with FUPDATE. See page 3-54.
- Space forward or backward on a disc or tape file with FSPACE. See page 3-75.
- Reset the logical record pointer to any logical record in a fixed-length record disc file with FPOINT. See page 3-77.
- Perform control operations on a file (or the device on which the file resides) with FCONTROL. See page 3-76.
- Activate and deactivate access mode options with FSETMODE. See page 3-77.
- Rename an open disc file with FRENAME. See page 3-77.
- Determine if an input file and a list file are *interactive* or *duplicative* with FRELATE. See page 3-78.
- Lock a file so that no two processes can share the file with FLOCK intrinsic (see page 3-52) or unlock the file with FUNLOCK. (see page 3-54).
- Close a file with FCLOSE. See page 3-35.

## FILE MANAGEMENT SYSTEM

The File Management System provides a uniform method of directing input and output of information. It handles various input/output applications, such as the transfer of information to and from user processes, compilers, and data management subsystems.

MPE treats each set of input or output information as a set of records arranged into a file. When a file is created, MPE allocates (or requests the operator to allocate) a device for its storage. Input read from a hardware device such as a card reader is accepted directly from that device. Similarly, output from an executing process is transmitted directly to the required device (such as a line printer).

You reference a file by the *file name* assigned to the file when it is created. When you reference an existing file, MPE determines the device or disc address where the file is stored and accesses the file for you. Throughout its existence, the file remains device independent.

The MPE File Management System allocates devices for the storage of files on the basis of specifications from you. For example, you can request a device by generic type name, or “device class”, as configured by the System Manager (such as any magnetic tape unit or line printer), or by the logical device number that refers to a specific device. Logical device numbers are related to all devices when MPE is configured.

## FILE CHARACTERISTICS

A file can contain MPE commands, programs, or data, or any combination of these elements, written in ASCII or binary code.

Within a file, information is organized as a set of *logical records*, which are fields of data input, processed, and output as a unit. A logical record is the smallest data grouping directly addressable by you. Its length is specified by you when you create or define the file. A *physical record* is one or more logical records, and is the basic unit that can be transferred between main memory and the device on which the file resides. Physical records can be longer, shorter, or the same size as the logical records in the file. In files on disc or magnetic tape, physical records are organized as *blocks* that always contain an integral number of logical records. On unit-record devices, the size of the physical records in a file is determined by the device. For example, each physical record read from a card reader consists of one punched card; each physical record written to a line printer consists of one line of print. On unit-record devices in multi-record mode, *logical* records are *not* blocked. For example, on punched cards, each logical record is assumed to begin at the first column of the card. Thus, to read a single 100-character logical record, the multi-record *option* must be specified in the FOPEN call to open the file. Then the record would be read as 80 characters from one card (physical record) and 20 characters from the next card. The next logical record is assumed to begin in the first column of the third card encountered. Similarly, when a file is transmitted to a printer, each logical record appears as one line of print (physical record), left-justified. If the logical record is longer than the print line, and the multirecord *option* is specified, the remaining information is continued on the next line, also left-justified.

On disc, files can be organized to permit either sequential or direct access. Direct-access files contain logical records of fixed or undefined length. Sequential-access files contain logical records of fixed or varying length.

MPE manages each file on disc as a set of *extents*. Each extent is an integral number of contiguously-located disc sectors. All extents (except possibly the last) are of equal size. When a file is opened, the first extent (containing at least one sector for file label information) is allocated immediately. Other extents, up to a maximum of 32, are allocated as needed. Alternatively, you can request immediate allocation of more than one extent when the file is opened. The size of each extent is determined as noted in the discussion of the :FILE command parameter *numextents* in the *MPE Commands Reference Manual*.

Each extent of a disc file must be totally contained within a given disc device. In addition, the extents that comprise a disc file are restricted to disc devices that are members of the device class specified when the file was created. Normally, each extent is arbitrarily assigned to any device with this class. Alternatively, each extent may be restricted to a specific disc device. The method of extent allocation is determined by the device class specification of the FOPEN intrinsic when the file is created.

When a file contains only fixed-length records, you can calculate the effective disc space (in sectors) required by the file with the following formula. This formula applies to every type of disc drive supported by MPE; however, it does not include disc space required by the label.

$$S = \left\lceil \frac{N}{B} \right\rceil \times \left\lceil \frac{LxB}{128} \right\rceil$$

- S* The number of disc sectors required by the file.
- N* The number of fixed-length logical records in the file.
- B* The number of logical records in a block (blocking factor).
- L* The length of each logical record, in 16-bit *words*.



The constant 128 is the number of words in a sector. Each expression within brackets is evaluated separately and rounded upward before the final multiplication takes place. (A notation in the form  $\lceil x \rceil$  means “ceiling (x)” — the smallest integer greater than or equal to x.)

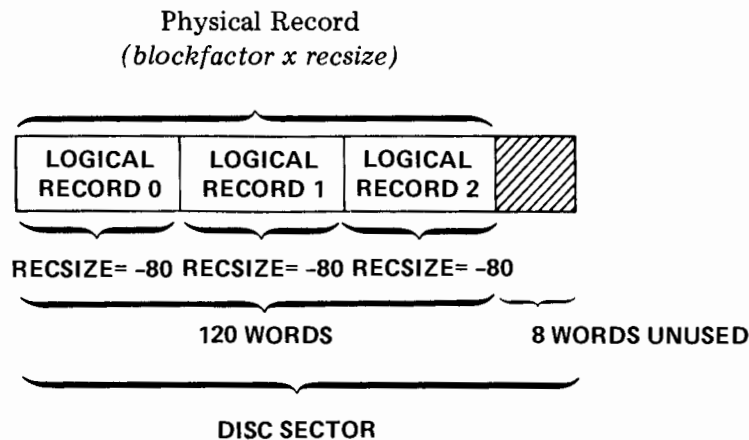
For example, the effective disc space for a file containing 100 records of 50 words each, with a blocking factor of 3, would be calculated as follows:

$$\begin{aligned} S &= \left\lceil \frac{100}{3} \right\rceil \times \left\lceil \frac{50 \times 3}{128} \right\rceil \\ &= [34] \times [2] \\ &= 68 \text{ sectors (plus 1 for the label)} \end{aligned}$$

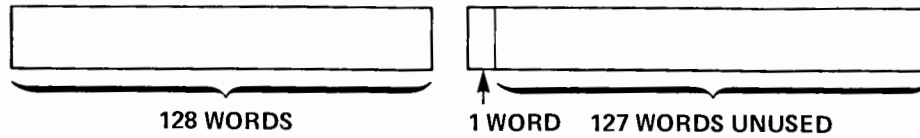
## RECORD FORMATS

A file can contain records written in one of three formats: *fixed-length*, *variable-length*, and *undefined length*. These formats are described below. (In all cases, *reclsize* is in words.)

For *fixed-length records*, physical records are blocks containing one or more logical records. The block size is determined by multiplying the block factor by the logical record size. (The block factor and logical record size are specified in the *blockfactor* and *reclsize* parameters of the FOPEN intrinsic.) On any one file, fixed-length records are all the same size. A 128-word physical record (block) containing three 80-byte, fixed-length logical records is illustrated below:



If you use a record of 129 words with a blockfactor of 1, you will waste 127 words of disc space, as follows:



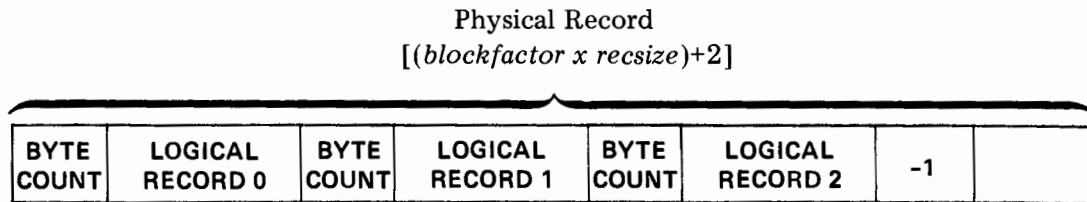
For optimum use of disc space, therefore, block size should be computed such that  $(reclsize \times blockfactor) \text{ modulo } 128 = 0$ .

For variable-length records (as for fixed-length records), physical records are blocks containing one or more logical records — but, on any one file, the record size may vary from record to record. The block size is determined by multiplying the blockfactor by the *reclsize* parameter specified in FOPEN, and adding two words (reserved for file-system use).

$$\text{Actual Blocksize} = (\text{blockfactor} \times \text{reclsize}) + 2$$

(in words)

In a block containing variable-length records, each logical record is preceded by a one-word *byte-count* showing the length of that record in bytes. The last record in the block is followed by a word containing “-1”, acting as the *block terminator*; the next logical record encountered will be the first record in the next block. The block format is:

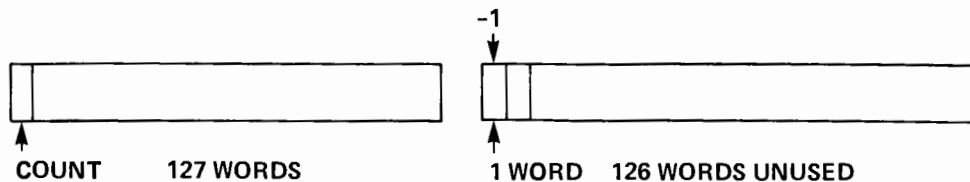


Unless block size is computed such that

$$(\text{reclsize} \times \text{blockfactor}) \text{ modulo } 128 = 0,$$

some disc space will be wasted.

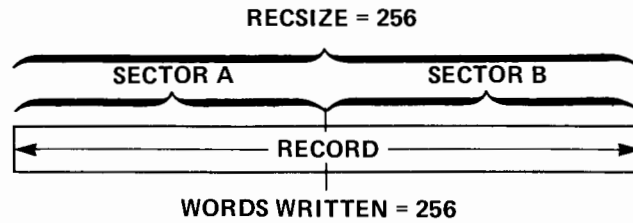
For example, if *reclsize* = 128 words and *blockfactor* = 1, 126 words of disc space will be wasted as follows:



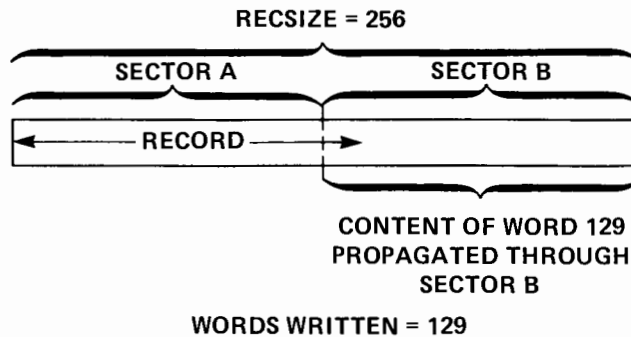
For undefined-length records, physical records and logical records are synonymous — that is, Physical record A is the same as logical record A. For records of this type, the *reclsize* parameter specified by the user denotes the size of the longest record to be transferred. The format of

undefined records written to disc, with respect to the disc sectors occupied, can be illustrated by three cases in which the user-specified *resize* is 256.

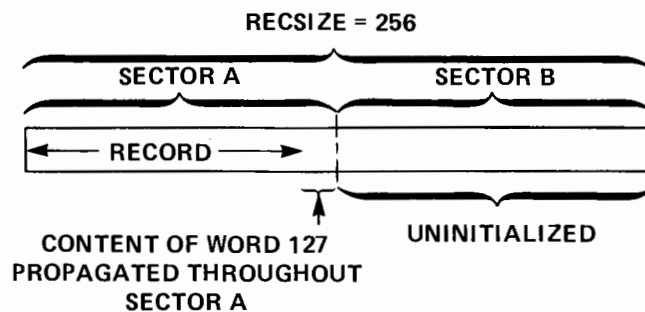
*Case 1:* You write a record 256 words long. The full record completely fills two disc sectors.



*Case 2:* You write a record 129 words long. The record written occupies all of sector A and the first word of sector B; the last word written is propagated throughout the remainder of sector B. (The rule is: if (*reclength*) modulo 128 is not zero, then the last word written is propagated through the current sector.)



*Case 3:* You write a record 127 words long. The record written occupies 127 words of Sector A; the last word of the record is propagated throughout the remainder (word 128) of Sector A. Sector B contains uninitialized data. (The rule is: any sector not written into will remain uninitialized to 0 (binary files) or blanks (ASCII files).)





## FILE DEVICE RELATIONSHIPS

Devices required by files are allocated automatically by MPE. You can specify these devices by type (such as any card reader or line printer), or by a logical device number related to a particular device (such as a specific line printer). (A unique logical device number is assigned to each device when the system is configured.) Regardless of what device a particular file resides on, when a user program asks to read that file, it references the file by its formal file designator. MPE then determines the device on which the file resides, or its disc address if applicable, and accesses it for you. When the user program writes information to a particular file to be output on a device such as a line printer, again the program refers to the file by its formal file designator. MPE then automatically allocates the required device to that file. Throughout its existence, every file remains device-independent; that is, it is always referenced by the same formal file designator regardless of where it currently resides. The user program always deals with logical records.

## NON-SHARABLE DEVICE ACCESS

The specification of a device by type when a file is opened implies a request for the initial allocation of a previously unopened device. The file system, during FOPEN, issues an allocation request to the console operator. After the operator answers, FOPEN continues execution. (The device specification is ignored when \$STDIN[X] and \$STDLIST are opened.) A job may *reallocate* an opened device by specifying the device's logical device number when the file is opened.

Multiple processes can asynchronously interleave accesses to reallocated devices. Since the file system does "anticipatory reads" on buffered input devices, multiple processes should specify Multi-Access (better) or Inhibit Buffering (NOBUF) if records must be transmitted in the same order as requested.

## FILE DOMAINS

The set of all permanent disc files in MPE is known as the *system file domain*. Within this domain, files are assigned to accounts and organized into groups under those accounts. You log on using an account and group which provides the basis for your local file references. You may be required to supply *passwords* for the account and group to log on, but thereafter (if the default MPE file security provisions are in effect) you can:

- Have unlimited access to any file within your log-on or home group. If, however, the file is protected by a *lockword*, you must know this lockword.
- Read, and execute programs residing in, any file in the public group of your account, and in the public group of the system account.

Potentially, if the MPE file security provisions at the account, group, and file levels were all suspended, and you knew all account and group names and file lockwords, you could access any permanent file in the system once you logged on. Note that once you log on, you *do not* need to know the *passwords* for other accounts and groups to access files assigned to them — you only need to know their account and group names. But, if any of these files are protected by a file lockword, you must know this lockword.

For every job or session running in the system, MPE recognizes another file domain, called the *job or session file domain*. This domain contains all temporary files opened and closed within the job or session without being saved (i.e., declared permanent). Files in these domains are deleted when the job or session terminates (if they are job/session temporary files), or when the creating program ends (if they are regular temporary files).

## FILE LABEL

MPE reads and writes file labels for files on disc during allocation of the devices on which the files reside. The format and content of file labels is presented in Appendix B.

## FILE ACCESSING

You access files through commands and intrinsic calls. Commands, described in the *MPE Commands Reference Manual*, are issued external to the user program and perform administrative functions, such as creating, deleting, or renaming a file. Files are opened (through the FOPEN intrinsic); operated on through various intrinsics that read information from them, write information to them, update them, or otherwise manipulate them; and, finally, they are closed (through the FCLOSE intrinsic).

Within a program, a file is accessed by its *formal file designator*. The formal file designator is the name by which the program recognizes the file. (In SPL, this is the name associated with it by the FOPEN intrinsic.) At the time a file is FOPENed, this formal file designator is always associated or equated with an *actual file designator*. The actual file designator is the name of the actual file to be used and the physical device upon which it resides, as recognized throughout the system by MPE. Thus, the actual file designator is an execute-time redefinition of the file specified in the program by the formal file designator. If you do not specify an actual file designator for a formal file designator, MPE uses the formal file designator for the actual file designator.

MPE recognizes actual file designators for four types of files:

- System-Defined Files
- User Pre-Defined (Back-Referenced) Files
- New Files
- Old Files

You can specify any of these designators programmatically.

### NOTE

For discussions of MPE commands referenced below, such as :JOB, :DATA, :EOJ, :EOD, :FILE, and :BUILD, see the *MPE Commands Reference Manual*.

## SYSTEM-DEFINED FILES

System-defined file designators indicate those files that MPE uniquely identifies as standard input/output devices for a job/session. They are referenced as follows:

### Actual File Designator

### Device/File Referenced

\$STDIN

A file name indicating the standard job or session input device (that from which the job or session is initiated). For a job, this is typically a card reader. For a session, this is typically a terminal. Input data images in the \$STDIN file should not contain a colon in column 1, since this indicates the end-of-data. (When data is to be delimited, this should be done through the :EOD command, which performs no other function.)

Actual File Designator	Device/File Referenced
\$STDINX	Equivalent to \$STDIN, except that MPE command images (those with a colon in column 1) encountered in a data file, are read without indicating the end of data. (However, the commands :JOB, :DATA, :EOJ, and :EOD are exceptions that always indicate the end-of-data but are otherwise ignored in this context; they are <i>never</i> read as data.)
\$STDLIST	A file name indicating the standard job or session listing device (customarily a printer for a batch job and a terminal for a session).
\$NULL	The name of a non-existent “ghost” file that is always treated as an empty file. When referenced as an input file by a program, that program receives an end-of-data indication upon each access. When referenced as an output file, the associated write request is accepted by MPE but no physical output is actually performed. Thus, \$NULL can be used to discard unneeded output from a running program.

### USER PRE-DEFINED (BACK-REFERENCED) FILES

A user pre-defined file is any file that was previously defined or re-defined in a :FILE command. In other words, it is a back-reference to that :FILE command. It is referenced by the following file designator format:

*\*formaldesignator*

*formaldesignator*      The name used in the *formaldesignator* parameter of the :FILE command.

### NEW FILES

New files are files that have not yet been created, and are being created/opened for the first time by the current program. New files can have the following actual file designators:

Actual File Designator	File Referenced
\$NEWPASS	A temporary disc file that can be automatically passed to any succeeding MPE command within the same job/session, which references it by the file name \$OLDPASS. Only one such file with this designation can exist in the job/session at any one time. (When \$NEWPASS is closed, its name is automatically changed to \$OLDPASS, and any previous file named \$OLDPASS in the job/session is deleted.)
<i>filereference</i>	Unless you specify otherwise, this is a temporary file, residing on disc, that is destroyed on termination of the creating program. If closed as a job/session temporary file, as shown later in this section, it is purged at the end of the job/session. If closed as a permanent file, it is saved until purged by you. Typically, this format consists of a file name containing up to eight alphanumeric characters, beginning with a letter, as discussed below. In addition, other elements (such as a group name, account name or lockword) can be specified.

## OLD FILES

Old files are existing named files presently in the system. They may be named by the following designators:

Actual File Designator	File Referenced
\$OLDPASS	The name of the temporary file last closed as \$NEWPASS.
<i>filereference</i>	Any other old file to which you have access. (The <i>filereference</i> format is discussed below.) It may be a job/session temporary file created in this or a previous program in the current job/session, a permanent file saved by any program, or a permanent file built (with the :BUILD command) in any job/session.

## INPUT/OUTPUT SETS

All file designators described previously can be classified as those used for input files (*Input Set*) and those used for output files (*Output Set*). These sets are defined as follows:

### Input Set

\$STDIN	The job/session input device.
\$STDINX	The job/session input device with commands allowed.
\$OLDPASS	The last \$NEWPASS file closed.
\$NULL	A constantly-empty file that will return an end-of-file indication whenever it is read.
<i>*formaldesignator</i>	A back-reference to a previously-defined file.
<i>filereference</i>	A file name, and perhaps account and group names and a lockword.

### Output Set

\$STDLIST	The job/session listing device.
\$OLDPASS	The last file passed.
\$NEWPASS	A new temporary file to be passed.
\$NULL	A constantly-empty file that returns a successful indication whenever information is written to it.
<i>*formaldesignator</i>	A back-reference to a previously-defined file.
<i>filereference</i>	A file name, and perhaps account and group names and a lockword.

## ACCESSING FILES ALREADY IN USE

When a user process attempts to access a file already being accessed by another process, the action taken by MPE depends on the current use of the file, as shown in figure 3-1.

Within a user program, the accessing and modification of files is requested through intrinsic calls. Each file referenced is first opened with the FOPEN intrinsic call. Then other operations, such as reading, writing, updating, and spacing forward or backward, can be performed on the file with other intrinsic calls. Finally, the file is closed with the FCLOSE intrinsic call, issued by the user process or (if not included in the user program) by MPE when the user process terminates.

If you are programming in SPL, you declare the intrinsics and write the intrinsic calls as you do other statements within your program. If you are programming in another language, however, such as FORTRAN, any intrinsics required are called automatically by MPE (intrinsics may be called directly from other languages and the methods are described in the manuals covering the languages).

In the FOPEN intrinsic call, you reference a particular file by its formal file designator. When the FOPEN intrinsic is executed, it returns to your program a *file number* by which the system uniquely identifies the file. The file number, rather than the file designator, is used by subsequent intrinsics in referencing the file. In an SPL program, you obtain this number through the normal conventions of the language. One such convention employs an SPL assignment statement to store the file number into a location specified by an identifier (name) which then can be used as an intrinsic call parameter to reference the file. The format of the assignment statement is discussed in the *SPL Reference manual*. Each intrinsic is declared and called as described in Section II.

The condition codes returned to your program by the file system intrinsics have the following general meanings. The specific meanings, of course, depend on the particular intrinsic and are described in Section II.

Condition Code	Meaning
CCE	The function requested by the intrinsic call was completed successfully.
CCG	MPE encountered the end of the file while servicing the request.
CCL	MPE could not service the request because an error occurred. Corrective action may be taken in some cases. (By issuing an FCHECK intrinsic call, you can have a more detailed error description transmitted to your process.) If, however, the error resulted from invalid parameters supplied by you in the intrinsic call, the error is fatal and the process is aborted, or a software error trap, if previously enabled by you, is activated (See the XSYSTRAP intrinsic).

When a file is accessed by a process running a program written in a language other than SPL, the file is generally (but not always) referenced by a file name. All intrinsic calls needed for opening, accessing, and closing the file are generated automatically by the user process, and the file name is equated with the file number used by the intrinsics to reference the file.

REQUESTED ACCESS GRANTED, UNLESS NOTED

REQUESTED ACCESS	CURRENT USE	FOPENed FOR INPUT		FOPENed FOR OUTPUT		FOPENed FOR INPUT/OUTPUT		PROGRAM FILE LOADED	BEING STORED	BEING RESTORED
		SHR	EAR	SHR	EAR	SHR	EAR			
FOPEN for Input	SHR	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted	Requested Access Granted	Error 91
	EAR	Requested Access Granted	Requested Access Granted	Error 90	Error 90	Error 90	Error 90	Error 90	Error 90	Error 91
FOPEN for Output	SHR	Requested Access Granted	Error 91	Requested Access Granted	Error 91	Requested Access Granted	Error 91	Error 91	Error 91	Error 91
	EAR	Requested Access Granted	Error 91	Error 90	Error 90	Error 90	Error 90	Error 90	Error 90	Error 91
FOPEN for Input/Output	SHR	Requested Access Granted	Input Granted	Requested Access Granted	Input Granted	Requested Access Granted	Input Granted	Input Granted	Input Granted	Error 91
	EAR	Requested Access Granted	Input Granted	Error 90	Error 90	Error 90	Error 90	Error 90	Error 90	Error 91
:RUN,CREATE		Requested Access Granted	Requested Access Granted	Error Message	Error Message	Error Message	Error Message	Requested Access Granted	Only if Loaded	Error Message
:STORE		Requested Access Granted	Requested Access Granted	Error Message	Error Message	Error Message	Error Message	Requested Access Granted	Error Message	Error Message
:RESTORE		Error Message	Error Message	Error Message	Error Message	Error Message	Error Message	Error Message	Error Message	Error Message

NOTES: 1. SHR = Share; EAR = Exclusive, allow reading.  
 2. Fully exclusive accesses cause any succeeding access (except :STORE) to fail.  
 3. Append access treated like output; Update treated like input/output.  
 4. Error 90 = Calling process requested exclusive access to a file to which another process has access.  
 5. Error 91 = Calling process requested access to a file to which another process has exclusive access.

Figure 3-1. Actions Resulting from Multiple Access of Files

When a new file is opened but not yet closed, it is always part of the job/session domain. At this time, the designator (SPL programs) or the file name (programs in other languages) assigned by you need not be unique. But when the file is saved, or closed without being deleted, MPE determines whether another file with the same designator name exists. If a name conflict occurs, a CCL condition code is returned to the user process, and the specific error is made available through the FCHECK intrinsic. When a program aborts, old files are returned to the domain in which they were found when opened; new files are deleted.

#### NOTE

All intrinsics discussed in this section, with the exception of FOPEN, FGETINFO, and FRENAME, can be called (in privileged mode) with the DB register pointing to a data segment other than the calling process' stack (split stack). All parameters referenced in any calls to these intrinsics are always accessed using the current DB-register setting.

Before a user process can read, write on, or otherwise manipulate a file, the process must initiate access to that file by opening it with the FOPEN intrinsic call (see page 3-24). This call applies to files on all devices. When the FOPEN intrinsic is executed, it returns to the user process the file number used to identify the file in subsequent intrinsic calls.

If the file is opened successfully (and the CCE condition code results), the file number returned is a positive integer ranging from 1 to 255. (Theoretically, one process may open a maximum of 255 files.) If the file cannot be opened, resulting in the CCL condition code, the file number returned is zero.

If a process issues more than one FOPEN call for the same file before it is closed, this results in multiple, logically-separate accesses of that file, and MPE returns a unique file number for each such access. Also, MPE maintains a separate logical record pointer (indicating the next sequential record to be accessed) for each such access.

In opening a file, FOPEN establishes a communication link between the file and your program by

- Allocating to your program the device on which the file resides. If the file resides on magnetic tape, FOPEN determines whether it is present in the system. (If it is not, FOPEN requests the system operator to supply the tape. Cataloging of tapes, however, is not done.) Generally, disc files can be shared concurrently among jobs and sessions. But magnetic tape and unit-record devices are allocated exclusively to the requesting job or session. For example, different processes within the same job may open and have concurrent access to files on the same magnetic tape or unit-record device; but this device cannot be accessed by another job until all accessing processes in this job have issued corresponding close requests (FCLOSE calls).
- Verifying your right to access the file under the security provisions existing at the account, group, and file levels.
- Determining that the file has not been allocated exclusively to another process (by the *exclusive* option in an FOPEN call issued by that process).

- Processing file labels (for files on disc). For new files on disc, FOPEN specifies the number of labels to be written.
- Allocating to the file the number of extents initially requested (for new disc files).
- Constructing the control blocks required by MPE for this particular access of the file. The information in these blocks is derived by merging specifications from four sources, listed below in descending order of precedence:
  1. The file label, obtainable only if the file is an *old* file on disc. This information overrides that from any other source. (Label formats are presented in Appendix B).
  2. The parameter list of a previous :FILE command referencing the same formal file designator named in this FOPEN call, if such a command was issued in this job or session. This information overrides that from the two sources listed next.
  3. The parameter list of this FOPEN intrinsic call.
  4. System default values provided by MPE (when values are not obtainable from the above three sources).

When information in one of these four sources conflicts with that in another, pre-empting takes place according to the order of precedence shown above. To determine the specifications actually taking effect, the user can call the FGETINFO intrinsic, described later in this section. Notice that certain sources do not always apply or convey all types of information. (For instance, no file label exists when a new file is opened and so all information must come from the last three sources above.)

## FILES ON NON-SHARABLE DEVICES

When a process opens a disc file, you specify whether the file is an old or new file; an old file is an existing, labeled file, and a new file implies that the file is to be created. When a process accesses a file that resides on a non-sharable device, the device's attributes may override your old/new specification. Specifically, devices used for input only (such as card readers) automatically imply *old* files; devices used for output only (such as line printers) automatically imply *new* files; serial input/output devices (such as teletype terminals and magnetic tape units) follow your old/new specifications.

When a job attempts to open an *old* file on a non-sharable device, MPE searches for the file in the Virtual Device Directory (VDD). If the file is not found, a message is transmitted to the console operator, asking him to locate the file by taking one of the following steps:

1. Indicate that the file resides on a device that is not in auto-recognition mode. No :DATA command is required — the operator simply allocates the device.
2. Make the file available on an auto-recognizing device, and allocate that device.
3. Indicate that the file does not exist on any device; the user's FOPEN request will be rejected.



When a job opens a *new* file on a non-sharable device (other than magnetic tape), the operator is not required to intervene. In these cases, the first available device is used. (A non-sharable device is considered directly available if it is not being used, or if it is being used by the requesting job and is requested by its logical device number.)

The specification of a device class when FOPEN is issued implies a request for the initial allocation of a previously unopened device. (The device parameter is ignored when \$STDIN[X] and \$STDLIST are opened.) A job may reallocate an opened device by specifying the device's logical device number when the file is opened. The FGETINFO intrinsic should be used to determine the logical device number assigned to an opened file.

When a job opens a *new* file on a magnetic tape unit, operator intervention is always required; the operator must make the tape available.

## HOW TO USE FILES

The remainder of this section explains what you can accomplish with files using the file system intrinsics. An attempt is made to show practical applications for the intrinsics, instead of merely reiterating the purpose of each intrinsic (which was discussed in Section II).

### INTERNAL OPERATIONS FOR FILE ACCESSING

Before a file can be used, it must be opened with the FOPEN intrinsic. If you are programming in SPL, you must call the FOPEN intrinsic from your program. The compilers for other languages, such as FORTRAN and COBOL, call the FOPEN intrinsic and open the file for you. In any case, however, whether called explicitly by your SPL program or called for you in a FORTRAN or COBOL program, the FOPEN intrinsic is used to open all files in a program. Several items which should be considered before using FOPEN are discussed in the following paragraphs.

For example, consider what occurs when a user coding a program in SPL performs a call to the FOPEN intrinsic to open a *new* disc file. A new disc file is a file that has not existed previously in the system. One of the fundamental things that occurs at FOPEN time is that an *access interface* is created for the file. This access interface may be an extra data segment that is created and which contains information about the file. In addition, a buffer space is allocated in this file segment to contain the number of records per block that the user has specified in the FOPEN call. The buffer space is large enough to receive a block of information from the disc.

The file segment is pointed to by an entry in the user's stack. This entry is called an *available file table* (AFT) and is part of the *process control block extension* (PCBX) in the user's stack. Upon the successful completion of an FOPEN call, an integer value is returned to the calling program. This integer value is an entry into the AFT, and the appropriate AFT entry in turn then points to the file segment that belongs to this particular file.

Figure 3-2 shows the stack and the AFT entry pointing to the file segment. The file segment contains buffers, in this case, enough room for three logical records. In the example shown in figure 3-2, each record is 80 bytes and the records are grouped into a block of three. Thus, there is enough room in the file segment to hold three logical records.

The next thing that occurs is that file space is allocated to the file. On the system disc there is a table of free space that is monitored by MPE. The file system refers to the file space table and deallocates a number of sectors for this file (the number of sectors deallocated depends on the

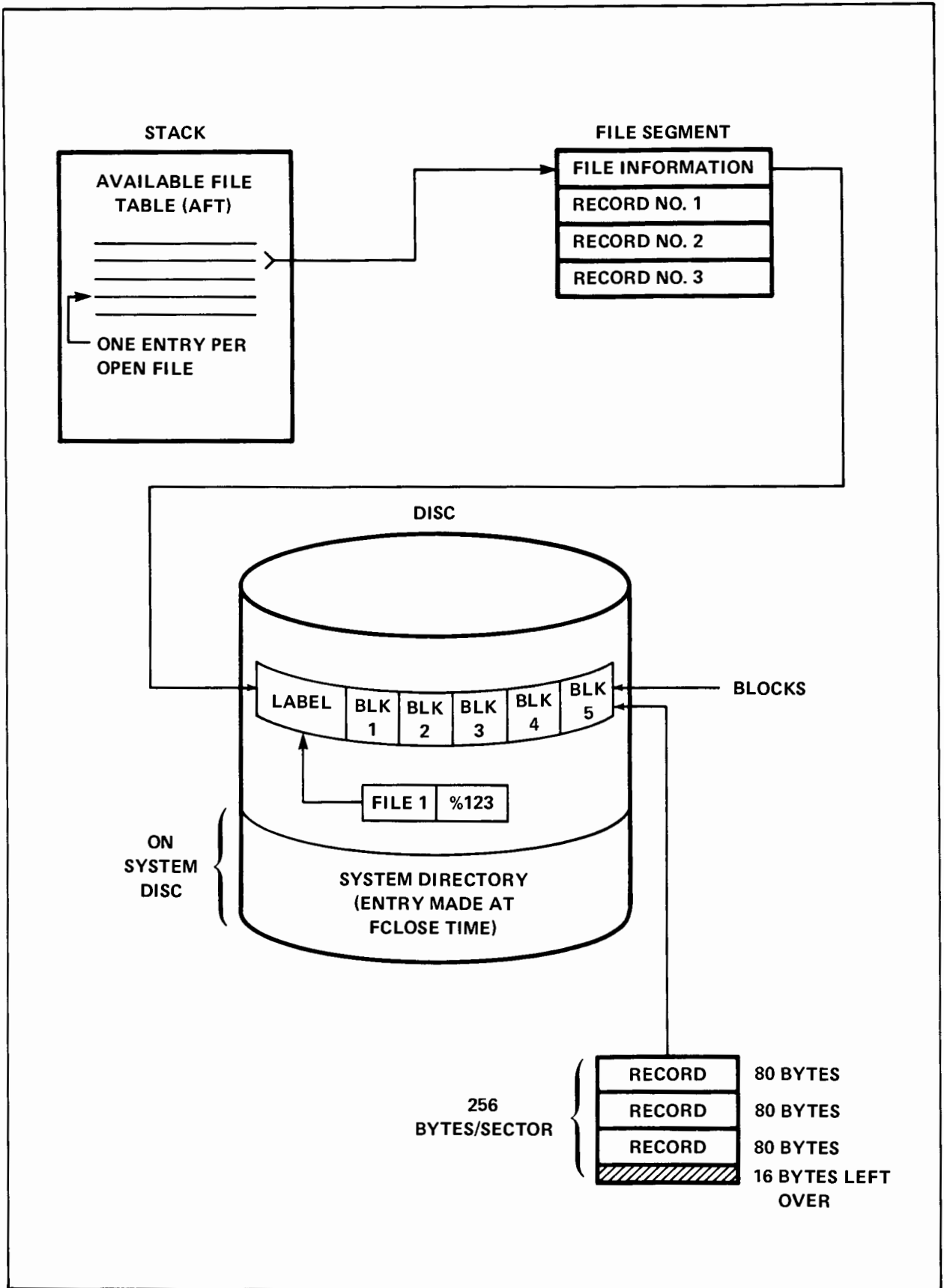


Figure 3-2. File Access Interface for New Disc Files

parameters specified in the FOPEN call). The file system then deallocates the free space and writes a *file label* in the first sector of the free space.

A partial list of items contained in a disc file label is shown below. Once their values are established, they cannot be changed by subsequent FOPEN operations.

- File name
- Sector address
- Maximum number of logical records
- Logical record size
- Block size
- Foptions (Exception: disallow file equations bit)
- Number of extents
- Extent size
- File code

The linkage, then, goes from the user's stack, via the available file table, to the file segment; from the file segment there is a pointer to the label on the disc itself. In the simplified example of figure 3-2, the file label is shown on the system disc, however, it could be on any disc in the system.

Since this is a new file, there is no information in the file. Therefore the access mechanism is the only information the system has for this file.

Depending on the FOPEN parameters specified, it is possible to write on this file. The example shows 80-byte records, three records per block, and one buffer. If an FWRITE intrinsic were called to write a single 80-byte record, that record would be moved from the user's stack to position number 1 in the file segment. As soon as that physical move from the stack to the file segment is complete, the FWRITE also is complete as far as the program is concerned. However, no actual write to the disc takes place. An FWRITE call to write record number 2 would consist of a move from the stack to the file segment and record number 2 would occupy position number 2 in the file segment. Subsequently, record number 3 would occupy position 3 in the file segment. Immediately upon the file segment being full, that is, when the third record has been written to the file segment, the entire block of information is then transferred to the disc. Thus, when the file system is used in a buffered manner with disc files, records actually are moved from the stack to the file segment, then, when the last record in a block has been moved into the file segment, a physical write to the disc occurs in a block fashion. That is, a whole block of information (in this case containing three records) is transferred to the system disc.

It is at FCLOSE time that you decide whether you want the file to remain in the system as a permanent file or a job/session temporary file, or whether you want the file to be deleted from the system.

At FCLOSE time, the access interface is dismantled and therefore if information about the file is to be saved in the system, the FCLOSE intrinsic is used to close the file with a permanent disposition. The name of the file, which is available to the system in the file label, is posted in the system directory under your log-on account and group. MPE finds that area of the directory and posts an entry in the system directory for that file name. If the name is FILE1, then FILE1 resides on the disc at a certain sector address. Referring to figure 3-2, you can see that in the system directory there will be an entry that consists of the file name and some sector address, for example, 123. The sector address 123 then points to the label.

As soon as the entry is made in the system directory, consisting of the name of the file and a pointer to the file label, then the FCLOSE operation continues and the file access interface is dismantled. The file segment is deleted from the system and the entry in your stack in the *available file table* (AFT) is purged. If the FCLOSE specifies the save permanent disposition, the name of the file will be placed in the system directory with a pointer to the file label.

As soon as the FCLOSE operation is complete, then, the disc file becomes a permanent file in the system, or, to use different terminology, it becomes an *old* disc file. If a file with the name of FILE1 already exists in your log-on account and group, it is not noticed until FCLOSE is issued, and, at that time, results in an error.

Figure 3-3 shows that now there is an entry in the system directory with a file name of FILE1 and a sector address of 123. To open this file, as an existing, or old file, the FOPEN parameters would have to be changed to specify an old file. Then the same type of operation that occurred before would occur again with one exception: since an existing file is being opened, it is not necessary to deallocate free space, what is necessary is for the system to establish a mechanism for the user stack area. In other words, the system makes an entry in the available file table, and creates another file segment, pointing to the existing file on the disc.

Figure 3-4 shows what occurs if an FOPEN call is issued for an old file named FILE1. In this case, FOPEN specifies an old file and must supply the name of this file; then MPE searches the system directory under the appropriate account and group for this file. Once the file is found, MPE then establishes the access mechanism, consisting of a file segment as before, and a pointer from the file segment to the file label on the disc.

The other type of old file is the job or session temporary file. One of the differences between a job temporary file and a permanent file is where the actual entry is placed when the file is closed. Each job or session has a table called the *job temporary file directory*. In the case of a file that is saved with temporary disposition, the name of the file and a pointer to the file label are stored in this job temporary file directory. (Note that the job temporary file directory is unique to each job or session.) Another difference between a job temporary file and a permanent file is that when a job/session terminates, all job/session temporary files are deleted from the system, and file space that was held by such files is returned to the system.

File characteristics are obtained from different sources, depending on whether the file is a new disc file, an old disc file, or a file on a device other than disc.

For a new disc file, the characteristics are established as shown below:

FOPEN:  
(create a new disc file)



The characteristics are obtained from:

FOPEN intrinsic parameters  
and defaults

overridden by:

:FILE command parameters

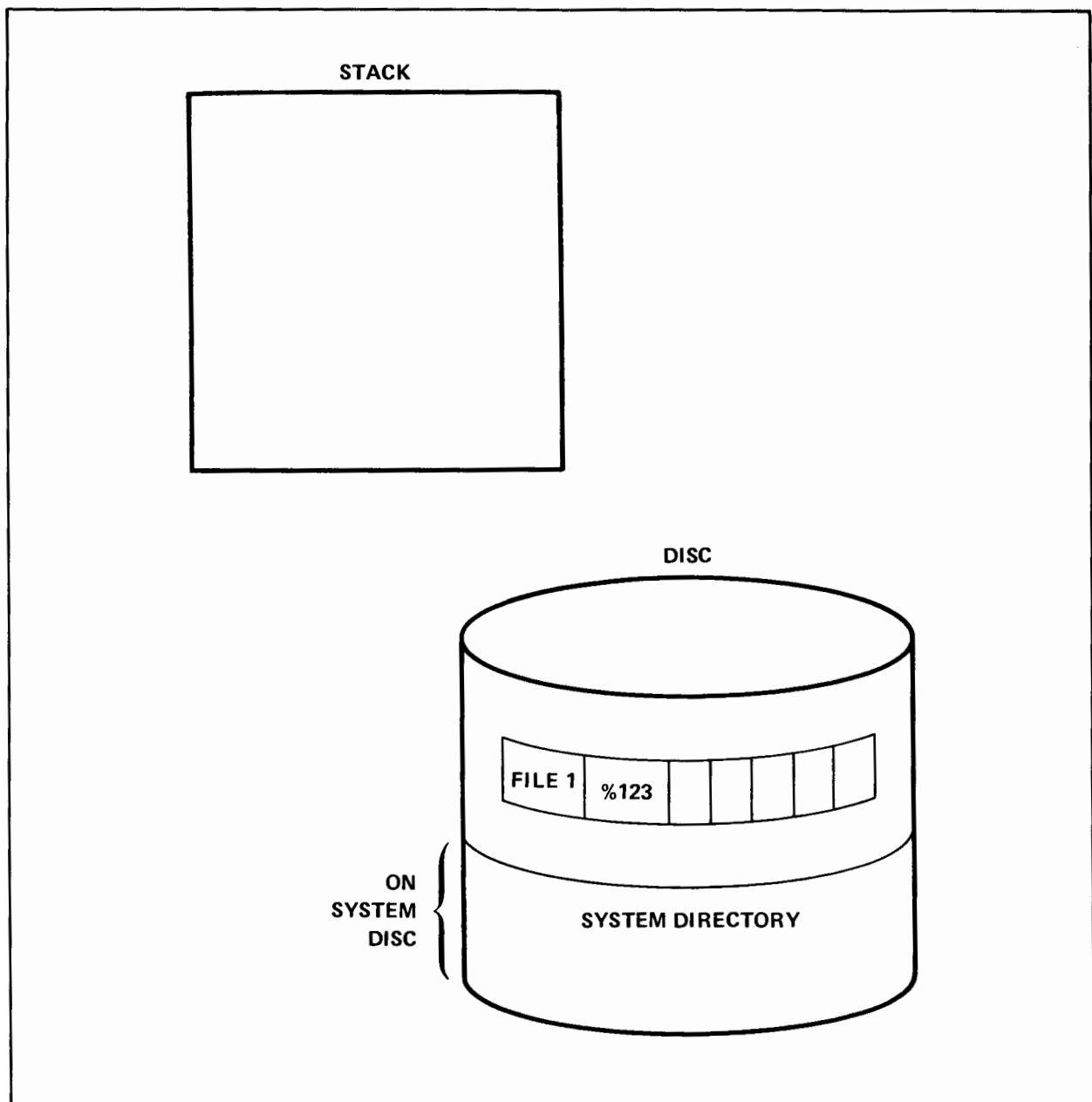


Figure 3-3. File Name and Sector Address Storage

A disc file and a file label are created according to the characteristics specified above. (This label remains with the file during its entire existence on the system.)

For an old disc file, the characteristics are established as follows:

FOPEN:  
 (open existing disc file)

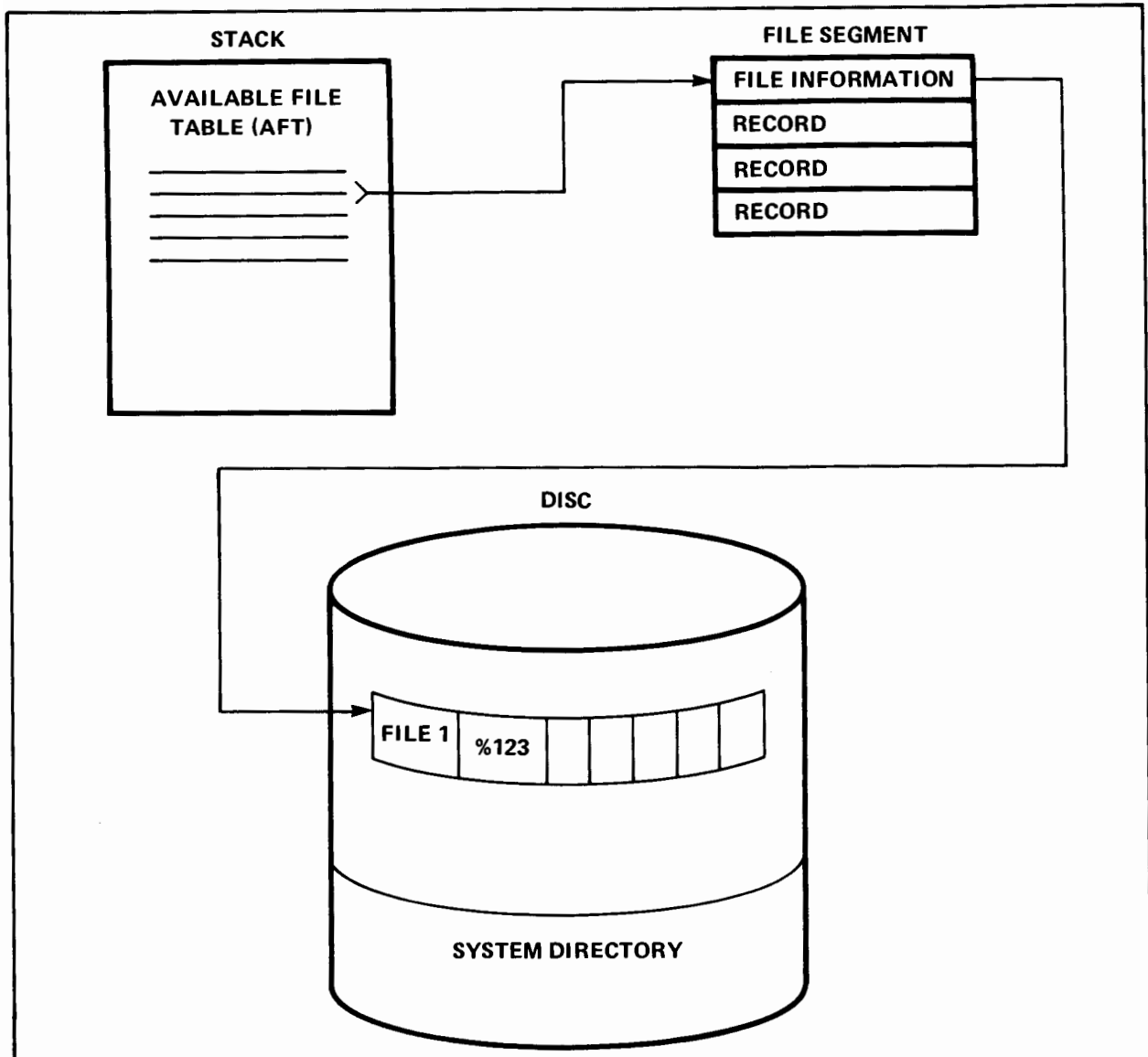


Figure 3-4. File Access Interface for Old Disc Files

The characteristics are obtained from:

FOPEN intrinsic parameters and defaults

overridden by:

:FILE command parameters

overridden by

Disc file label

The existing file can be either an old temporary file or an old permanent file.

When a file is opened on a device other than disc, the file characteristics are established as follows:

#### FOPEN

(open a device file)

The characteristics are obtained from:

FOPEN intrinsic parameters  
and defaults

overridden by:

:FILE command parameters

overridden by:

Device-dependent restraints  
imposed by the file system.

Note that if you have the ND (non-sharable device) capability, the file system allows you to open a physical I/O device in the same manner as you would open a disc file. (Discs are the only devices which are considered by MPE to be simultaneously sharable among several users.) When a non-sharable device has been FOPENed, it is referred to as a *devicefile*. The physical characteristics of each different device available to the file system can differ substantially and these differences affect the characteristics which are permitted for corresponding device files. For this reason, the file system imposes a number of device-dependent restrictions on device files. Card reader files, for example, are required to have read-only access with a block factor of one. A summary of these restrictions is presented in table 3-1.

It also should be noted that some non-sharable devices can be spooled by MPE. This means that data input from and output to such devices is stored temporarily on the disc in transit from the physical devices to and from the user program. Because data can be temporarily buffered in a disc file, the program assumes that all physical device files which it requires are constantly available to it. Input data typically is read and stored before a program requires it, and output data is delayed until the program's file operations are complete (at FCLOSE time). Other than these external variations, the differences between a spooled and a non-spooled device file are insignificant to the program.

When a non-sharable device file is opened, the device has to be *allocated* by the system so that the calling process can access the file. MPE classifies devices as OLD or NEW, OLD only, or NEW only, depending on the device type. Table 3-2 shows the manner in which devices are classified. Included in table 3-2 is the device name, its octal device number, and whether it is considered to be OLD/NEW, OLD only, or NEW only.

The flowchart shown in figure 3-5 illustrates how MPE allocates a non-sharable device when an FOPEN request is received.

First, MPE considers the device type requested by the FOPEN call. If the device type is input only, this is considered to be an OLD file. Because MPE considers the file to be an OLD file, it searches for a pre-defined input file. For example, a file identified with a :DATA command. If no such file is found, MPE sends a message to the Console Operator asking for the logical device number of the input device.

Table 3-1. Device-Dependent Restrictions

INPUT ONLY DEVICES (SERIAL)

Card Reader/Paper Tape Reader

No carriage control  
Undefined-length records  
If card reader, ASCII only (can only read ASCII cards without using FCONTROL)  
Blockfactor = 1  
Domain = 1 (OLD permanent)  
If not ASCII, then NOBUF  
If access type = 1, 2, 3, then access violation results

INPUT/OUTPUT DEVICES (PARALLEL)

Terminals

ASCII  
NOBUF  
Undefined-length records  
Blockfactor = 1

INPUT/OUTPUT DEVICES (SERIAL)

Magnetic Tape Drive

No restriction other than no file label

OUTPUT ONLY (SERIAL)

Line Printer/Card Punch/Paper Tape Punch/Plotter

If Paper Tape Punch, ASCII only  
Undefined-length records  
Blockfactor = 1  
Domain = NEW  
Access Type = 1, write only (if read only specified, access violation results)

UNDEFINED (COMMON CHECKING)

If carriage control specified and not ASCII, access violation results



Table 3-2. Classification of Devices

DEVICE NAME	DEVICE TYPE NUMBER (OCTAL)	CLASSIFICATION
Moving-Head Disc	00	NEW or OLD
Fixed-Head Disc	01	NEW or OLD
Card Reader	10	OLD only
Paper Tape Reader	11	OLD only
Terminal	20	NEW or OLD
Printing Reader Punch	24	NEW or OLD
Synchronous Single-Line Controller	26	NEW or OLD
Programmable Controller	27	NEW or OLD
Magnetic Tape Drive	30	NEW or OLD
Line Printer	40	NEW only
Card Punch	41	NEW only
Paper Tape Punch	42	NEW only
Plotter	43, 44, 45	NEW only

If FOPEN specified a device type that is considered by MPE to be output only, MPE considers this to be a NEW file. Normally, NEW files do not require special attention. If the device is available, it will be allocated to the user. Printer forms message, plotter, or magnetic tape requests are the exceptions, however, and require operator intervention.

If a device was specified that is an input/output type of device, MPE next considers the user access requested in the FOPEN call. If read only was requested, the file is considered to be an OLD file. If write only, MPE considers the file to be a NEW file.

If the FOPEN call requested a user access of input/output, or any other mode (except read only or write only), MPE next looks at the type of file domain specified in the call (NEW or OLD) and opens the file accordingly. The system device directories contain entries for each device that contains a file. Non-spooled devices can have only one file (for example, a card deck in the read hopper of a card reader), but spooled devices can have several file entries (for example, card decks which have been read in by the device and are stored as spoolfiles on disc to await access). Such device files are identified by :DATA commands. Information from a :DATA command image is used to build the device directory entry and identifies the file by file name, user name, and account name. The data file may be accessed by a user program when its request matches the :DATA

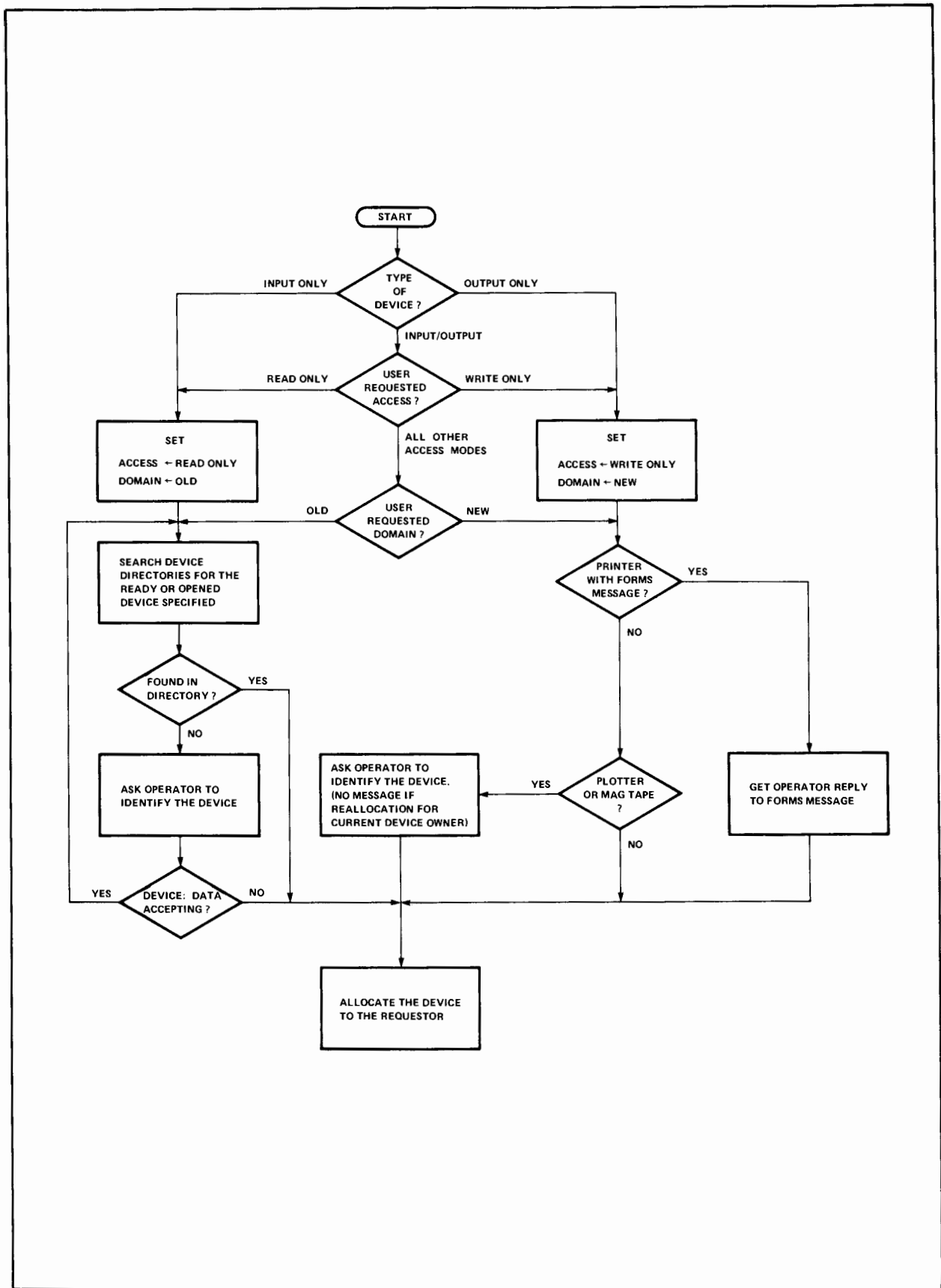


Figure 3-5. Device Allocation Flowchart

information and the file is in the READY state. In the case of an unspooled card reader, this means that only the :DATA card has been read in, and the rest of the deck awaits processing. In the case of a spooled card reader, however, this means that the :DATA card and the entire deck have been read and await processing in the form of a disc spoolfile.

If the entry in the device directory indicates that the device is OPENED, a user process has already FOPENed the device file successfully (device or spoolfile). In this case, access to the same non-sharable device is granted only if the requesting process is the same process which has the file open. In other words, the original owner of the device issued another call to FOPEN to open the same file. These subsequent calls to FOPEN will not require operator intervention — the first device allocation request is the only one issued to the operator. This technique might be used by a program which does a great deal of magnetic tape processing but wants to avoid multiple tape allocation messages.

A condition code error is returned to the calling process if:

1. Device type specified an input only device and requested access was write only.
2. Device type specified an output only device and the requested access was read only.

A message to the operator will be printed if:

1. The device is a card reader and a pre-defined file (read in with a :DATA card) cannot be located to match the file requested.
2. The device is a magnetic tape device.
3. The device is a plotter.
4. The device is a line printer and uses the forms message option.

## OPENING FILES

### OPENING A NEW DISC FILE

Figure 3-6 contains an SPL program which opens two files: a card reader file and a new disc file.

The second FOPEN call in figure 3-6.

```
OUT:=FOPEN(OUTPUT,%4,%101,128);
```

opens the new disc file. The parameters specified are

*formaldesignator*            DATAONE, which is contained in the byte array OUTPUT.

*foptions*                    %4, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	Binary
														4		Octal

```

00001000 00000 0 $CONTROL USLIMIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INPUT(0:6):="INFILE "
00004000 00005 1 BYTE ARRAY DEV(0:4):="CARD "
00005000 00004 1 BYTE ARRAY OUTPUT(0:7):="DATAONE "
00006000 00005 1 ARRAY BUFFER(0:127)
00007000 00005 1 INTEGER IN,OUT,LGTH
00008000 00005 1
00009000 00005 1 INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT,FILE,INFO,QUIT
00010000 00005 1
00011000 00005 1 << END OF DECLARATIONS >>
00012000 00005 1
00013000 00005 1 IN:=FOPEN(INPUT,%5,,40,DEV) <<CARD READER>>
00014000 00012 1 IF < THEN <<CHECK FOR ERROR>>
00015000 00013 1 BEGIN
00016000 00013 2 PRINT,FILE,INFO(IN) <<PRINT ERROR>>
00017000 00015 2 QUIT(1) <<ABORT>>
00018000 00017 2 END
00019000 00017 1
00020000 00017 1 OUT:=FOPEN(OUTPUT,%4,%101,128) <<NEW DISC FILE>>
00021000 00030 1 IF < THEN <<CHECK FOR ERROR>>
00022000 00031 1 BEGIN
00023000 00031 2 PRINT,FILE,INFO(OUT) <<PRINT ERROR>>
00024000 00033 2 QUIT(2) <<ABORT>>
00025000 00035 2 END
00026000 00035 1
00027000 00035 1 COPY LOOP:
00028000 00035 1 LGTH:=FREAD(IN,BUFFER,40) <<READ A CARD>>
00029000 00043 1 IF < THEN <<CHECK FOR ERROR>>
00030000 00044 1 BEGIN
00031000 00044 2 PRINT,FILE,INFO(IN) <<PRINT ERROR>>
00032000 00046 2 QUIT(3) <<ABORT>>
00033000 00050 2 END
00034000 00050 1 IF > THEN GO END OF FILE <<CHECK FOR EOF>>
00035000 00051 1
00036000 00051 1 FWRITE(OUT,BUFFER,LGTH,0) <<COPY CARD TO DISC>>
00037000 00056 1 IF <> THEN <<CHECK FOR ERROR>>
00038000 00057 1 BEGIN
00039000 00057 2 PRINT,FILE,INFO(OUT) <<PRINT ERROR>>
00040000 00061 2 QUIT(4) <<ABORT>>
00041000 00063 2 END
00042000 00063 1
00043000 00063 1 GO COPY LOOP <<CONTINUE COPYING>>
00044000 00066 1
00045000 00066 1 END OF FILE:
00046000 00066 1 FCLOSE(OUT,%11,0) <<MAKE PERMANENT>>
00047000 00072 1 IF < THEN <<CHECK FOR ERROR>>
00048000 00073 1 BEGIN
00049000 00073 2 PRINT,FILE,INFO(OUT) <<PRINT ERROR>>
00050000 00075 2 QUIT(5) <<ABORT>>
00051000 00077 2 END
00052000 00077 1 END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00213
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:44

```

Figure 3-6. Opening a New Disc File

The above bit pattern specifies the following file options:

Domain: New file, no search of system or job temporary file directory is necessary. Bits (14:2) = 00.

ASCII/Binary: ASCII. Bit (13:1) = 1.

*options*

%101, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	Binary
								1		0				1		Octal

The above bit pattern specifies the following access options:

Access Type: Write access only. Bits (12:4) = 0001.

Exclusive: Exclusive access. Bits (8:2) = 01.

All other parameters are omitted from the FOPEN intrinsic call.

Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable OUT.

The condition code is checked with the

```
IF < THEN
```

statement. If the condition code is CCL, signifying that the FOPEN request was denied, the next four statements, starting with the BEGIN statement, are executed.

The statement

```
PRINT'FILE'INFO(OUT);
```

calls the PRINT'FILE'INFO intrinsic, which prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FOPEN. The parameter (OUT) specifies the file number returned through the FOPEN intrinsic. If the file was not opened successfully, OUT = 0, where 0 specifies that the FILE INFORMATION DISPLAY will reflect the status of the file referenced in the last call to FOPEN. See Section X for a discussion of the FILE INFORMATION DISPLAY

The QUIT intrinsic call

```
QUIT(2);
```

aborts the process. The parameter (2) is an arbitrary user-supplied number. When a QUIT intrinsic is executed, this number is printed as part of the resulting abort message, allowing you to determine, in the case of multiple QUIT intrinsic calls in a program, which specific QUIT call was executed.

## NOTE

The QUIT intrinsic causes MPE to close all files with no change. Thus, new files are deleted, old files are saved and assigned to the same domain to which they belonged previously.

### OPENING AN OLD DISC FILE

Figure 3-7 contains an SPL program that opens three files: an old disc file, \$STDIN, and \$STDLIST.

The statement

```
DFILE1:=FOPEN(DATA1,%5,%345,128):
```



opens the old disc file. The parameters specified are

*formal designator*            DATAONE, which is contained in the byte array DATA1.

*foptions*                    %5, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	Binary
												5			Octal	

The above bit pattern specifies the following file options:

Domain: Old permanent file, the system file directory should be searched. Bits (14:2) = 01.

ASCII/Binary: ASCII. Bit (13:1) = 1.

*aoptions*                    %345, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	1	Binary
						3			4			5			Octal	

The above bit pattern specifies the following access options:

Access Type: Update access. (This file is updated later in the program with the FUPDATE intrinsic.) Bits (12:4) = 0101.

Multirecord: Non-multirecord mode. Bit (11:1) = 0.

Dynamic Locking: Dynamic locking allowed. Bit (10:1) = 1.

Exclusive: Share access. Bits (8:2) = 11.

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1   BYTE ARRAY DATA1(0:7):="DATAONE "
00004000 00005 1   ARRAY BUFFER(0:127)
00005000 00005 1   INTEGER DFILE1, LGTH, DUMMY, IN, LIST
00006000 00005 1
00007000 00005 1   INTRINSIC FOPEN, FREAD, FUPDATE, FLOCK, FUNLOCK, FCLOSE,
00008000 00005 1   PRINT, FILE, INFO, QUIT, FWRITE, FREAD
00009000 00005 1
00010000 00005 1   PROCEDURE FILEERROR(FILENO, QUITNO)
00011000 00000 1     VALUE QUITNO
00012000 00000 1     INTEGER FILENO, QUITNO
00013000 00000 1     BEGIN
00014000 00000 2       PRINT, FILE, INFO(FILENO)
00015000 00002 2       QUIT(QUITNO)
00016000 00004 2     END
00017000 00000 1
00018000 00000 1   <<END OF DECLARATIONS>>
00019000 00000 1
00020000 00000 1   DFILE1:=FOPEN(DATA1,%5,%345,128) <<OLD DISC FILE>>
00021000 00011 1   IF < THEN FILEERROR(DFILE1,1) <<CHECK FOR ERROR>>
00022000 00015 1
00023000 00015 1   IN:=FOPEN(,%244) <<$STDIN>>
00024000 00024 1   IF < THEN FILEERROR(IN,2) <<CHECK FOR ERROR>>
00025000 00030 1
00026000 00030 1   LIST:=FOPEN(,%614,%1) <<$STDLIST>>
00027000 00040 1   IF < THEN FILEERROR(LIST,3) <<CHECK FOR ERROR>>
00028000 00044 1
00029000 00044 1   UPDATE'LOOP:
00030000 00044 1     FLOCK(DFILE1,1) <<LOCK FILE/SUSPEND>>
00031000 00047 1     IF < THEN FILEERROR(DFILE1,4) <<CHECK FOR ERROR>>
00032000 00053 1
00033000 00053 1     LGTH:=FREAD(DFILE1,BUFFER,128) <<GET EMPLOYEE RECD>>
00034000 00061 1     IF < THEN FILEERROR(DFILE1,5) <<CHECK FOR ERROR>>
00035000 00065 1     IF > THEN GO END'OF'FILE <<CHECK FOR EOF>>
00036000 00070 1
00037000 00070 1     FWRITE(LIST,BUFFER,-20,%320) <<EMPLOYEE NAME>>
00038000 00075 1     IF <> THEN FILEERROR(LIST,6) <<CHECK FOR ERROR>>
00039000 00101 1
00040000 00101 1     DUMMY:=FREAD(IN,BUFFER(30),5) <<EMPLOYEE NUMBER>>
00041000 00110 1     IF < THEN FILEERROR(IN,7) <<CHECK FOR ERROR>>
00042000 00114 1     IF > THEN GO END'OF'FILE
00043000 00115 1
00044000 00115 1     FUPDATE(DFILE1,BUFFER,128) <<EMPLOYEE RECORD>>
00045000 00121 1     IF <> THEN FILEERROR(DFILE1,8) <<CHECK FOR ERROR>>
00046000 00125 1
00047000 00125 1     FUNLOCK(DFILE1) <<ALLOW OTHER ACCESS>>
00048000 00127 1     IF <> THEN FILEERROR(DFILE1,9) <<CHECK FOR ERROR>>
00049000 00133 1
00050000 00133 1     GO UPDATE'LOOP <<CONTINUE UPDATE>>
00051000 00140 1
00052000 00140 1   END'OF'FILE:
00053000 00140 1     FUNLOCK(DFILE1) <<ALLOW OTHER ACCESS>>
00054000 00142 1     IF <> THEN FILEERROR(DFILE1,10) <<CHECK FOR ERROR>>
00055000 00146 1
00056000 00146 1     FCLOSE(DFILE1,0,0) <<DISP=NO CHANGE>>
00057000 00151 1     IF < THEN FILEERROR(DFILE1,11) <<CHECK FOR ERROR>>
00058000 00155 1   END.
    PRIMARY DB STORAGE=%007;   SECONDARY DB STORAGE=%00204
    NO. ERRORS=0001;           NO. WARNINGS=000
    PROCESSOR TIME=0:00:03;    ELAPSED TIME=0:00:17

```

Figure 3-7. Opening an Old Disc File

All other parameters are omitted in the FOPEN intrinsic call.

Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable DFILE1.

The condition code is checked with the statement

```
IF < THEN FILEERROR(DFILE1,1);
```

If the condition code is CCL, the error-check procedure FILEERROR (see statements 10 through 16 in the program) is called and two parameters, DFILE1 and 1, are passed to it for FILENO and QUITNO (see statement number 10). DFILE1 contains the file number (assigned to it when the FOPEN intrinsic opened the file) to be passed by FILENO, and 1 represents an arbitrary user-supplied number to be passed by QUITNO.

The FILEERROR intrinsic passes the file number (through FILENO) to the PRINT'FILE'INFO intrinsic. If the file was not opened successfully, FILENO = 0, where 0 specifies the status of the file referenced in the last call to FOPEN. The PRINT'FILE'INFO intrinsic prints a FILE INFORMATION DISPLAY on the standard output device, enabling you to determine the error number returned by FOPEN. See Section X for a discussion of the FILE INFORMATION DISPLAY.

The QUIT intrinsic call (statement 15)

```
QUIT(QUITNO);
```

aborts the program's process. The value of QUITNO is 1 and this number is printed as part of the resulting abort message, allowing you to determine, in the case of multiple QUIT intrinsic calls in a program, which specific QUIT call was executed.

## OPENING A FILE ON A DEVICE OTHER THAN DISC

Figure 3-8 contains an SPL program that opens a card reader file and a disc file, reads the contents of a card deck and writes the records read from the card deck into the disc file and, finally, closes the disc file as a permanent file.

### NOTE

If a card deck is read in by the spooler before the program which references the deck executes, the system finds an entry for the card reader file in the device directory and allocation is automatic. If the card deck is not read before the program executes, however, the system will print a message on the system console requesting the Console Operator to reply with the logical device number of the device on which the file resides. The message and the reply are of the following form:

```
?IO/13:44/#S28/28/LDEV# FOR "HRG" ON CARD (NUM)
```

```
=REPLY 28, 5
```



```

00001000 00000 0  SCONTROL USLIMIT
00002000 00000 0  BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:6):="INFILE "
00004000 00005 1  BYTE ARRAY DEV(0:4):="CARD "
00005000 00004 1  BYTE ARRAY OUTPUT(0:7):="DATAONE "
00006000 00005 1  ARRAY BUFFER(0:127)
00007000 00005 1  INTEGER IN,OUT,LGTH
00008000 00005 1
00009000 00005 1  INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT
00010000 00005 1
00011000 00005 1  << END OF DECLARATIONS >>
00012000 00005 1
00013000 00005 1  IN:=FOPEN(INPUT,%5,,40,DEV)  <<CARD READER>>
00014000 00012 1  IF < THEN  <<CHECK FOR ERROR>>
00015000 00013 1  BEGIN
00016000 00013 2  PRINT'FILE'INFO(IN)  <<PRINT ERROR>>
00017000 00015 2  QUIT(1)  <<ABORT>>
00018000 00017 2  END
00019000 00017 1
00020000 00017 1  OUT:=FOPEN(OUTPUT,%4,%101,128)  <<NEW DISC FILE>>
00021000 00030 1  IF < THEN  <<CHECK FOR ERROR>>
00022000 00031 1  BEGIN
00023000 00031 2  PRINT'FILE'INFO(OUT)  <<PRINT ERROR>>
00024000 00033 2  QUIT(2)  <<ABORT>>
00025000 00035 2  END
00026000 00035 1
00027000 00035 1  COPY:LOOP:
00028000 00035 1  LGTH:=FREAD(IN,BUFFER,40)  <<READ A CARD>>
00029000 00043 1  IF < THEN  <<CHECK FOR ERROR>>
00030000 00044 1  BEGIN
00031000 00044 2  PRINT'FILE'INFO(IN)  <<PRINT ERROR>>
00032000 00046 2  QUIT(3)  <<ABORT>>
00033000 00050 2  END
00034000 00050 1  IF > THEN GO END'OF'FILE  <<CHECK FOR EOF>>
00035000 00051 1
00036000 00051 1  FWRITE(OUT,BUFFER,LGTH,0)  <<COPY CARD TO DISC>>
00037000 00056 1  IF <> THEN  <<CHECK FOR ERROR>>
00038000 00057 1  BEGIN
00039000 00057 2  PRINT'FILE'INFO(OUT)  <<PRINT ERROR>>
00040000 00061 2  QUIT(4)  <<ABORT>>
00041000 00063 2  END
00042000 00063 1  GO COPY:LOOP  <<CONTINUE COPYING>>
00043000 00063 1
00044000 00066 1
00045000 00066 1  END'OF'FILE:
00046000 00066 1  FCLOSE(OUT,%11,0)  <<MAKE PERMANENT>>
00047000 00072 1  IF < THEN  <<CHECK FOR ERROR>>
00048000 00073 1  BEGIN
00049000 00073 2  PRINT'FILE'INFO(OUT)  <<PRINT ERROR>>
00050000 00075 2  QUIT(5)  <<ABORT>>
00051000 00077 2  END
00052000 00077 1  END.
PRIMARY DB STORAGE=%007  SECONDARY DB STORAGE=%00213
NO. ERRORS=000  NO. WARNINGS=000
PROCESSOR TIME=0:00:03  ELAPSED TIME=0:00:44

```

Figure 3-8. Opening a File on a Device Other Than Disc

The NOT READY message was printed because the read hopper of the card reader was emptied by the spooler when the INFILE deck was read.

In figure 3-8, the statement

```
IN:=FOPEN(INPUT,%5,,40,DEV);
```

calls the FOPEN intrinsic to open the card reader file. The parameters specified are

*formal designator* INFILE, which is contained in the byte array INPUT.

*options* %5, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	Binary
													5	Octal		

The above bit pattern specifies the following file options:

Domain: Old permanent file, system file domain. Bits (14:2) = 01.  
 ASCII/Binary: ASCII. Bit (13:1) = 1.

*options* Omitted. All bits are set to zero, access defaults to READ only.

*resize* 40 words.

*device* CARD. The byte array DEV, containing the string "CARD", is specified for the *device* parameter.

All other parameters are omitted in the FOPEN intrinsic call.

Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable IN.

The next statement in the program

```
IF < THEN
```

checks the condition code. If the condition code is CCL, signifying that the FOPEN request was denied, the next four statements, starting with the BEGIN statement, are executed.

The statement

```
PRINT'FILE'INFO(IN);
```

calls the PRINT'FILE'INFO intrinsic, which prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FOPEN. The

parameter (IN) specifies the file number returned through the FOPEN intrinsic. If the file was not opened successfully, IN = 0, where 0 specifies that the FILE INFORMATION DISPLAY will reflect the status of the file referenced in the last call to FOPEN. See Section X for a discussion of the FILE INFORMATION DISPLAY.

The QUIT intrinsic call

```
QUIT(1);
```

aborts the process. The parameter (1) is an arbitrary user-specified number. When a QUIT intrinsic is executed, this number is printed as part of the resulting abort message, allowing you to determine, in the case of multiple QUIT intrinsic calls in a program, which specific QUIT call was executed.

### ISSUING FREAD AND FWRITE INTRINSIC CALLS FOR \$STDIN AND \$STDLIST

If the standard input device (\$STDIN) and standard list device (\$STDLIST) are opened with the FOPEN intrinsic, then FREAD and FWRITE intrinsic calls can be used with these devices. For example, the FREAD intrinsic can be used to transfer information entered from a terminal to a buffer in the stack; and the FWRITE intrinsic can be used to transfer information from a buffer in the stack directly to the standard list device.

**OPENING \$STDIN.** Figure 3-9 contains a program that opens \$STDIN so that FREAD intrinsic calls can be issued directly against the standard input device (a terminal in this case; the program was run interactively).

The standard input device is opened with the FOPEN intrinsic call

```
IN:=FOPEN(,%244);
```

The parameters specified in the above intrinsic call are as follows:

*formal designator*                      Omitted.  
 Default: A temporary, nameless file that can be read, but not saved, is assigned.

*foptions*                                %244, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	Binary
								2		4			4			Octal

The above bit pattern specifies the following file options:

- Domain: New file, no search of system or job temporary file directory is necessary. Bits (14:2) = 00.
- ASCII/Binary: ASCII. Bit (13:1) = 1.
- Default Designator: \$STDIN. Bits (10:3) = 100.
- Record Format: Undefined length. Bits (8:2) = 10.

```

00001000 00000 0 $CONTROL USLIMIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA1(0:7):="DATAONE "
00004000 00005 1 ARRAY BUFFER(0:127)
00005000 00005 1 INTEGER DFILE1, LGTH, DUMMY, IN, LIST
00006000 00005 1
00007000 00005 1 INTRINSIC FOPEN, FREAD, FUPDATE, FLOCK, FUNLOCK, FCLOSE,
00008000 00005 1 PRINT'FILE'INFO, QUIT, FWRITE, FREAD
00009000 00005 1
00010000 00005 1 PROCEDURE FILEERROR(FILENO, QUITNO)
00011000 00000 1 VALUE QUITNO
00012000 00000 1 INTEGER FILENO, QUITNO
00013000 00000 1 BEGIN
00014000 00000 2 PRINT'FILE'INFO(FILENO)
00015000 00002 2 QUIT(QUITNO)
00016000 00004 2 END
00017000 00000 1
00018000 00000 1 <<END OF DECLARATIONS>>
00019000 00000 1
00020000 00000 1 DFILE1:=FOPEN(DATA1,%5,%345,128) <<OLD DISC FILE>>
00021000 00011 1 IF < THEN FILEERROR(DFILE1,1) <<CHECK FOR ERROR>>
00022000 00015 1
00023000 00015 1 IN:=FOPEN(,%244) <<$STDIN>>
00024000 00024 1 IF < THEN FILEERROR(IN,2) <<CHECK FOR ERROR>>
00025000 00030 1
00026000 00030 1 LIST:=FOPEN(,%614,%1) <<$STDLIST>>
00027000 00040 1 IF < THEN FILEERROR(LIST,3) <<CHECK FOR ERROR>>
00028000 00044 1
00029000 00044 1 UPDATE'LOOP:
00030000 00044 1 FLOCK(DFILE1,1) <<LOCK FILE/SUSPEND>>
00031000 00047 1 IF < THEN FILEERROR(DFILE1,4) <<CHECK FOR ERROR>>
00032000 00053 1
00033000 00053 1 LGTH:=FREAD(DFILE1,BUFFER,128) <<GET EMPLOYEE RECD>>
00034000 00061 1 IF < THEN FILEERROR(DFILE1,5) <<CHECK FOR ERROR>>
00035000 00065 1 IF > THEN GO END'OF'FILE <<CHECK FOR EOF>>
00036000 00070 1
00037000 00070 1 FWRITE(LIST,BUFFER,-20,%320) <<EMPLOYEE NAME>>
00038000 00075 1 IF <> THEN FILEERROR(LIST,6) <<CHECK FOR ERROR>>
00039000 00101 1
00040000 00101 1 DUMMY:=FREAD(IN,BUFFER(30),5) <<EMPLOYEE NUMBER>>
00041000 00110 1 IF < THEN FILEERROR(IN,7) <<CHECK FOR ERROR>>
00042000 00114 1 IF > THEN GO END'OF'FILE
00043000 00115 1
00044000 00115 1 FUPDATE(DFILE1,BUFFER,128) <<EMPLOYEE RECORD>>
00045000 00121 1 IF <> THEN FILEERROR(DFILE1,8) <<CHECK FOR ERROR>>
00046000 00125 1
00047000 00125 1 FUNLOCK(DFILE1) <<ALLOW OTHER ACCESS>>
00048000 00127 1 IF <> THEN FILEERROR(DFILE1,9) <<CHECK FOR ERROR>>
00049000 00133 1
00050000 00133 1 GO UPDATE'LOOP <<CONTINUE UPDATE>>
00051000 00140 1
00052000 00140 1 END'OF'FILE:
00053000 00140 1 FUNLOCK(DFILE1) <<ALLOW OTHER ACCESS>>
00054000 00142 1 IF <> THEN FILEERROR(DFILE1,10) <<CHECK FOR ERROR>>
00055000 00146 1
00056000 00146 1 FCLOSE(DFILE1,0,0) <<DISP=NO CHANGE>>
00057000 00151 1 IF < THEN FILEERROR(DFILE1,11) <<CHECK FOR ERROR>>
00058000 00155 1 END.
PRIMARY DR STORAGE=%007; SECONDARY DB STORAGE=%00204
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:17

```

Figure 3-9. Opening \$STDIN and \$STDLIST

*aoptions* Omitted. All bits are set to zero, access defaults to READ only.

All other parameters are omitted in the FOPEN intrinsic call.

Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable IN.

The next statement in the program

```
IF < THEN FILEERROR(IN,2);
```

checks the condition code. If the condition code is CCL, signifying that the FOPEN request was denied, the error-check procedure FILEERROR is called.

The FILEERROR procedure (see statements 10 through 16 in the program) calls the PRINT'FILE'INFO intrinsic, which prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FOPEN.

The QUIT intrinsic call (statement number 15) aborts the process.

OPENING \$STDLIST. In figure 3-9, the statement

```
LIST:=FOPEN(%14,%1);
```

opens the standard list device so that the FWRITE intrinsic can be used to transfer information directly to the device.

The parameters specified in the above intrinsic call are

*formal designator* Omitted.  
Default: A temporary, nameless file that can be written on, but not saved, is assigned.

*foptions* %614, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	Binary
							6			1			4			Octal

The above bit pattern specifies the following file options:

Domain: New file, no search of system or job temporary file directory is necessary. Bits (14:2) = 00.

ASCII/Binary: ASCII. Bit (13:1) = 1.

Default Designator: \$STDLIST. Bits (10:3) = 001.

Record Format: Undefined length. Bits (8:2) = 10.

Carriage Control: Carriage control character expected. Bit (7:1) = 1.

*options*

%1, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	Binary
														1		Octal

The foregoing bit pattern specifies the following access options:

Access Type: Write access only. Bits (12:4) = 0001.

All other parameters are omitted in the FOPEN intrinsic call.

Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable LIST.

The next statement in the program

```
IF < THEN FILEERROR(LIST,3);
```

checks the condition code. If the condition code is CCL, signifying that the FOPEN request was denied, the error-check procedure FILEERROR is called.

The FILEERROR procedure (see statements 10 through 16 in the program) calls the PRINT'FILE'INFO intrinsic, which prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FOPEN.

The QUIT intrinsic call (statement number 15) aborts the process.

## CLOSING FILES

To terminate access to a file, you use the FCLOSE intrinsic. The FCLOSE intrinsic applies to files on all devices, and de-allocates the device on which the file resides. If a file was FOPENed concurrently several times by the same user, the device is not de-allocated until the last "nested" FCLOSE intrinsic is executed.

The FCLOSE intrinsic may be used to change the disposition of a file. For example, a file opened as a new file can be closed and saved as an old file with permanent or temporary disposition.

When the FOPEN intrinsic opens a file specified as new in the *foptions* parameter (bits 14 and 15 = 00), no search of the job temporary or system file domains is conducted to ensure that a file of the same name does not exist already. If such a file is closed and saved with the FCLOSE intrinsic, however, a search is conducted. The job temporary file domain is searched if the file is to be saved as a temporary job/session file and the system file domain is searched if the file is to be saved as a permanent file. If a file of the same name is found in either directory, an error code is returned to the calling process. Thus, it is possible to open a new file with the same file name as an existing file, but an error will result if an FCLOSE intrinsic attempts to save such a file in the same domain with a file of the same name.

Similarly, when the FOPEN intrinsic opens a file specified as old temporary in the *foptions* parameter (bits 14 and 15 = 10), only the job temporary file domain (not the system file domain) is searched. If such a file is closed and saved as a permanent file with the FCLOSE intrinsic, the system file domain is searched. If a file of the same name is found, an error code is returned to the calling process.

If an FCLOSE intrinsic call is not issued in a program in which files have been opened, MPE closes all files automatically when the program's process terminates. In this case, all opened files are closed with the same disposition they had before being opened. New files are deleted, old files are saved and assigned to the domain to which they belonged previously — either permanent or temporary.

### CLOSING A NEW FILE AS A TEMPORARY FILE

Figure 3-10 contains an FCLOSE intrinsic call that closes a new file as a temporary job file.

The FCLOSE intrinsic call

```
FCLOSE(DFILE2,2,0);
```

closes the file specified by DFILE2. The parameters specified in the above intrinsic call are

*filenum* Contained in the identifier DFILE2. The file number was assigned to DFILE2 when FOPEN opened the file.

*disposition* 2, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	Binary
													2	Octal		

The above bit pattern specifies the following:

Domain Disposition: Temporary job file (rewound).  
 The file is retained in the user's temporary (job/session) file domain and can thus be re-opened by any process within the job/session. The uniqueness of the file name is checked; if a file of this name already exists in the job temporary file domain, an error code is returned. If the file resides on magnetic tape, the tape is rewound but not unloaded. Bits (13:3) = 010.

Disc Space Disposition: Unused disc space not returned to the system.  
 Bit (12:1) = 0.

*seccode* 0, unrestricted access.

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000 00005 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00005000 00005 1 ARRAY LABL(0:8):="EMPLOYEE DATA FILE";
00006000 00011 1 ARRAY BUFFER(0:127);
00007000 00011 1 INTEGER DFILE1,DFILE2,DUMMY;
00008000 00011 1 DOUBLE REC;
00009000 00011 1
00010000 00011 1 INTRINSIC FOPEN,FWRITELABEL,FGETINFO,FREAD,FWRITEDIR,FCLOSE,
00011000 00011 1 PRINT'FILE'INFO,QUIT;
00012000 00011 1
00013000 00011 1 PROCEDURE FILERROR(FILENO,QUITNO);
00014000 00000 1 VALUE QUITNO;
00015000 00000 1 INTEGER FILENO,QUITNO;
00016000 00000 1 BEGIN
00017000 00000 2 PRINT'FILE'INFO(FILENO);
00018000 00002 2 QUIT(QUITNO);
00019000 00004 2 END;
00020000 00000 1
00021000 00000 1 <<END OF DECLARATIONS>>
00022000 00000 1
00023000 00000 1 DFILE1:=FOPEN(DATA1,%5,%100); <<OLD FILE-DATAONE>>
00024000 00010 1 IF < THEN FILERROR(DFILE1,1); <<CHECK FOR ERROR>>
00025000 00014 1
00026000 00014 1 DFILE2:=FOPEN(DATA2,%4,%4,128,,1); <<NEW FILE-DATATWO>>
00027000 00027 1 IF < THEN FILERROR(DFILE2,2); <<CHECK FOR ERROR>>
00028000 00033 1
00029000 00033 1 FWRITELABEL(DFILE2,LABL,9,0); <<FILE ID>>
00030000 00041 1 IF <> THEN FILERROR(DFILE2,3); <<CHECK FOR ERROR>>
00031000 00045 1
00032000 00045 1 FGETINFO(DFILE1,,,,,,,,,REC); <<LOCATE EOF>>
00033000 00053 1 IF < THEN FILERROR(DFILE1,4); <<CHECK FOR ERROR>>
00034000 00057 1
00035000 00057 1 INVERT'LOOP;
00036000 00057 1 DUMMY:=FREAD(DFILE1,BUFFER,128); <<OLD FILE RECORD>>
00037000 00065 1 IF < THEN FILERROR(DFILE1,5); <<CHECK FOR ERROR>>
00038000 00071 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00039000 00072 1
00040000 00072 1 REC:=REC-10; <<LAST REDC NO>>
00041000 00076 1 FWRITEDIR(DFILE2,BUFFER,128,REC); <<INVERT REC ORDER>>
00042000 00103 1 IF <> THEN FILERROR(DFILE2,6); <<CHECK FOR ERROR>>
00043000 00107 1
00044000 00107 1 GO INVERT'LOOP; <<CONTINUE OPERATION>>
00045000 00116 1
00046000 00116 1 END'OF'FILE;
00047000 00116 1 FCLOSE(DFILE2,2,0); <<SAVE NEW AS TEMP>>
00048000 00122 1 IF < THEN FILERROR(DFILE2,7); <<CHECK FOR ERROR>>
00049000 00126 1
00050000 00126 1 FCLOSE(DFILE1,4,0); <<DELETE OLD FILE>>
00051000 00132 1 IF < THEN FILERROR(DFILE1,8); <<CHECK FOR ERROR>>
00052000 00136 1 END.
PRIMARY DB STORAGE=%011; SECONDARY DB STORAGE=%00221
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0100104; ELAPSED TIME=0100159

```

Figure 3-10. Closing a New File as a Temporary File



A condition code of CCL is returned if the file is not closed successfully. The statement

```
IF < THEN FILEERROR(DFILE2,7);
```

checks the condition code and, if the condition code is CCL, the error-check procedure FILEERROR (see statements 13 through 19 in the program) is called.

The FILEERROR procedure calls the PRINT'FILE'INFO intrinsic, which prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned to FCLOSE.

The QUIT intrinsic call

```
QUIT(QUITNO);
```

aborts the process.

#### NOTE

The QUIT intrinsic causes MPE to close all files with no change. Thus, new files are deleted, old files are saved and assigned to the same domain to which they belonged previously.

### CLOSING A NEW FILE AS A PERMANENT FILE

Figure 3-11 contains an FCLOSE intrinsic call that closes a new file as a permanent file.

The FCLOSE intrinsic call

```
FCLOSE(OUT,%11,0);
```

closes the disc file specified by OUT. The parameters specified are

*filenum* Contained in the identifier OUT. The file number was assigned to OUT when the FOPEN intrinsic opened the file.

*disposition* %11, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	Binary
											1			1		Octal

The above bit pattern specifies the following:

Domain Disposition: Permanent file. The file is saved in the system domain. If the file is a new or old temporary file on disc, an entry is created for it in the

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INPUT(0:6):="INFILE "
00004000 00005 1 BYTE ARRAY DEV(0:4):="CARD "
00005000 00004 1 BYTE ARRAY OUTPUT(0:7):="DATAONE "
00006000 00005 1 ARRAY BUFFER(0:127)
00007000 00005 1 INTEGER IN,OUT,LGTH
00008000 00005 1
00009000 00005 1 INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT,FILE,INFO,QUIT
00010000 00005 1
00011000 00005 1 << END OF DECLARATIONS >>
00012000 00005 1
00013000 00005 1 IN:=FOPEN(INPUT,%5,.40,DEV) <<CARD READER>>
00014000 00012 1 IF < THEN <<CHECK FOR ERROR>>
00015000 00013 1 BEGIN
00016000 00013 2 PRINT,FILE,INFO(IN) <<PRINT ERROR>>
00017000 00015 2 QUIT(1) <<ABORT>>
00018000 00017 2 END
00019000 00017 1
00020000 00017 1 OUT:=FOPEN(OUTPUT,%4,%101,128) <<NEW DISC FILE>>
00021000 00030 1 IF < THEN <<CHECK FOR ERROR>>
00022000 00031 1 BEGIN
00023000 00031 2 PRINT,FILE,INFO(OUT) <<PRINT ERROR>>
00024000 00033 2 QUIT(2) <<ABORT>>
00025000 00035 2 END
00026000 00035 1 COPY,LOOP:
00027000 00035 1 LGTH:=FREAD(IN,BUFFER,40) <<READ A CARD>>
00028000 00035 1 IF < THEN <<CHECK FOR ERROR>>
00029000 00043 1 BEGIN
00030000 00044 1 PRINT,FILE,INFO(IN) <<PRINT ERROR>>
00031000 00044 2 PRINT,FILE,INFO(IN) <<PRINT ERROR>>
00032000 00046 2 QUIT(3) <<ABORT>>
00033000 00050 2 END
00034000 00050 1 IF > THEN GO END,OF,FILE <<CHECK FOR EOF>>
00035000 00051 1
00036000 00051 1 FWRITE(OUT,BUFFER,LGTH,0) <<COPY CARD TO DISC>>
00037000 00056 1 IF <> THEN <<CHECK FOR ERROR>>
00038000 00057 1 BEGIN
00039000 00057 2 PRINT,FILE,INFO(OUT) <<PRINT ERROR>>
00040000 00061 2 QUIT(4) <<ABORT>>
00041000 00063 2 END
00042000 00063 1 GO COPY,LOOP <<CONTINUE COPYING>>
00043000 00063 1
00044000 00066 1 END,OF,FILE:
00045000 00066 1 FCLOSE(OUT,%11,0) <<MAKE PERMANENT>>
00046000 00066 1 IF < THEN <<CHECK FOR ERROR>>
00047000 00072 1 BEGIN
00048000 00073 1 PRINT,FILE,INFO(OUT) <<PRINT ERROR>>
00049000 00073 2 PRINT,FILE,INFO(OUT) <<PRINT ERROR>>
00050000 00075 2 QUIT(5) <<ABORT>>
00051000 00077 2 END
00052000 00077 1 END,
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00213
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:44

```



Figure 3-11. Closing a New File as a Permanent File

system file directory. (An error code is returned if a file of the same name exists already in the system directory.) If it is an old permanent file on disc, this disposition value has no effect. If the file is stored on magnetic tape that tape is rewound and unloaded. Bits (13:3) = 001.

Disc Space Disposition: Unused disc space returned to the system. Bits (12:1) = 1.

*seccode* 0, unrestricted access.

A condition code of CCL is returned if the file is not closed successfully. The statement

```
IF < THEN
```

checks the condition code and, if it is CCL, the next four statements, starting with the BEGIN statement, are executed.

The statement

```
PRINT'FILE'INFO(OUT);
```

calls the PRINT'FILE'INFO intrinsic, which prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FCLOSE.

The QUIT intrinsic call

```
QUIT(5);
```

aborts the process.

## RENAMING A FILE

You can change the name of an open disc file with the FRENAME intrinsic call. This intrinsic effectively changes the actual designator (including lockword, if any) of the file. The file must be either:

1. A new file, or
2. An existing file to which you have write access.

When the FCLOSE intrinsic is called in figure 3-12, a check is made to determine if a file of the same name exists and, if one does exist, the FRENAME intrinsic is used to rename the file being closed.

The statement

```
FCLOSE(DFILE2,1,0);
```

attempts to close the file specified by DFILE2 as a permanent file. The file specified by the file number contained in DFILE2 is "DATATWO", which was opened as an old temporary file. If the file is closed successfully, a CCE condition code is returned and program control is transferred to

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA2(0:17):="DATATWO "
00004000 00005 1 BYTE ARRAY LISTFILE(0:8):="LISTFILE "
00005000 00006 1 BYTE ARRAY ALTNAME(0:7):="ALTDATA "
00006000 00005 1 ARRAY BUFFER(0:127)
00007000 00005 1 ARRAY MESSAGE(0:18):="DUPLICATE FILE NAME - FIX DURING BREAK"
00008000 00023 1 INTEGER DFILE2,LIST,ERROR
00009000 00023 1 DOUBLE REC:=0D
00010000 00023 1
00011000 00023 1 INTRINSIC FOPEN,FREADLABEL,FREADDIR,FWRITE,FCLOSE,FRENAME,
00012000 00023 1 FREADSEEK,CAUSEBREAK,FCHECK,PRINT'FILE'INFO,QUIT
00013000 00023 1
00014000 00023 1 PROCEDURE FILERROR(FILENO,QUITNO)
00015000 00000 1 VALUE QUITNO
00016000 00000 1 INTEGER FILENO,QUITNO
00017000 00000 1 BEGIN
00018000 00000 2 PRINT'FILE'INFO(QUITNO)
00019000 00002 2 QUIT(QUITNO)
00020000 00004 2 END
00021000 00000 1
00022000 00000 1 <<END OF DECLARATIONS>>
00023000 00000 1
00024000 00000 1 DFILE2:=FOPEN(DATA2,%6,%4,128) <<OLD TEMP FILE>>
00025000 00011 1 IF < THEN FILERROR(DFILE2,1) <<CHECK FOR ERROR>>
00026000 00015 1
00027000 00015 1 LIST:=FOPEN(LISTFILE,%14,%1) <<$STDLIST>>
00028000 00025 1 IF < THEN FILERROR(LIST,2) <<CHECK FOR ERROR>>
00029000 00031 1
00030000 00031 1 FREADLABEL (DFILE2,BUFFER,128,0) <<FILE ID>>
00031000 00037 1 IF <> THEN FILERROR(DFILE2,3) <<CHECK FOR ERROR>>
00032000 00043 1 FWRITE (LIST,BUFFER,9,0) <<DISPLAY ID>>
00033000 00050 1 IF <> THEN FILERROR(LIST,4) <<CHECK FOR ERROR>>
00034000 00054 1
00035000 00054 1 LIST*LOOP:
00036000 00054 1 FREADDIR (DFILE2,BUFFER,128,REC) <<EVERY OTHER RECD>>
00037000 00061 1 IF < THEN FILERROR(DFILE2,5) <<CHECK FOR ERROR>>
00038000 00065 1 IF > THEN GO END'OF'FILE <<CHECK FOR EOF>>
00039000 00066 1
00040000 00066 1 REC:=REC*2D <<EVERY OTHER RECD>>
00041000 00072 1 FREADSEEK (DFILE2,REC) <<FILL SYSTEM BUFFER>>
00042000 00075 1 IF < THEN FILERROR(DFILE2,5) <<CHECK FOR ERROR>>
00043000 00101 1
00044000 00101 1 FWRITE (LIST,BUFFER,35,0) <<ALTERNATE RECORDS>>
00045000 00106 1 IF <> THEN FILERROR(LIST,7) <<CHECK FOR ERROR>>
00046000 00112 1
00047000 00112 1 GO LIST*LOOP; <<CONTINUE LISTING>>
00048000 00117 1
00049000 00117 1 END'OF'FILE:
00050000 00117 1 FCLOSE (DFILE2,1,0) <<MAKE PERMANENT>>
00051000 00123 1 IF = THEN GO DONE <<LISTING DONE>>
00052000 00124 1 FCHECK (DFILE2,ERROR) <<FCLOSE ERROR>>
00053000 00131 1 IF ERROR=100 THEN <<DUPLICATE FILE NAME>>
00054000 00134 1 BEGIN
00055000 00134 2 FRENAME (DFILE2,ALTNAME) <<CHANGE FILE NAME>>
00056000 00137 2 CLOSE:
00057000 00137 2 FCLOSE (DFILE2,1,0) <<TRY AGAIN>>
00058000 00143 2 IF = THEN GO DONE <<GOOD FCLOSE>>
00059000 00144 2 PRINT'FILE'INFO (DFILE2) <<PRINT ERROR>>
00060000 00146 2 FWRITE (LIST,MESSAGE,19,0) <<SEEK HELP>>
00061000 00153 2 CAUSEBREAK <<SESSION BREAK>>
00062000 00154 2 GO CLOSE <<LOOP BACK>>
00063000 00155 2 END
00064000 00155 1 DONE;END.
PRIMARY DB STORAGE=%012; SECONDARY DB STORAGE=%00240
NO. ERRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:04; ELAPSED TIME=0:00:58

```

Figure 3-12. FRENAME Intrinsic Example

statement label DONE, terminating program execution. If CCE is not returned, the FCHECK intrinsic is called to determine the error number. The statement

```
IF ERROR=100 THEN
```

checks whether the error number is 100 (duplicate file name in the system file directory). Note that even though the file DATATWO was opened successfully from the job temporary file directory, it is possible that some other user already has a permanent file named DATATWO in the system file directory, hence an error to this effect will be returned when the program attempts to close a job temporary file as a permanent file.

The statement

```
FRENAME(DFILE2,ALTNAME);
```

attempts to rename the file to the actual designator (ALTDATA) contained in the byte array ALTNAME. The second FCLOSE call then attempts to close the file under this new name.

If the second FCLOSE call fails, the PRINT'FILE'INFO intrinsic causes a FILE INFORMATION DISPLAY to be printed on the standard list device. In addition, the statement

```
FWRITE(LIST,MESSAGE,19,0);
```

prints the message

```
DUPLICATE FILE NAME — FIX DURING BREAK
```

and the CAUSEBREAK intrinsic call causes a session break.

MPE prompts with a colon on the terminal and now you can enter MPE commands.

#### NOTE

The :RENAME command can be used to rename a file. However, this command cannot be used to rename a file that is currently open in a program. For example, if a file of the alternate name (ALTDATA) also exists in the system file directory, the :RENAME command must be used to rename *this* file instead of the file opened by the program. Thus, a :RENAME command of the form

```
:RENAME ALTDATA,ALTAAAA,TEMP
```

(attempting to rename the *opened* temporary file ALTDATA) will result in the error message

```
EXCLUSIVE VIOLATION
```

The :RENAME command must be used to rename the old, *unopened* file in the system directory, as follows:

```
:RENAME ALTDATA,ALTAAAA
```

See the *MPE Commands Reference Manual* for a further discussion of the :RENAME command.

Once :RESUME is typed to resume program execution, the statement

```
GO CLOSE;
```

transfers program control back to the label CLOSE and the FCLOSE sequence is tried again.

## WRITING A FILE SYSTEM ERROR-CHECK PROCEDURE

As you noticed in some of the examples, the statements

```
BEGIN
  PRINT'FILE'INFO(filenum);
  QUIT(num);
END;
```

were repeated after each intrinsic call. Instead of repeating this code throughout a program with multiple intrinsic calls, however, it is more efficient (because less code is generated) to write an error-check procedure and merely call this procedure where necessary in a program.

Figure 3-13 contains a program which includes an error-check procedure, and a single statement calls this procedure if an error occurs. The program opens a card reader and a disc file, reads the card file, writes these records into the disc file, then closes the disc file.

The error check procedure (statements 10 through 17 in figure 3-13) contains two parameters: FILENO (integer) and QUITNO (integer by value). FILENO is an identifier through which is passed the file number. This file number is used by PRINT'FILE'INFO to print a FILE INFORMATION DISPLAY for that file.

The QUIT intrinsic aborts the program's process and prints the QUITNO as part of the abort message, enabling you to determine the point at which the process was aborted.

## READING A FILE IN SEQUENTIAL ORDER

To read records, or portions of records, from a file in sequential order, you use the FREAD intrinsic.

When the FREAD intrinsic executes, a logical record pointer advances to the next record. Then, the next time the FREAD intrinsic is called, the next record is read. Even if a portion of a record is read, a subsequent FREAD ignores the unread portion of the last record (because the logical record pointer has advanced) and begins reading the next record.

### NOTE

The logical record pointer is a number kept by MPE to indicate the next sequential record to be accessed in a file.

The program shown in figure 3-14 reads a card file. The FREAD statement

```
LGTH:=FREAD(IN,BUFFER,40);
```

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INPUT(0:6):="INFILE ";
00004000 00005 1 BYTE ARRAY DEV(0:4):="CARD ";
00005000 00004 1 BYTE ARRAY OUTPUT(0:7):="DATAONE ";
00006000 00005 1 ARRAY BUFFER(0:127);
00007000 00005 1 INTEGER IN,OUT,LGTH;
00008000 00005 1
00009000 00005 1 INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT,FILE,INFO,QUIT;
00010000 00005 1
00011000 00005 1 PROCEDURE FILEERROR(FILENO,QUITNO);
00012000 00000 1 VALUE QUITNO;
00013000 00000 1 INTEGER FILENO,QUITNO;
00014000 00000 1 BEGIN
00015000 00000 2 PRINT,FILE,INFO(FILENO);
00016000 00002 2 QUIT(QUITNO);
00017000 00004 2 END;
00018000 00000 1
00019000 00000 1 << END OF DECLARATIONS >>
00020000 00000 1
00021000 00000 1 IN:=FOPEN(INPUT,%5,,40,DEV); <<CARD READER>>
00022000 00012 1 IF < THEN FILEERROR(IN,1); <<CHECK FOR ERROR>>
00023000 00016 1
00024000 00016 1 OUT:=FOPEN(OUTPUT,%4,%101,128); <<NEW DISC FILE>>
00025000 00027 1 IF < THEN FILEERROR(OUT,2); <<CHECK FOR ERROR>>
00026000 00033 1
00027000 00033 1 COPY'LOOP:
00028000 00033 1 LGTH:=FREAD(IN,BUFFER,40); <<READ A CARD>>
00029000 00041 1 IF < THEN FILEERROR(IN,3); <<CHECK FOR ERROR>>
00030000 00045 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00031000 00046 1
00032000 00046 1 FWRITE(OUT,BUFFER,LGTH,0); <<COPY CARD TO DISC>>
00033000 00053 1 IF <> THEN FILEERROR(OUT,4); <<CHECK FOR ERROR>>
00034000 00057 1
00035000 00057 1 GO COPY'LOOP; <<CONTINUE COPYING>>
00036000 00062 1
00037000 00062 1 END'OF'FILE:
00038000 00062 1 FCLOSE(OUT,%11,0); <<MAKE PERMANENT>>
00039000 00066 1 IF < THEN FILEERROR(OUT,5); <<CHECK FOR ERROR>>
00040000 00072 1 END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00213
NO. ERRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:03; ELAPSED TIME=0:00:31

```

Figure 3-13. Error-Check Procedure Example

```

00001000 00000 0  $CONTROL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:6):="INFILE "
00004000 00005 1  BYTE ARRAY DEV(0:4):="CARD "
00005000 00004 1  BYTE ARRAY OUTPUT(0:7):="DATAONE "
00006000 00005 1  ARRAY BUFFER(0:127)
00007000 00005 1  INTEGER IN,OUT,LGTH
00008000 00005 1
00009000 00005 1  INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT
00010000 00005 1
00011000 00005 1  << END OF DECLARATIONS >>
00012000 00005 1
00013000 00005 1  IN:=FOPEN(INPUT,%5,,40,DEV)
00014000 00012 1  IF < THEN
00015000 00013 1  BEGIN
00016000 00013 2  PRINT'FILE'INFO(IN)
00017000 00015 2  QUIT(1)
00018000 00017 2  END
00019000 00017 1  OUT:=FOPEN(OUTPUT,%4,%101,128)
00020000 00017 1  IF < THEN
00021000 00030 1  BEGIN
00022000 00031 1  PRINT'FILE'INFO(OUT)
00023000 00031 2  QUIT(2)
00024000 00033 2  END
00025000 00035 2  COPY'LOOP:
00026000 00035 1  LGTH:=FREAD(IN,BUFFER,40)
00027000 00035 1  IF < THEN
00028000 00043 1  BEGIN
00029000 00044 1  PRINT'FILE'INFO(IN)
00030000 00044 2  QUIT(3)
00031000 00046 2  END
00032000 00050 2  IF > THEN GO END'OF'FILE
00033000 00050 1  FWRITE(OUT,BUFFER,LGTH,0)
00034000 00051 1  IF <> THEN
00035000 00056 1  BEGIN
00036000 00057 1  PRINT'FILE'INFO(OUT)
00037000 00057 2  QUIT(4)
00038000 00063 2  END
00039000 00063 1  GO COPY'LOOP
00040000 00063 1  END'OF'FILE
00041000 00066 1  FCLOSE(OUT,%11,0)
00042000 00066 1  IF < THEN
00043000 00072 1  BEGIN
00044000 00073 1  PRINT'FILE'INFO(OUT)
00045000 00073 2  QUIT(5)
00046000 00075 2  END
00047000 00077 2  END.
00048000 00077 1  PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00213
00049000 00077 1  NO. ERRORS=000; NO. WARNINGS=000
00050000 00077 1  PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:44

```

Figure 3-14. FREAD and FWRITE Intrinsic Example



reads a record from the card reader file designated by the variable IN (the file number was assigned to IN when the FOPEN intrinsic opened the file) and transfers this record to the array BUFFER in the stack. The statement reads up to 40 words from the record, then returns a positive value to LGTH which indicates the actual length of the information transferred.

If an error occurs during execution of the FREAD intrinsic, a condition code of CCL is returned. The statement

```
IF < THEN
```

checks the condition code and, if the condition code is CCL, the next four statements are executed. The PRINT'FILE'INFO intrinsic call causes a FILE INFORMATION DISPLAY to be printed on the output device so that you can determine the error number returned by FREAD, and the QUIT intrinsic aborts the process.

When the end-of-file is encountered on the card file, a condition code of CCG is returned. The statement

```
IF > THEN GO END'OF'FILE;
```

checks for this condition code and, when it occurs, transfers program control to the label END'OF'FILE. If the end-of-file condition is not encountered, the FWRITE statement is executed and the

```
GO COPY'LOOP;
```

statement transfers program control back to the beginning of the copy loop. The FREAD intrinsic is called again and the next record is read.

## WRITING RECORDS INTO A FILE IN A SEQUENTIAL ORDER

To write records, or portions of records, from your buffer to a file in sequential order, you use the FWRITE intrinsic.

When the FWRITE intrinsic executes, the logical record pointer advances to the next record. Then, the next time the FWRITE intrinsic is called, information is written into the next record position. When information is written to a file composed of fixed-length records (and buffering is not specified in the FOPEN call), the file system pads all short records with binary zeros for a binary file, or ASCII blanks for an ASCII file to bring the records up to the fixed length required. If *nobuff* was specified in FOPEN, automatic buffering is not provided by MPE.

The FWRITE statement in figure 3-14.

```
FWRITE(OUT,BUFFER,LGTH,0);
```

writes a record from the array BUFFER into the disc file designated by the variable OUT. (The file number was assigned to OUT when FOPEN opened the file.) The length of the record is specified by LGTH. (LGTH was assigned its value when FREAD read the record and transferred it to BUFFER, so in this case the same number of words being read from the card reader are being written to the disc.)

The *control* parameter is specified as 0 to indicate that no carriage control code is included in the record. (Carriage control, of course, is not necessary for a disc file but the parameter is included because all FWRITE parameters are required.)

A condition code of CCE signifies that the FWRITE request was granted. The statement

```
IF <> THEN
```

checks for a “not equal” condition code and, if CCG or CCL is returned, the next four statements are executed. The PRINT'FILE'INFO intrinsic causes a FILE INFORMATION DISPLAY to be printed on the output device, enabling you to determine the error number returned by FWRITE. The QUIT intrinsic aborts the process.

If CCE is returned, the next four statements are not executed, the GO COPY'LOOP statement is executed, and the FREAD and FWRITE intrinsic calls are repeated until FREAD detects the end of the card file.

## READING A FILE IN DIRECT-ACCESS MODE

As you recall from the discussion of the FREAD intrinsic, a record read with that intrinsic is determined by the position of the logical record pointer. Each successive FREAD then reads the next record in sequence because the logical record pointer advances one record each time FREAD is executed. It is possible, however, to access specific records in a disc file with the FREADDIR intrinsic. The *record number* to be read is specified as one of the parameters in the FREADDIR intrinsic call. Note that the FREADDIR intrinsic call may be issued only for a disc file composed of fixed-length or undefined-length records.

Figure 3-15 contains a program that reads every other record in a disc file using the FREADDIR intrinsic. The FREADDIR intrinsic call

```
FREADDIR(DFILE2,BUFFER,128,REC);
```

reads a record from the file designated by DFILE2 (the file number was assigned to DFILE2 when the FOPEN intrinsic opened the file) and transfers this record to the array BUFFER in the stack. Up to 128 words are read from the record. The parameter REC specifies which record is read. The double integer value OD (double integers are indicated by the suffix D in SPL) was assigned to REC (see statement number 9 in the program), and so the first time the LIST'LOOP is executed, the first record in the file (logical record number 0) is read. REC is incremented by 2D each time the loop is executed, therefore, physical record number 3 (logical record number 2) is read the second time the loop is executed, then 5, 7, etc. The logical record pointer is advanced by one each time the FREADDIR intrinsic is executed. Since the record number to be read is specified by REC, however, the FREADDIR intrinsic does not necessarily read records in sequential order, as does the FREAD intrinsic.

If the information is not read successfully by a FREADDIR intrinsic call, a CCL condition code is returned. The statement

```
IF < THEN FILERROR(DFILE2,3);
```

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00004000 00005 1 BYTE ARRAY LISTFILE(0:8):="LISTFILE ";
00005000 00006 1 BYTE ARRAY ALTNAME(0:7):="ALTDATA ";
00006000 00005 1 ARRAY BUFFER(0:127);
00007000 00005 1 ARRAY MESSAGE(0:18):="DUPLICATE FILE NAME - FIX DURING BREAK";
00008000 00023 1 INTEGER DFILE2,LIST,ERROR;
00009000 00023 1 DOUBLE REC:=0D;
00010000 00023 1
00011000 00023 1 INTRINSIC FOPEN,FREADLABEL,FREADDIR,FWRITE,FCLOSE,FRENAME,
00012000 00023 1 FREADSEEK,CAUSEBREAK,FCHECK,PRINT'FILE'INFO,QUIT;
00013000 00023 1
00014000 00023 1 PROCEDURE FILEERROR(FILENO,QUITNO);
00015000 00000 1 VALUE QUITNO;
00016000 00000 1 INTEGER FILENO,QUITNO;
00017000 00000 1 BEGIN
00018000 00000 2 PRINT'FILE'INFO(QUITNO);
00019000 00002 2 QUIT(QUITNO);
00020000 00004 2 END;
00021000 00000 1
00022000 00000 1 <<END OF DECLARATIONS>>
00023000 00000 1
00024000 00000 1 DFILE2:=FOPEN(DATA2,%6,%4,128); <<OLD TEMP FILE>>
00025000 00011 1 IF < THEN FILEERROR(DFILE2,1); <<CHECK FOR ERROR>>
00026000 00015 1
00027000 00015 1 LIST:=FOPEN(LISTFILE,%14,%1); <<$STDLIST>>
00028000 00025 1 IF < THEN FILEERROR(LIST,2); <<CHECK FOR ERROR>>
00029000 00031 1
00030000 00031 1 FREADLABEL(DFILE2,BUFFER,128,0); <<FILE ID>>
00031000 00037 1 IF <> THEN FILEERROR(DFILE2,3); <<CHECK FOR ERROR>>
00032000 00043 1 FWRITE(LIST,BUFFER,9,0); <<DISPLAY ID>>
00033000 00050 1 IF <> THEN FILEERROR(LIST,4); <<CHECK FOR ERROR>>
00034000 00054 1
00035000 00054 1 LIST'LOOP:
00036000 00054 1 FREADDIR(DFILE2,BUFFER,128,REC); <<EVERY OTHER RECD>>
00037000 00061 1 IF < THEN FILEERROR(DFILE2,5); <<CHECK FOR ERROR>>
00038000 00065 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00039000 00066 1
00040000 00066 1 REC:=REC+2D; <<EVERY OTHER RECD>>
00041000 00072 1 FREADSEEK(DFILE2,REC); <<FILL SYSTEM BUFFER>>
00042000 00075 1 IF < THEN FILEERROR(DFILE2,6); <<CHECK FOR ERROR>>
00043000 00101 1
00044000 00101 1 FWRITE(LIST,BUFFER,35,0); <<ALTERNATE RECORDS>>
00045000 00106 1 IF <> THEN FILEERROR(LIST,7); <<CHECK FOR ERROR>>
00046000 00112 1
00047000 00112 1 GO LIST'LOOP; <<CONTINUE LISTING>>
00048000 00117 1
00049000 00117 1 END'OF'FILE;
00050000 00117 1 FCLOSE(DFILE2,1,0); <<MAKE PERMANENT>>
00051000 00123 1 IF = THEN GO DONE; <<LISTING DONE>>
00052000 00124 1 FCHECK(DFILE2,ERROR); <<FCLOSE ERROR>>
00053000 00131 1 IF ERROR=100 THEN <<DUPLICATE FILE NAME>>
00054000 00134 1 BEGIN
00055000 00134 2 FRENAME(DFILE2,ALTNAME); <<CHANGE FILE NAME>>
00056000 00137 2 CLOSE:
00057000 00137 2 FCLOSE(DFILE2,1,0); <<TRY AGAIN>>
00058000 00143 2 IF = THEN GO DONE; <<GOOD FCLOSE>>
00059000 00144 2 PRINT'FILE'INFO(DFILE2); <<PRINT ERROR>>
00060000 00146 2 FWRITE(LIST,MESSAGE,19,0); <<SEEK HELP>>
00061000 00153 2 CAUSEBREAK; <<SESSION BREAK>>
00062000 00154 2 GO CLOSE; <<LOOP BACK>>
00063000 00155 2 END;
00064000 00155 1 DONE;END,
PRIMARY DB STORAGE=%012; SECONDARY DB STORAGE=%00240
NO. FRRRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:04; ELAPSED TIME=0:00:58

```

Figure 3-15. FREADDIR and FREADSEEK Intrinsic Example

checks the condition code and, if it is CCL, calls the error-check procedure `FILERROR`. The `FILERROR` procedure prints a `FILE INFORMATION DISPLAY` on the standard list device, enabling you to determine the error number returned by `FREADDIR`, then aborts the program's process.

A condition code of CCG signifies an end-of-file condition and the statement

```
IF > THEN GO END'OF'FILE;
```

transfers program control to the label `END'OF'FILE` when the end-of-file condition is encountered.



## OPTIMIZING DIRECT-ACCESS FILE READING

If you know in advance that a certain record is to be read from a file with the `FREADDIR` intrinsic, you can speed up the I/O process by issuing a `FREADSEEK` intrinsic call.

The `FREADSEEK` intrinsic moves the record from the file to a system buffer. Then, when the `FREADDIR` intrinsic call is issued, the record is transferred from this system buffer to the buffer in the stack specified by `FREADDIR`. The I/O process is enhanced when `FREADSEEK` is used, because the system buffer already contains the record to be read before the `FREADDIR` call is issued.

The `LIST'LOOP` in figure 3-15 performs the following functions:

1. Issues a `FREADDIR` intrinsic call to transfer a record (specified by `REC`) from a file (specified by `DFILE2`) to an array (`BUFFER`) in the stack.
2. Increments `REC` by 2D.
3. Issues a `FREADSEEK` intrinsic call to read the record specified by the new value of `REC` and to transfer this record to a system buffer.
4. Lists the record in the stack array (`BUFFER`) on the standard list device.
5. Repeats the loop.

The next time `LIST'LOOP` is executed, the `FREADDIR` intrinsic reads the record from the file system buffer to the stack array (`BUFFER`), eliminating the need for file access and thus reducing the execution time of the loop.

## WRITING RECORDS INTO A FILE IN DIRECT-ACCESS MODE

To write information into a specific record in a disc file, you use the `FWRITEDIR` intrinsic.

Unlike the `FWRITE` intrinsic, which writes records into a file depending on the position of the logical record pointer, the `FWRITEDIR` intrinsic can write into any record of a file by specifying the logical record number as a parameter.

The `FWRITEDIR` intrinsic call may be issued only for disc files of fixed-length or undefined-length records.

Figure 3-16 contains a program that reads records from one file and writes these records, in inverse order, into a second file using the FWRITEDIR intrinsic.

The FGETINFO intrinsic (see page 3-63) is used to locate the end-of-file in the file to be read. This information is returned to the variable REC.

The FREAD statement

```
DUMMY:=FREAD(DFILE1,BUFFER,128);
```

reads up to 128 words from the first record of the file DATAONE (specified by the file number assigned to DFILE1 by the FOPEN intrinsic when the file was opened) and transfers this information to the array BUFFER.

The statement

```
REC:=REC-1D;
```

decrements REC by the double integer value 1D to arrive at the logical record number of the last record in the file. (REC contains a current value of last physical record (*last logical record + 1D*) as a result of the FGETINFO intrinsic call.)

The FWRITEDIR statement

```
FWRITEDIR(DFILE2,BUFFER,128,REC);
```

writes the record contained in the array BUFFER to the file specified by DFILE2. The parameters specified in the FWRITEDIR intrinsic call are

<i>filenum</i>	Contained in DFILE2, which was assigned the file number of the file DATATWO when the FOPEN intrinsic opened the file.
<i>target</i>	BUFFER, the array that contains the record to be written.
<i>tcount</i>	128 words
<i>recnum</i>	REC, which contains the logical record number of the last record in the file.

If the FWRITEDIR request is successful, a CCE condition code is returned. The statement

```
IF <> THEN FILEERROR(DFILE2,6);
```

checks for a “not equal” condition code and, if such a condition code is returned, the error-check procedure FILEERROR is called.

The FILEERROR procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FWRITEDIR, then aborts the program’s process.

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1   BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000 00005 1   BYTE ARRAY DATA2(0:7):="DATATWO ";
00005000 00005 1   ARRAY LABL(0:8):="EMPLOYEE DATA FILE";
00006000 00011 1   ARRAY BUFFER(0:127);
00007000 00011 1   INTEGER DFILE1,DFILE2,DUMMY;
00008000 00011 1   DOUBLE REC;
00009000 00011 1
00010000 00011 1   INTRINSIC FOPEN,FWRITELABEL,FGETINFO,FREAD,FWRITEDIR,FCLOSE,
00011000 00011 1   PRINT'FILE'INFO,QUIT;
00012000 00011 1
00013000 00011 1   PROCEDURE FILEERROR(FILENO,QUITNO);
00014000 00000 1   VALUE QUITNO;
00015000 00000 1   INTEGER FILENO,QUITNO;
00016000 00000 1   BEGIN
00017000 00000 2   PRINT'FILE'INFO(FILENO);
00018000 00002 2   QUIT(QUITNO);
00019000 00004 2   END;
00020000 00000 1
00021000 00000 1   <<END OF DECLARATIONS>>
00022000 00000 1
00023000 00000 1   DFILE1:=FOPEN(DATA1,%5,%100);           <<OLD FILE-DATAONE>>
00024000 00010 1   IF < THEN FILEERROR(DFILE1,1);       <<CHECK FOR ERROR>>
00025000 00014 1
00026000 00014 1   DFILE2:=FOPEN(DATA2,%4,%4,128,,1);       <<NEW FILE-DATATWO>>
00027000 00027 1   IF < THEN FILEERROR(DFILE2,2);       <<CHECK FOR ERROR>>
00028000 00033 1
00029000 00033 1   FWRITELABEL(DFILE2,LABL,9,0);           <<FILE ID>>
00030000 00041 1   IF <> THEN FILEERROR(DFILE2,3);       <<CHECK FOR ERROR>>
00031000 00045 1
00032000 00045 1   FGETINFO(DFILE1,,,,,,,,,REC);           <<LOCATE EOF>>
00033000 00053 1   IF < THEN FILEERROR(DFILE1,4);       <<CHECK FOR ERROR>>
00034000 00057 1
00035000 00057 1   INVERT'LOOP;
00036000 00057 1   DUMMY:=FREAD(DFILE1,BUFFER,128);       <<OLD FILE RECORD>>
00037000 00065 1   IF < THEN FILEERROR(DFILE1,5);       <<CHECK FOR ERROR>>
00038000 00071 1   IF > THEN GO END'OF'FILE;           <<CHECK FOR EOF>>
00039000 00072 1
00040000 00072 1   REC:=REC-1;                               <<LAST REDC NO>>
00041000 00076 1   FWRITEDIR(DFILE2,BUFFER,128,REC);       <<INVERT REC ORDER>>
00042000 00103 1   IF <> THEN FILEERROR(DFILE2,6);       <<CHECK FOR ERROR>>
00043000 00107 1
00044000 00107 1   GO INVERT'LOOP;                               <<CONTINUE OPERATION>>
00045000 00116 1
00046000 00116 1   END'OF'FILE;
00047000 00116 1   FCLOSE(DFILE2,2,0);                               <<SAVE NEW AS TEMP>>
00048000 00122 1   IF < THEN FILEERROR(DFILE2,7);       <<CHECK FOR ERROR>>
00049000 00126 1
00050000 00126 1   FCLOSE(DFILE1,4,0);                               <<DELETE OLD FILE>>
00051000 00132 1   IF < THEN FILEERROR(DFILE1,8);       <<CHECK FOR ERROR>>
00052000 00136 1   END.
PRIMARY DB STORAGE=%011;   SECONDARY DB STORAGE=%00221
NO. ERRORS=000;           NO. WARNINGS=000
PROCESSOR TIME=0:100:04;   ELAPSED TIME=0:00:59

```

Figure 3-16. FWRITEDIR Intrinsic Example

If a condition code of CCE is returned, the

```
IF <> THEN FILEERROR(DFILE2,6);
```

statement is not executed and the

```
GO INVERT'LOOP;
```

statement transfers program control to the statement label INVERT'LOOP, causing the invert loop to be repeated.

The second time the loop is executed, the FREAD intrinsic reads the second record from DATAONE and the FWRITE intrinsic writes this record into the next-to-last record in DATATWO (REC has been decremented again by 1D). The loop repeats until the last record is read from DATAONE.

## LOCKING AND UNLOCKING FILES

Occasionally, for example, when a file is opened so that records can be changed, it is advantageous to dynamically *lock* the file to ensure that another user does not attempt to change the same record at the same time. This is accomplished with the FLOCK intrinsic; a locked file is unlocked with the FUNLOCK intrinsic.

When an FOPEN intrinsic specifying the dynamic locking *option* is issued against a disc file, a Resource Identification Number (RIN) is established for that file. A user's process then can call intrinsics that dynamically lock and unlock the file by alternately acquiring and releasing exclusive use of this RIN. When a file resides on a unit-record device, locking the file (RIN) is equivalent to locking the device itself.

Because the RIN's used in dynamic file locking are global RIN's, the user employing the file-locking intrinsics must follow the rules governing global RIN's (see Section VI). Specific capability-class rules governing file locking are:

1. *Standard Capabilities.* A user's running process (program) can lock only one file at a time.
2. *Process-Handling Optional Capability.* Within the job process structure, only one file can be locked at any one time.
3. *Multiple RIN Optional Capability.* No restrictions are imposed.

If several processes of the same job are allowed access to a file in an exclusive mode, *local* (as opposed to global) RIN's are available without limitation.

Figure 3-17 contains a program that updates the file DATAONE. The FLOCK intrinsic call

```
FLOCK(DFILE1,1);
```

locks the file. The parameters specified in the intrinsic call are

<i>filenum</i>	Contained in DFILE1, which was assigned the file number of DATAONE when the FOPEN intrinsic opened the file.
----------------	--

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA1(0:7):="DATAONE "
00004000 00005 1 ARRAY BUFFER(0:127);
00005000 00005 1 INTEGER DFILE1, LGTH, DUMMY, IN, LIST;
00006000 00005 1
00007000 00005 1 INTRINSIC FOPEN, FREAD, FUPDATE, FLOCK, FUNLOCK, FCLOSE,
00008000 00005 1 PRINT'FILE'INFO, QUIT, FWRITE, FREAD;
00009000 00005 1
00010000 00005 1 PROCEDURE FILERERROR(FILENO, QUITNO);
00011000 00000 1 VALUE QUITNO;
00012000 00000 1 INTEGER FILENO, QUITNO;
00013000 00000 1 BEGIN
00014000 00000 2 PRINT'FILE'INFO(FILENO);
00015000 00002 2 QUIT(QUITNO);
00016000 00004 2 END;
00017000 00000 1
00018000 00000 1 <<END OF DECLARATIONS>>
00019000 00000 1
00020000 00000 1 DFILE1:=FOPEN(DATA1,%5,%345,128); <<OLD DISC FILE>>
00021000 00011 1 IF < THEN FILERERROR(DFILE1,1); <<CHECK FOR ERROR>>
00022000 00015 1
00023000 00015 1 IN:=FOPEN(,%244); <<$STDIN>>
00024000 00024 1 IF < THEN FILERERROR(IN,2); <<CHECK FOR ERROR>>
00025000 00030 1
00026000 00030 1 LIST:=FOPEN(,%614,%1); <<$STDLIST>>
00027000 00040 1 IF < THEN FILERERROR(LIST,3); <<CHECK FOR ERROR>>
00028000 00044 1
00029000 00044 1 UPDATE'LOOP:
00030000 00044 1 FLOCK(DFILE1,1); <<LOCK FILE/SUSPEND>>
00031000 00047 1 IF < THEN FILERERROR(DFILE1,4); <<CHECK FOR ERROR>>
00032000 00053 1
00033000 00053 1 LGTH:=FREAD(DFILE1,BUFFER,128); <<GET EMPLOYEE RECD>>
00034000 00061 1 IF < THEN FILERERROR(DFILE1,5); <<CHECK FOR ERROR>>
00035000 00065 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00036000 00070 1
00037000 00070 1 FWRITE(LIST,BUFFER,-20,%320); <<EMPLOYEE NAME>>
00038000 00075 1 IF <> THEN FILERERROR(LIST,6); <<CHECK FOR ERROR>>
00039000 00101 1
00040000 00101 1 DUMMY:=FREAD(IN,BUFFER(30),5); <<EMPLOYEE NUMBER>>
00041000 00110 1 IF < THEN FILERERROR(IN,7); <<CHECK FOR ERROR>>
00042000 00114 1 IF > THEN GO END'OF'FILE;
00043000 00115 1
00044000 00115 1 FUPDATE(DFILE1,BUFFER,128); <<EMPLOYEE RECORD>>
00045000 00121 1 IF <> THEN FILERERROR(DFILE1,8); <<CHECK FOR ERROR>>
00046000 00125 1
00047000 00125 1 FUNLOCK(DFILE1); <<ALLOW OTHER ACCESS>>
00048000 00127 1 IF <> THEN FILERERROR(DFILE1,9); <<CHECK FOR ERROR>>
00049000 00133 1
00050000 00133 1 GO UPDATE'LOOP; <<CONTINUE UPDATE>>
00051000 00140 1
00052000 00140 1 END'OF'FILE:
00053000 00140 1 FUNLOCK(DFILE1); <<ALLOW OTHER ACCESS>>
00054000 00142 1 IF <> THEN FILERERROR(DFILE1,10); <<CHECK FOR ERROR>>
00055000 00146 1
00056000 00146 1 FCLOSE(DFILE1,0,0); <<DISP=NO CHANGE>>
00057000 00151 1 IF < THEN FILERERROR(DFILE1,11); <<CHECK FOR ERROR>>
00058000 00155 1 END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00204
NO. ERRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:03; ELAPSED TIME=0:00:17

```

Figure 3-17. FLOCK and FUNLOCK Intrinsic Example



*lockcond*

1, which specifies that the file is to be locked *unconditionally*. This means that if the file cannot be locked immediately, the calling process is suspended until the file can be locked.

A condition code of CCL is returned if the FLOCK request was not granted. The statement

```
IF < THEN FILERROR(DFILE1,4);
```

checks for a condition code of CCL and, if it is returned, the error-check procedure FILERROR is called. The FILERROR procedure prints a FILE INFORMATION DISPLAY on the standard output device, enabling you to determine the error number returned by FLOCK, then aborts the program's process.

The statements below perform the following:

```
LGTH:=FREAD(DFILE1,BUFFER,128);      <<GET EMPLOYEE RECD>>
IF < THEN FILERROR(DFILE1,5);        <<CHECK FOR ERROR>>
IF > THEN GO END'OF'FILE;            <<CHECK FOR EOF>>

FWRITE(LIST,BUFFER,-20,%320);        <<EMPLOYEE NAME>>
IF <> THEN FILERROR(LIST,6);        <<CHECK FOR ERROR>>

DUMMY:=FREAD(IN,BUFFER(30),5);       <<EMPLOYEE NUMBER>>
IF < THEN FILERROR(IN,7);           <<CHECK FOR ERROR>>
IF > THEN GO END'OF'FILE;

FUPDATE(DFILE1,BUFFER,128);          <<EMPLOYEE RECORD>>
IF <> THEN FILERROR(DFILE1,8);      <<CHECK FOR ERROR>>
```

1. Read a record from the file DATAONE.
2. Print 20 bytes of this record (employee name) on the standard list device (a terminal in this case; the program was run interactively).
3. Read an employee number from the terminal into the array BUFFER starting at word 30.
4. Update the record by writing the information contained in BUFFER, including the employee number, into file DATAONE.

The statement

```
FUNLOCK(DFILE1);
```

unlocks the file DATAONE (the file number of which is specified by DFILE1), thus allowing other users to access the file. Note that this statement follows each update in UPDATE'LOOP and is repeated in END'OF'FILE to insure that the file is unlocked in case an end-of-file condition causes a branch out of UPDATE'LOOP before the file is unlocked.

## UPDATING A FILE

To update a logical record of a disc file, you use the FUPDATE intrinsic.

The FUPDATE intrinsic affects the *last* logical record (or block for NOBUF files) accessed by any intrinsic call for the file named, and writes information from a buffer in the stack into this record. Note that the record number need not be supplied in the FUPDATE intrinsic call; FUPDATE automatically updates the *last* record referenced in any intrinsic call.

The file containing the record to be updated must have been opened with the update *option* specified in the FOPEN call and must not contain variable-length records.

Figure 3-18 contains a program that opens an old disc file and updates records in the file. The update information (employee number) is entered from a terminal (the program was run interactively) into a buffer in the stack, then the contents of the buffer are used to update the record.

The statement

```
LGTH:=FREAD(DFILE1,BUFFER,128);
```

reads an employee record from the file specified by DFILE1 into the array BUFFER in the stack.

The statement

```
FWRITE(LIST,BUFFER,-20,%320);
```

then displays this record on the terminal (\$STDLIST has been opened with the FOPEN intrinsic and the resulting file number was assigned to LIST).

The statement

```
DUMMY:=FREAD(IN,BUFFER(30),5);
```

reads an employee number, entered on the terminal (\$STDIN has been opened with the FOPEN intrinsic and the resulting file number was assigned to IN), into the array BUFFER starting at word 30.

The statement

```
FUPDATE(DFILE1,BUFFER,128);
```

then calls the FUPDATE intrinsic to update the last record accessed in the file specified by DFILE1. The contents of BUFFER (including the employee number entered from the terminal) are written into this record. Up to 128 words are written.

If the FUPDATE request was granted, a CCE condition code results. The statement

```
IF < > THEN FILERROR(DFILE1,9);
```

checks for a “not equal” condition code and, if such is the case, calls the error-check procedure FILERROR. The procedure FILERROR prints a FILE INFORMATION DISPLAY on the terminal, enabling you to determine the error number returned by FUPDATE, then aborts the program’s calling process.

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA(0:7):="DATAONE "
00004000 00005 1 ARRAY BUFFER(0:127)
00005000 00005 1 INTEGER DFILE1, LGTH, DUMMY, IN, LIST
00006000 00005 1
00007000 00005 1 INTRINSIC FOPEN, FREAD, FUPDATE, FLOCK, FUNLOCK, FCLOSE,
00008000 00005 1 PRINT 'FILE' INFO, QUIT, FWRITE, FREAD
00009000 00005 1
00010000 00005 1 PROCEDURE FILERERROR(FILENO, QUITNO)
00011000 00000 1 VALUE QUITNO
00012000 00000 1 INTEGER FILENO, QUITNO
00013000 00000 1 BEGIN
00014000 00000 2 PRINT 'FILE' INFO(FILENO)
00015000 00002 2 QUIT(QUITNO)
00016000 00004 2 END
00017000 00000 1
00018000 00000 1 <<END OF DECLARATIONS>>
00019000 00000 1
00020000 00000 1 DFILE1:=FOPEN(DATA1,%5,%345,128) <<OLD DISC FILE>>
00021000 00011 1 IF < THEN FILERERROR(DFILE1,1) <<CHECK FOR ERROR>>
00022000 00015 1
00023000 00015 1 IN:=FOPEN(,%244) <<$STDIN>>
00024000 00024 1 IF < THEN FILERERROR(IN,2) <<CHECK FOR ERROR>>
00025000 00030 1
00026000 00030 1 LIST:=FOPEN(,%614,%1) <<$STDLIST>>
00027000 00040 1 IF < THEN FILERERROR(LIST,3) <<CHECK FOR ERROR>>
00028000 00044 1
00029000 00044 1 UPDATE *LOOP:
00030000 00044 1 FLOCK(DFILE1,1) <<LOCK FILE/SUSPEND>>
00031000 00047 1 IF < THEN FILERERROR(DFILE1,4) <<CHECK FOR ERROR>>
00032000 00053 1
00033000 00053 1 LGTH:=FREAD(DFILE1,BUFFER,128) <<GET EMPLOYEE RECD>>
00034000 00061 1 IF < THEN FILERERROR(DFILE1,5) <<CHECK FOR ERROR>>
00035000 00065 1 IF > THEN GO END*OF*FILE <<CHECK FOR EOF>>
00036000 00070 1
00037000 00070 1 FWRITE(LIST,BUFFER,-20,%320) <<EMPLOYEE NAME>>
00038000 00075 1 IF <> THEN FILERERROR(LIST,6) <<CHECK FOR ERROR>>
00039000 00101 1
00040000 00101 1 DUMMY:=FREAD(IN,BUFFER(30),5) <<EMPLOYEE NUMBER>>
00041000 00110 1 IF < THEN FILERERROR(IN,7) <<CHECK FOR ERROR>>
00042000 00114 1 IF > THEN GO END*OF*FILE
00043000 00115 1
00044000 00115 1 FUPDATE(DFILE1,BUFFER,128) <<EMPLOYEE RECORD>>
00045000 00121 1 IF <> THEN FILERERROR(DFILE1,8) <<CHECK FOR ERROR>>
00046000 00125 1
00047000 00125 1 FUNLOCK(DFILE1) <<ALLOW OTHER ACCESS>>
00048000 00127 1 IF <> THEN FILERERROR(DFILE1,9) <<CHECK FOR ERROR>>
00049000 00133 1
00050000 00133 1 GO UPDATE *LOOP: <<CONTINUE UPDATE>>
00051000 00140 1
00052000 00140 1 END*OF*FILE:
00053000 00140 1 FUNLOCK(DFILE1) <<ALLOW OTHER ACCESS>>
00054000 00142 1 IF <> THEN FILERERROR(DFILE1,10) <<CHECK FOR ERROR>>
00055000 00146 1
00056000 00146 1 FCLOSE(DFILE1,0,0) <<DISP-NO CHANGE>>
00057000 00151 1 IF < THEN FILERERROR(DFILE1,11) <<CHECK FOR ERROR>>
00058000 00155 1 END.
PRIMARY DR STORAGE=%007: SECONDARY DB STORAGE=%00204
NO. ERRORS=0001 NO. WARNINGS=000
PROCESSOR TIME=0:00:03: ELAPSED TIME=0:00:17

```

Figure 3-18. FUPDATE Intrinsic Example

## USING THE IOWAIT INTRINSIC

Figure 3-19 shows a program that opens several terminals for input.

The statement

```
OUT:=FOPEN(OUTPUT,4,1,,DEV);
```

opens the line printer for output and the WHILE statement begins a loop to open the terminals.

In order to open a file with both the NOBUF and NO-WAIT *aoptions* specified, the program must be running in privileged mode, and this program is switched to privileged mode with the statement

```
GETPRIVMODE;
```

The statement

```
FILE:=FOPEN(TNAM,%405,%4404,36,DEV(3));
```

opens a terminal. The parameters specified are

*formal designator* DATAIN, which is contained in the byte array TNAM.

*foptions* %405, for which the bit pattern is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	1	Binary
							4			0				5		Octal

The above bit pattern specifies the following file options:

Domain: Old permanent file, system file domain. Bits (14:2) = 01.

ASCII/Binary: ASCII. Bit (13:1) = 1.

File Designator: Actual file designator = formal file designator. Bits (10:3) = 000.

Record Format: Fixed-length records. Bits (8:2) = 00.

Carriage Control: Carriage-control character expected. Bit (7:1) = 1.

*aoptions*

%4404, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	1	0	0	1	0	0	0	0	0	1	0	0	Binary
				4			4			0				4		Octal

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1   BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00004000 00005 1   BYTE ARRAY TNAM(0:6):="DATAIN ";
00005000 00005 1   BYTE ARRAY DEV(0:7):="LP TERM ";
00006000 00005 1   INTEGER OUT,FILE,LGTH,I:=-1,PROMPT:="? ",DONE:=0;
00007000 00005 1   EQUATE MAXTRM=3;
00008000 00005 1   ARRAY BUFR(0:36*MAXTRM);
00009000 00005 1   INTEGER ARRAY OPEN(0:MAXTRM);
00010000 00005 1   DEFINE CCL = IF < THEN QUIT#,
00011000 00005 1       CCG = IF > THEN QUIT#,
00012000 00005 1       CCNE= IF <> THEN QUIT#;
00013000 00005 1
00014000 00005 1   INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,GETPRIVMODE,GETUSERMODE,
00015000 00005 1       IOWAIT,QUIT;
00016000 00005 1
00017000 00005 1   <<END OF DECLARATIONS>>
00018000 00005 1
00019000 00005 1   OUT:=FOPEN(OUTPUT,4,1,,DEV); CCL(1);           <<LINEPRINTER OUTPUT>>
00020000 00015 1   WHILE (I:=I+1)<MAXTRM DO                       <<LOOP-SET UP TERMS>>
00021000 00023 1       BEGIN
00022000 00023 2           GETPRIVMODE; CCG(2);           <<FOR NOWAIT FOPEN>>
00023000 00027 2           FILE:=FOPEN(TNAM,%405,%4404,36,DEV(3)); <<DATA INPUT TERMINAL>>
00024000 00042 2           CCL(3);                       <<CHECK FOR ERROR>>
00025000 00045 2           GETUSERMODE; CCG(4);           <<FOR NOWAIT I/O>>
00026000 00051 2           OPEN(I):=FILE;                 <<SAVE FILE NUMBERS>>
00027000 00054 2           FWRITE(FILE,PROMPT,1,%320); CCNE(5); <<OUTPUT ? PROMPT>>
00028000 00064 2           IOWAIT(FILE); CCNE(6);         <<COMPLETE REQUEST>>
00029000 00075 2           FREAD(FILE,BUFR(I*36),-72); CCNE(7); <<INPUT DATA=NOWAIT>>
00030000 00111 2           END;
00031000 00116 1   WAIT:
00032000 00116 1       FILE:=IOWAIT(0,,LGTH); CCL(8);       <<WAIT FOR 1ST DONE>>
00033000 00130 1       IF > THEN                       <<EOF ON TERM READ>>
00034000 00131 1           BEGIN
00035000 00131 2               FCLOSE(FILE,0,0); CCL(9);   <<TERMINAL FILE>>
00036000 00137 2               IF(DONE:=DONE+1)>=MAXTRM THEN GO EXIT; <<ALL TERMS CLOSED?>>
00037000 00143 2           END
00038000 00143 1       ELSE
00039000 00145 1           BEGIN
00040000 00145 2               I:=-1;                       <<SET BUFFER INDEX>>
00041000 00147 2               DO I:=I+1                     <<INCR BUFFER INDEX>>
00042000 00147 2                   UNTIL OPEN(I)=FILE OR I=MAXTRM; <<SEARCH FOR FILE NO>>
00043000 00157 2                   IF I=MAXTRM THEN QUIT(10); <<FILE NOT FOUND>>
00044000 00164 2                   FWRITE(OUT,BUFR(I*36),-LGTH,0); <<COPY INPUT TO LP>>
00045000 00174 2                   CCNE(11);                 <<CHECK FOR ERROR>>
00046000 00177 2                   FWRITE(FILE,PROMPT,1,%320); CCNE(12); <<OUTPUT ? PROMPT>>
00047000 00207 2                   IOWAIT(FILE); CCNE(13);   <<COMPLETE REQUEST>>
00048000 00220 2                   FREAD(FILE,BUFR(I*36),-72); CCNE(14); <<INPUT DATA=NOWAIT>>
00049000 00234 2                   END;
00050000 00234 1                   GO TO WAIT;               <<CONTINUE>>
00051000 00235 1   EXIT:END,
PRIMARY DB STORAGE=%013; SECONDARY DB STORAGE=%00175
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:02; ELAPSED TIME=0:00:08

```

Figure 3-19. Using the IOWAIT Intrinsic

The above bit pattern specifies the following access options:

Access Type: Input/output. Bits (12:4) = 0100.

Multirecord: Non-multirecord. Bit (11:1) = 0.

Dynamic Locking: Disallowed. Bit (10:1) = 0.

Exclusive: Exclusive access. Default when I/O access is specified and bits (8:2) = 00.

Inhibit Buffering: Selected (NOBUF). Bit (7:1) = 1.

No-Wait I/O: Selected. Bit (4:1) = 1.

*reclsize*                    36 words.

*device*                    T (terminal), specified in element (3) of byte array DEV.

Once the file is opened, the program is switched back to the non-privileged mode with the statement

```
GETUSERMODE;
```

The first file number is saved in FILEBASE, a prompt is displayed on the terminal, and the IOWAIT intrinsic is called to wait until the request is completed. Input from the terminal is read and stored in BUFR at the location determined by the file number. (Input from the first terminal opened starts at BUFR location 0, the next terminal input starts at location 36, and so forth.)

The statements

```
FILE:=IOWAIT(0,,LGTH);
```

```
IF > THEN
```

wait for an end-of-file indication (the user enters an :EOF: command) from the first terminal on which the input is complete. If the end-of-file indication is received, this terminal is closed.

The input from the terminal is printed on the line printer and another prompt is displayed. Again, the IOWAIT intrinsic is called to wait until the request is completed. When DONE = MAXTRM (all terminals closed), control is passed to EXIT and the program terminates.

Note that the IODONTWAIT intrinsic (not shown in figure 3-19) behaves the same as IOWAIT with one exception: If IOWAIT is called and no I/O has completed, the calling process is suspended until some I/O completes; if IODONTWAIT is called and no I/O has completed, control is returned to the calling process. Thus, the program shown in figure 3-19 would not have suspended if the IODONTWAIT intrinsic had been called, and control would have returned to the program.

## WRITING AND READING USER FILE LABELS

MPE allows you to write and read user-defined labels on disc files with the FWRITELABEL and FREADLABEL intrinsics. Such labels are very useful; for example, labels can be used on files that are updated frequently in order to determine the time of the last update.

In figure 3-20 the FOPEN intrinsic call

```
DFILE2:=FOPEN(DATA2,%4,%4,128,,,1);
```

opens a *new* file and specifies 1 for the *userlabels* parameter (last parameter before parenthesis in this example), meaning that one 128-word record will be set aside for user labels. Any attempt to write a label beyond this 128-word limit will result in a CCG condition code and the intrinsic request will be denied. Note that any subsequent FWRITELABEL intrinsic calls will write over an existing label.

The statement

```
FWRITELABEL(DFILE2,LABL,9,0);
```

calls the intrinsic FWRITELABEL to write a user-supplied label. The parameters specified in the intrinsic call are

<i>filenum</i>	Supplied by DFILE2, which was assigned the file number when the FOPEN intrinsic opened the file.
<i>target</i>	The array LABL, containing the string “EMPLOYEE DATA FILE”, which will be written as the user file label.
<i>tcount</i>	9 words, specifying the length of the string to be transferred from the array LABL.
<i>labelid</i>	0, specifying the number of the label. (0 = first label, 1 = second label, etc.).

If the label is written successfully, a CCE condition code results. The statement

```
IF <> THEN FILEERROR(DFILE2,3);
```

checks for a “not equal” condition code and, if such is the case, calls the error-check procedure FILEERROR. The FILEERROR procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FWRITELABEL, then aborts the program’s process.

## READING A USER FILE LABEL

To read a user file label, you use the FREADLABEL intrinsic. Before reading occurs, MPE checks to ensure that you have *read-access* capability for the file on which the file label is to be read. The file therefore must be opened with one of the following access type *aoptions*:

- Read access only. Bits (12:4) = 0000.
- Input/output access. Bit (12:4) = 0100.
- Update access. Bits (12:4) = 0101.

In figure 3-21, the FOPEN intrinsic call

```
DFILE2:=FOPEN(DATA2,%6,%4,128);
```

contains the *aoptions* parameter %4, which specifies input/output access.



```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000 00005 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00005000 00005 1 ARRAY LABL(0:8):="EMPLOYEE DATA FILE";
00006000 00011 1 ARRAY BUFFER(0:127);
00007000 00011 1 INTEGER DFILE1,DFILE2,DUMMY;
00008000 00011 1 DOUBLE REC;
00009000 00011 1
00010000 00011 1 INTRINSIC FOPEN,FWRITELABEL,FGETINFO,FREAD,FWRITEDIR,FCLOSE,
00011000 00011 1 PRINT*FILE*INFO,QUIT;
00012000 00011 1
00013000 00011 1 PROCEDURE FILERERROR(FILENO,QUITNO);
00014000 00000 1 VALUE QUITNO;
00015000 00000 1 INTEGER FILENO,QUITNO;
00016000 00000 1 BEGIN
00017000 00000 2 PRINT*FILE*INFO(FILENO);
00018000 00002 2 QUIT(QUITNO);
00019000 00004 2 END;
00020000 00000 1
00021000 00000 1 <<END OF DECLARATIONS>>
00022000 00000 1
00023000 00000 1 DFILE1:=FOPEN(DATA1,%5,%100); <<OLD FILE-DATAONE>>
00024000 00010 1 IF < THEN FILERERROR(DFILE1,1); <<CHECK FOR ERROR>>
00025000 00014 1
00026000 00014 1 DFILE2:=FOPEN(DATA2,%4,%4,128,,1); <<NEW FILE-DATATWO>>
00027000 00027 1 IF < THEN FILERERROR(DFILE2,2); <<CHECK FOR ERROR>>
00028000 00033 1
00029000 00033 1 FWRITELABEL(DFILE2,LABL,9,0); <<FILE ID>>
00030000 00041 1 IF <> THEN FILERERROR(DFILE2,3); <<CHECK FOR ERROR>>
00031000 00045 1
00032000 00045 1 FGETINFO(DFILE1,,,,,,,,,REC); <<LOCATE EOF>>
00033000 00053 1 IF < THEN FILERERROR(DFILE1,4); <<CHECK FOR ERROR>>
00034000 00057 1
00035000 00057 1 INVERT*LOOP;
00036000 00057 1 DUMMY:=FREAD(DFILE1,BUFFER,128); <<OLD FILE RECORD>>
00037000 00065 1 IF < THEN FILERERROR(DFILE1,5); <<CHECK FOR ERROR>>
00038000 00071 1 IF > THEN GO END*OF*FILE; <<CHECK FOR EOF>>
00039000 00072 1
00040000 00072 1 REC:=REC-1; <<LAST REDC NO>>
00041000 00076 1 FWRITEDIR(DFILE2,BUFFER,128,REC); <<INVERT REC ORDER>>
00042000 00103 1 IF <> THEN FILERERROR(DFILE2,6); <<CHECK FOR ERROR>>
00043000 00107 1
00044000 00107 1 GO INVERT*LOOP; <<CONTINUE OPERATION>>
00045000 00116 1
00046000 00116 1 END*OF*FILE;
00047000 00116 1 FCLOSE(DFILE2,2,0); <<SAVE NEW AS TEMP>>
00048000 00122 1 IF < THEN FILERERROR(DFILE2,7); <<CHECK FOR ERROR>>
00049000 00126 1
00050000 00126 1 FCLOSE(DFILE1,4,0); <<DELETE OLD FILE>>
00051000 00132 1 IF < THEN FILERERROR(DFILE1,8); <<CHECK FOR ERROR>>
00052000 00136 1 END.
PRIMARY DB STORAGE=%011; SECONDARY DB STORAGE=%0221
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:04; ELAPSED TIME=0:00:59

```

Figure 3-20. FWRITELABEL Intrinsic Example



```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00004000 00005 1 BYTE ARRAY LISTFILE(0:8):="LISTFILE ";
00005000 00006 1 BYTE ARRAY ALTNAME(0:7):="ALTDATA ";
00006000 00005 1 ARRAY BUFFER(0:127);
00007000 00005 1 ARRAY MESSAGE(0:18):="DUPLICATE FILE NAME - FIX DURING BREAK";
00008000 00023 1 INTEGER DFILE2,LIST,ERROR;
00009000 00023 1 DOUBLE REC:=0D;
00010000 00023 1
00011000 00023 1 INTRINSIC FOPEN,FREADLABEL,FREADDIR,FWRITE,FCLOSE,FRENAME,
00012000 00023 1 FREADSEEK,CAUSEBREAK,FCHECK,PRINT'FILE'INFO,QUIT;
00013000 00023 1
00014000 00023 1 PROCEDURE FILERROR(FILENO,QUITNO);
00015000 00000 1 VALUE QUITNO;
00016000 00000 1 INTEGER FILENO,QUITNO;
00017000 00000 1 BEGIN
00018000 00000 2 PRINT'FILE'INFO(QUITNO);
00019000 00002 2 QUIT(QUITNO);
00020000 00004 2 END;
00021000 00000 1
00022000 00000 1 <<END OF DECLARATIONS>>
00023000 00000 1
00024000 00000 1 DFILE2:=FOPEN(DATA2,%6,%4,128); <<OLD TEMP FILE>>
00025000 00011 1 IF < THEN FILERROR(DFILE2,1); <<CHECK FOR ERROR>>
00026000 00015 1
00027000 00015 1 LIST:=FOPEN(LISTFILE,%14,%1); <<$STDLIST>>
00028000 00025 1 IF < THEN FILERROR(LIST,2); <<CHECK FOR ERROR>>
00029000 00031 1
00030000 00031 1 FREADLABEL(DFILE2,BUFFER,128,0); <<FILE ID>>
00031000 00037 1 IF <> THEN FILERROR(DFILE2,3); <<CHECK FOR ERROR>>
00032000 00043 1 FWRITE(LIST,BUFFER,9,0); <<DISPLAY ID>>
00033000 00050 1 IF <> THEN FILERROR(LIST,4); <<CHECK FOR ERROR>>
00034000 00054 1
00035000 00054 1 LIST*LOOP:
00036000 00054 1 FREADDIR(DFILE2,BUFFER,128,REC); <<EVERY OTHER RECD>>
00037000 00061 1 IF < THEN FILERROR(DFILE2,5); <<CHECK FOR ERROR>>
00038000 00065 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00039000 00066 1
00040000 00066 1 REC:=REC+2D; <<EVERY OTHER RECD>>
00041000 00072 1 FREADSEEK(DFILE2,REC); <<FILL SYSTEM BUFFER>>
00042000 00075 1 IF < THEN FILERROR(DFILE2,6); <<CHECK FOR ERROR>>
00043000 00101 1
00044000 00101 1 FWRITE(LIST,BUFFER,35,0); <<ALTERNATE RECORDS>>
00045000 00106 1 IF <> THEN FILERROR(LIST,7); <<CHECK FOR ERROR>>
00046000 00112 1
00047000 00112 1 GO LIST*LOOP; <<CONTINUE LISTING>>
00048000 00117 1
00049000 00117 1 END'OF'FILE;
00050000 00117 1 FCLOSE(DFILE2,1,0); <<MAKE PERMANENT>>
00051000 00123 1 IF = THEN GO DONE; <<LISTING DONE>>
00052000 00124 1 FCHECK(DFILE2,ERROR); <<FCLOSE ERROR>>
00053000 00131 1 IF ERROR=100 THEN <<DUPLICATE FILE NAME>>
00054000 00134 1 BEGIN
00055000 00134 2 FRENAME(DFILE2,ALTNAME); <<CHANGE FILE NAME>>
00056000 00137 2 CLOSE;
00057000 00137 2 FCLOSE(DFILE2,1,0); <<TRY AGAIN>>
00058000 00143 2 IF = THEN GO DONE; <<GOOD FCLOSE>>
00059000 00144 2 PRINT'FILE'INFO(DFILE2); <<PRINT ERROR>>
00060000 00146 2 FWRITE(LIST,MESSAGE,19,0); <<SEEK HELP>>
00061000 00153 2 CAUSEBREAK; <<SESSION BREAK>>
00062000 00154 2 GO CLOSE; <<LOOP BACK>>
00063000 00155 2 END;
00064000 00155 1 DONE;END.
PRIMARY DB STORAGE=%012; SECONDARY DB STORAGE=%00240
NO. FRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:04; ELAPSED TIME=0:00:58

```

Figure 3-21. FREADLABEL Intrinsic Example

The statement

```
FREADLABEL(DFILE2,BUFFER,128,0);
```

reads a user file label from the file specified by DFILE2. The parameters specified in the intrinsic call are

<i>filenum</i>	Supplied by DFILE2, which was assigned the file number when the FOPEN intrinsic opened the file.
<i>target</i>	BUFFER, the array in the stack to which the file label is transferred.
<i>tcount</i>	128, specifying the maximum number of words to be transferred.
<i>labelid</i>	0, specifying the number of the label to be read.

If the label is read, a CCE condition code results. The statement

```
IF <> THEN FILERROR(DFILE2,3);
```

checks for a “not equal” condition code and, if such is the case, calls the error-check procedure FILERROR. The FILERROR procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FREADLABEL, then aborts the program’s process.

## OBTAINING FILE ACCESS INFORMATION

You can request access and status information about an open file with the FGETINFO intrinsic.

The program shown in figure 3-22 uses the FGETINFO intrinsic call to locate the end-of-file, as follows:

```
FGETINFO(DFILE1,,,,,,,,,REC);
```

All parameters of the FGETINFO intrinsic except *filenum*, which supplies the file number of the file for which information is to be obtained, are optional. Note that all parameters in the above intrinsic call except *filenum* and *eof* are omitted. The commas between DFILE1 and REC signify to MPE that the parameters are omitted. Omissions from the end of the parameter list need not be signified to MPE by commas; instead, the parenthesis after REC signifies that this is the last parameter in the intrinsic call.

A double integer value equal to the number of logical records currently in the file is returned to REC.

If the FGETINFO request is not granted, a CCL condition code is returned. The statement

```
IF < THEN FILERROR(DFILE1,4);
```

checks for a condition code of CCL and, if such is the case, calls the error-check procedure FILERROR. This procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FGETINFO, then aborts the program’s process.

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000 00005 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00005000 00005 1 ARRAY LABL(0:8):="EMPLOYEE DATA FILE";
00006000 00011 1 ARRAY BUFFER(0:127);
00007000 00011 1 INTEGER DFILE1,DFILE2,DUMMY;
00008000 00011 1 DOUBLE REC;
00009000 00011 1
00010000 00011 1 INTRINSIC FOPEN,FWRITELABEL,FGETINFO,FREAD,FWRITEDIR,FCLOSE,
00011000 00011 1 PRINT'FILE'INFO,QUIT;
00012000 00011 1
00013000 00011 1 PROCEDURE FILERERROR(FILENO,QUITNO);
00014000 00000 1 VALUE QUITNO;
00015000 00000 1 INTEGER FILENO,QUITNO;
00016000 00000 1 BEGIN
00017000 00000 2 PRINT'FILE'INFO(FILENO);
00018000 00002 2 QUIT(QUITNO);
00019000 00004 2 END;
00020000 00000 1
00021000 00000 1 <<END OF DECLARATIONS>>
00022000 00000 1
00023000 00000 1 DFILE1:=FOPEN(DATA1,%5,%100); <<OLD FILE-DATAONE>>
00024000 00010 1 IF < THEN FILERERROR(DFILE1,1); <<CHECK FOR ERROR>>
00025000 00014 1
00026000 00014 1 DFILE2:=FOPEN(DATA2,%4,%4,128,,1); <<NEW FILE-DATATWO>>
00027000 00027 1 IF < THEN FILERERROR(DFILE2,2); <<CHECK FOR ERROR>>
00028000 00033 1
00029000 00033 1 FWRITELABEL(DFILE2,LABL,9,0); <<FILE ID>>
00030000 00041 1 IF <> THEN FILERERROR(DFILE2,3); <<CHECK FOR ERROR>>
00031000 00045 1
00032000 00045 1 FGETINFO(DFILE1,,,,,,,,,REC); <<LOCATE EOF>>
00033000 00053 1 IF < THEN FILERERROR(DFILE1,4); <<CHECK FOR ERROR>>
00034000 00057 1
00035000 00057 1 INVERT'LOOP;
00036000 00057 1 DUMMY:=FREAD(DFILE1,BUFFER,128); <<OLD FILE RECORD>>
00037000 00065 1 IF < THEN FILERERROR(DFILE1,5); <<CHECK FOR ERROR>>
00038000 00071 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00039000 00072 1
00040000 00072 1 REC:=REC-1; <<LAST REDC NO>>
00041000 00076 1 FWRITEDIR(DFILE2,BUFFER,128,REC); <<INVERT REC ORDER>>
00042000 0103 1 IF <> THEN FILERERROR(DFILE2,6); <<CHECK FOR ERROR>>
00043000 0107 1
00044000 0107 1 GO INVERT'LOOP; <<CONTINUE OPERATION>>
00045000 0116 1
00046000 0116 1 END'OF'FILE;
00047000 0116 1 FCLOSE(DFILE2,2,0); <<SAVE NEW AS TEMP>>
00048000 0122 1 IF < THEN FILERERROR(DFILE2,7); <<CHECK FOR ERROR>>
00049000 0126 1
00050000 0126 1 FCLOSE(DFILE1,4,0); <<DELETE OLD FILE>>
00051000 0132 1 IF < THEN FILERERROR(DFILE1,8); <<CHECK FOR ERROR>>
00052000 0136 1 END.
PRIMARY DB STORAGE=%011; SECONDARY DB STORAGE=%0221
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:04; ELAPSED TIME=0:00:59

```

Figure 3-22. FGETINFO Intrinsic Example

## OBTAINING FILE-ERROR INFORMATION

When a file system intrinsic returns a condition code indicating that an error occurred, you can request more details about the error in order to correct it. The FCHECK intrinsic call is used for this purpose.

In figure 3-23, the FCHECK intrinsic is used to determine the error number if a condition code error is returned when the FCLOSE intrinsic is executed. The statement

```
FCHECK(DFILE2,ERROR);
```

specifies that error information is to be returned for the file number designated by DFILE2. The error number is returned to ERROR as a 16-bit code. The statement

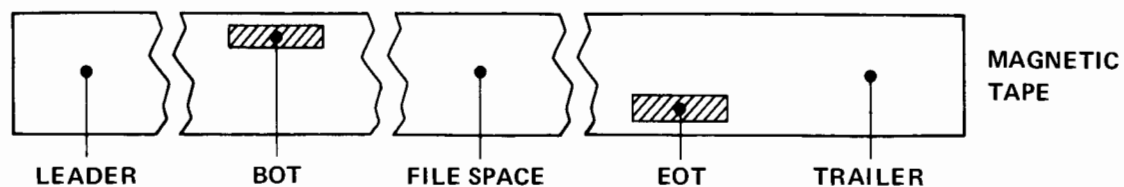
```
IF ERROR=100 THEN
```

checks the error number returned and, if ERROR = 100, executes a file rename procedure.

## MAGNETIC TAPE CONSIDERATIONS

Every standard reel of magnetic tape designed for digital computer use has two reflective markers located on the back side of the tape (opposite the recording surface). One of these marks is located behind the tape leader at the beginning of tape (BOT) position, and the other is located in front of the tape trailer at the end of tape (EOT) position.

These markers are sensed by the tape drive itself and their position on the tape (left or right side) determines whether they indicate the start or end of tape positions. (See below.)



As far as the magnetic tape hardware and software are concerned, the BOT marker is much more significant than the EOT marker because BOT signals the start of recorded information, but EOT simply indicates that the remaining tape supply is running low and the program writing the tape should bring the operation to an orderly conclusion. The difference in treatment of these two physical tape markers is reflected by the file system intrinsics when the file being read, written, or controlled is a magnetic tape device file. The following paragraphs discuss the characteristics of each appropriate intrinsic.

### FWRITE

When a user program attempts to write over or beyond the physical EOT tape marker, the FWRITE intrinsic returns an error condition code (CCL). The actual data has been written to the tape, and a call to FCHECK reveals a file error indicating END OF TAPE. All writes to the tape after the EOT tape marker has been crossed transfer the data successfully but return a CCL condition code until the tape crosses the EOT marker again in the reverse direction (rewind or backspace).

```

00001000 00000 0 $CONTROL USLIMIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA2(0:7):="DATATWO ";
00004000 00005 1 BYTE ARRAY LISTFILE(0:8):="LISTFILE ";
00005000 00006 1 BYTE ARRAY ALTNAME(0:7):="ALTDATA ";
00006000 00005 1 ARRAY BUFFER(0:127);
00007000 00005 1 ARRAY MESSAGE(0:18):="DUPLICATE FILE NAME - FIX DURING BREAK";
00008000 00023 1 INTEGER DFILE2,LIST,ERROR;
00009000 00023 1 DOUBLE REC:=0D;
00010000 00023 1
00011000 00023 1 INTRINSIC FOPEN,FREADLABEL,FREADDIR,FWRITE,FCLOSE,FRENAME,
00012000 00023 1 FREADSEEK,CAUSEBREAK,FCHECK,PRINT'FILE'INFO,QUIT;
00013000 00023 1
00014000 00023 1 PROCEDURE FILERROR(FILENO,QUITNO);
00015000 00000 1 VALUE QUITNO;
00016000 00000 1 INTEGER FILENO,QUITNO;
00017000 00000 1 BEGIN
00018000 00000 2 PRINT'FILE'INFO(QUITNO);
00019000 00002 2 QUIT(QUITNO);
00020000 00004 2 END;
00021000 00000 1
00022000 00000 1 <<END OF DECLARATIONS>>
00023000 00000 1
00024000 00000 1 DFILE2:=FOPEN(DATA2,%6,%4,128); <<OLD TEMP FILE>>
00025000 00011 1 IF < THEN FILERROR(DFILE2,1); <<CHECK FOR ERROR>>
00026000 00015 1
00027000 00015 1 LIST:=FOPEN(LISTFILE,%14,%1); <<$STDLIST>>
00028000 00025 1 IF < THEN FILERROR(LIST,2); <<CHECK FOR ERROR>>
00029000 00031 1
00030000 00031 1 FREADLABEL(DFILE2,BUFFER,128,0); <<FILE ID>>
00031000 00037 1 IF <> THEN FILERROR(DFILE2,3); <<CHECK FOR ERROR>>
00032000 00043 1 FWRITE(LIST,BUFFER,9,0); <<DISPLAY ID>>
00033000 00050 1 IF <> THEN FILERROR(LIST,4); <<CHECK FOR ERROR>>
00034000 00054 1
00035000 00054 1 LIST'LOOP:
00036000 00054 1 FREADDIR(DFILE2,BUFFER,128,REC); <<EVERY OTHER RECD>>
00037000 00061 1 IF < THEN FILERROR(DFILE2,5); <<CHECK FOR ERROR>>
00038000 00065 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00039000 00066 1
00040000 00066 1 REC:=REC+2D; <<EVERY OTHER RECD>>
00041000 00072 1 FREADSEEK(DFILE2,REC); <<FILL SYSTEM BUFFER>>
00042000 00075 1 IF < THEN FILERROR(DFILE2,6); <<CHECK FOR ERROR>>
00043000 00101 1
00044000 00101 1 FWRITE(LIST,BUFFER,35,0); <<ALTERNATE RECORDS>>
00045000 00106 1 IF <> THEN FILERROR(LIST,7); <<CHECK FOR ERROR>>
00046000 00112 1
00047000 00112 1 GO LIST'LOOP; <<CONTINUE LISTING>>
00048000 00117 1
00049000 00117 1 END'OF'FILE:
00050000 00117 1 FCLOSE(DFILE2,1,0); <<MAKE PERMANENT>>
00051000 00123 1 IF = THEN GO DONE; <<LISTING DONE>>
00052000 00124 1 FCHECK(DFILE2,ERROR); <<FCLOSE ERROR>>
00053000 00131 1 IF ERROR=100 THEN <<DUPLICATE FILE NAME>>
00054000 00134 1 BEGIN
00055000 00134 2 FRENAME(DFILE2,ALTNAME); <<CHANGE FILE NAME>>
00056000 00137 2 CLOSE:
00057000 00137 2 FCLOSE(DFILE2,1,0); <<TRY AGAIN>>
00058000 00143 2 IF = THEN GO DONE; <<GOOD FCLOSE>>
00059000 00144 2 PRINT'FILE'INFO(DFILE2); <<PRINT ERROR>>
00060000 00146 2 FWRITE(LIST,MESSAGE,19,0); <<SEEK HELP>>
00061000 00153 2 CAUSEBREAK; <<SESSION BREAK>>
00062000 00154 2 GO CLOSE; <<LOOP BACK>>
00063000 00155 2 END;
00064000 00155 1 DONE;END,
PRIMARY DB STORAGE=%012; SECONDARY DB STORAGE=%00240
NO. FRRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:04; ELAPSED TIME=0:00:58

```

Figure 3-23. FCHECK Intrinsic Example

## **FREAD**

A user program can read data written over an EOT marker and beyond the marker into the tape trailer. The intrinsic returns no error condition code (CCL or CCG) and does not initiate a file system error code when the EOT marker is encountered.

## **FSPACE**

A user program can space records over or beyond the EOT marker without receiving an error condition code (CCL or CCG) or a file system error. The intrinsic does, however, return a CCG condition code when a logical file mark is encountered. If the user program attempts to backspace records over the BOT marker, the intrinsic returns a CCG condition code and remains positioned on the BOT marker.

## **FCONTROL (WRITE EOF)**

If a user program writes a logical end of file (EOF) mark on a magnetic tape over the reflective EOT marker, or in the tape trailer after the marker, the FCONTROL intrinsic returns an error condition code (CCL) and sets a file system error to indicate END OF TAPE. The file mark is actually written to the tape.

## **FCONTROL (FORWARD SPACE TO FILE MARK)**

A user program which spaces forward to logical tape file marks (EOFs) with the FCONTROL intrinsic cannot detect passing the physical EOT marker. No special condition code is returned.

## **FCONTROL (BACKWARD SPACE TO FILE MARK)**

The EOT reflective marker is not detected by FCONTROL during backspace file (EOF) operations. If the intrinsic discovers a BOT marker before it finds a logical EOF, it returns a condition code of CCE and treats the BOT as if it were a logical EOF. Subsequent backspace file operations requested when the file is at BOT are treated as errors and return a CCL condition code and set a file system error to indicate INVALID OPERATION.

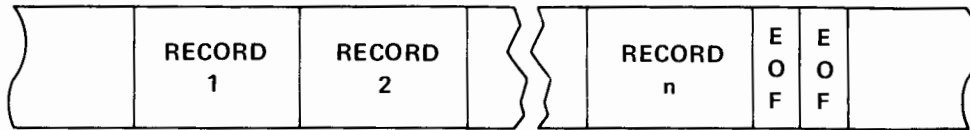
In summary, only those intrinsics which cause the magnetic tape to write information are capable of sensing the physical EOT marker. If a program designed to read a magnetic tape needed to detect the EOT marker, it could be done by using the FCONTROL intrinsic to read the physical status of the tape drive itself. When the drive passes the EOT marker and is moving in the forward direction, tape status bit 5 (%2000) is set and remains on until the drive detects the EOT marker during a rewind or backspace operation. Under normal circumstances, however, it is not necessary to check for EOT during read operations. The responsibility for detecting end of tape and concluding tape operations in an orderly manner belongs to the program which originally created (wrote) the tape.

A program which needed to create a multi-volume (multiple reel) tape file would normally write tape records until the status returned from FWRITE indicated an EOT condition. Writing could be continued in a limited manner to reach a logical point at which to break the file. Then several file marks and a trailing tape label would typically be added, the tape rewound, another reel mounted, and the data transfer continued. The program designed to read such a multi-volume file must expect to find and check for the EOF and label sequence written by the tape's creator. Since the logical end of the tape may be somewhat past the physical EOT marker, the format and conventions used to create the tape are of more importance than determining the location of the EOT.

## END-OF-FILE MARKS ON MAGNETIC TAPE

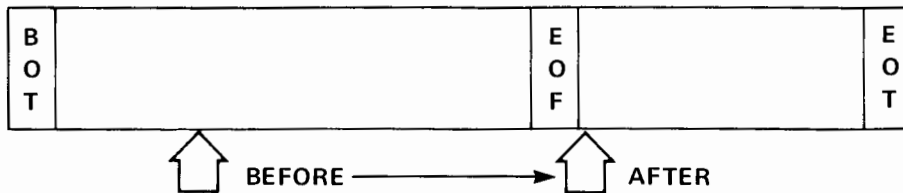
An FWRITE to magnetic tape, followed by any intrinsic call which reverses tape motion (for example, backspace a record, backspace a file, or rewind) causes the file system to write an EOF mark before initiating the reverse motion.

For example, if a user program has just written several data records to magnetic tape, writes a file mark, rewinds the tape, and closes the file, the tape file will be terminated by two file marks (EOF). The first of these was requested by the user by calling FCONTROL to write an EOF, and the second was provided by the system because the direction of tape motion had been reversed after a write (rewind). See below.

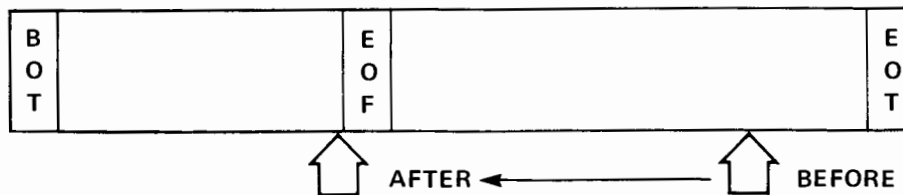


## SPACING FILE MARKS

When you space forward to a tape mark (EOF), the tape recording heads have just read the EOF and are positioned beyond it, as follows:



When you space backward to a tape mark (EOF), the mark is recognized as the tape travels in the reverse direction. The tape heads then are left positioned just in front of the EOF that was read, as follows:



### NOTE

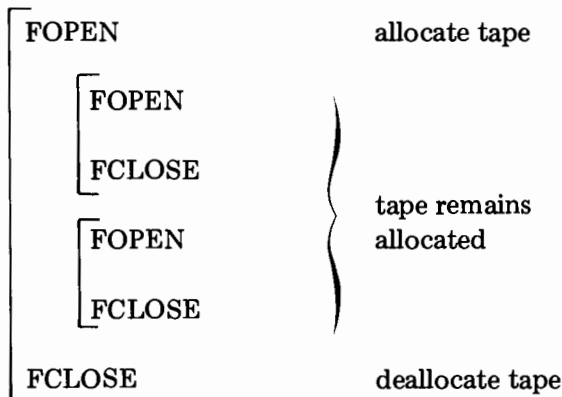
BOT (beginning of tape) and EOT (end of tape) correspond to the reflective markers on the reel of magnetic tape.

When FREAD has found a logical file mark and returned a condition code of CCG, the EOF mark has been read and the tape heads are positioned immediately following the mark (similar to space forward to tape mark above).

## USING THE FCLOSE INTRINSIC WITH MAGNETIC TAPE

The operation of the FCLOSE intrinsic as used with magnetic tape is outlined in the flowchart of figure 3-24.

Note that a tape closed with the temporary no-rewind disposition will be rewound and unloaded if certain additional conditions are not met. It is possible for a single process to FOPEN a magnetic tape device using a device class and later FOPEN the same device again using its logical device number. This may be done in such a manner that both magnetic tape files are open concurrently. The second FOPEN does not require any operator intervention (for example, for device allocation). When FOPEN/FCLOSE calls are arranged in a nested fashion, tape files may be closed without deallocating the physical device, as follows:



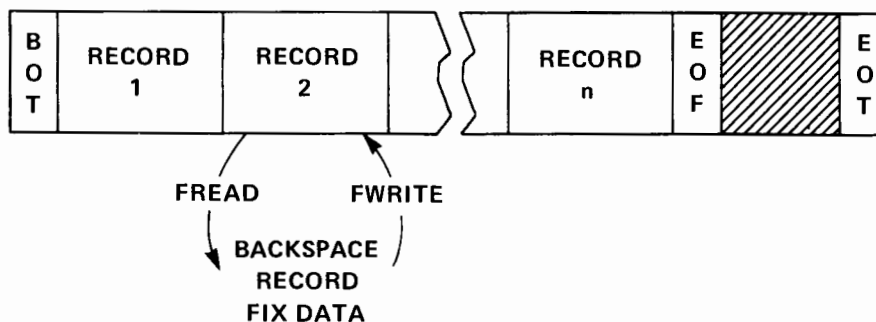
Such nesting of FOPEN/FCLOSE pairs is required to keep an FCLOSED tape from rewinding. A tape closed with the temporary, no-rewind disposition will be rewound and unloaded unless the process closing it has another file currently open on the device.

Note also that when a temporary no-wind tape is deallocated, the file system has not placed an end-of-file mark at the end of the data file.

## UPDATING MAGNETIC TAPE FILES

As a physical data storage device, magnetic tape is not designed to enable the replacement of a single record in an existing file. An attempt to perform this type of operation will cause problems in maintaining the integrity of records on the tape. Magnetic tape files, therefore, should not be maintained (updated) on an individual record basis but should be updated during copy operations from one file to another.

As an example of the type of problems that can occur, consider the results of attempting to read a tape record, modify its data, backspace the tape, and overwrite the original record, as follows:





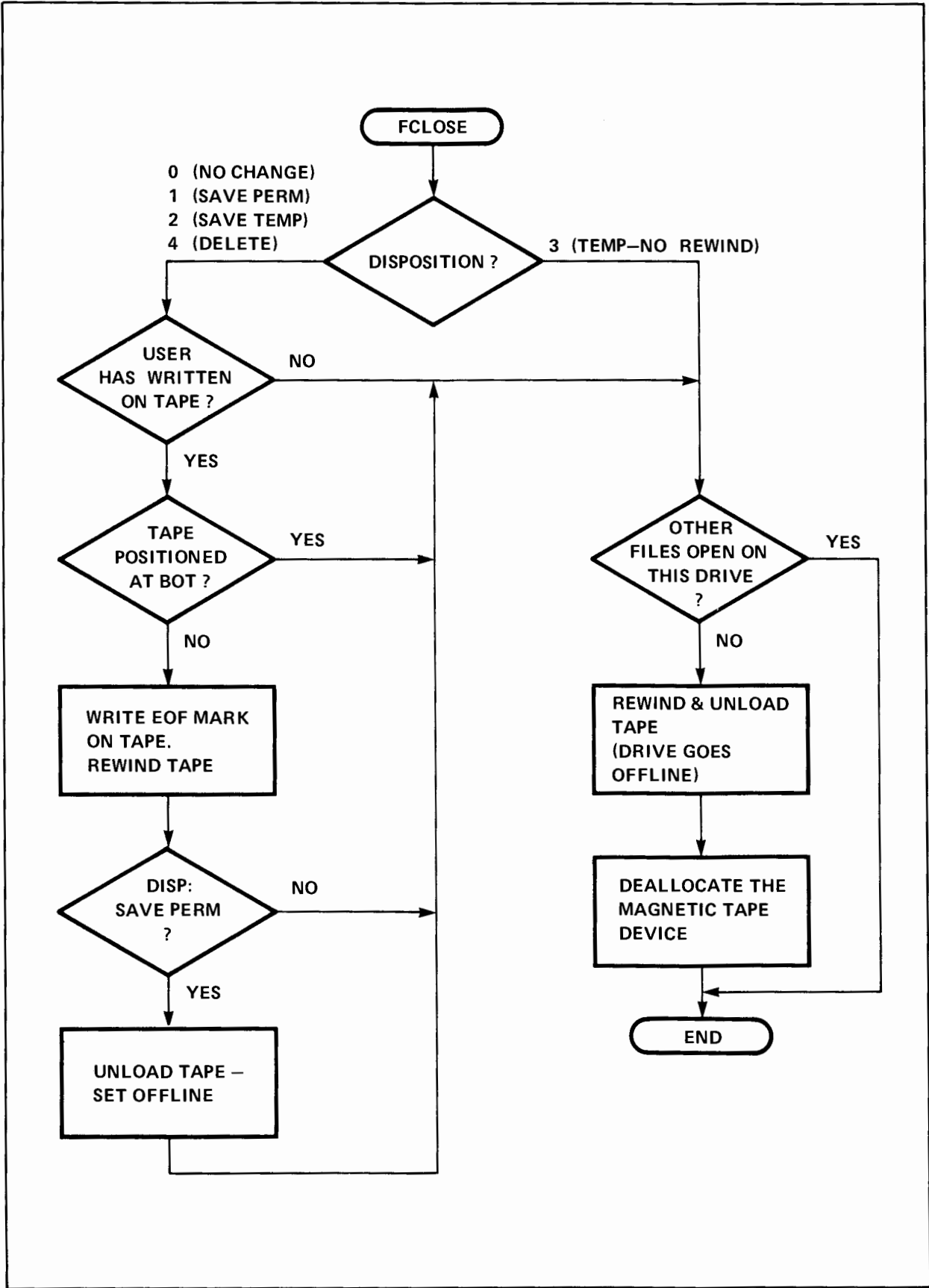
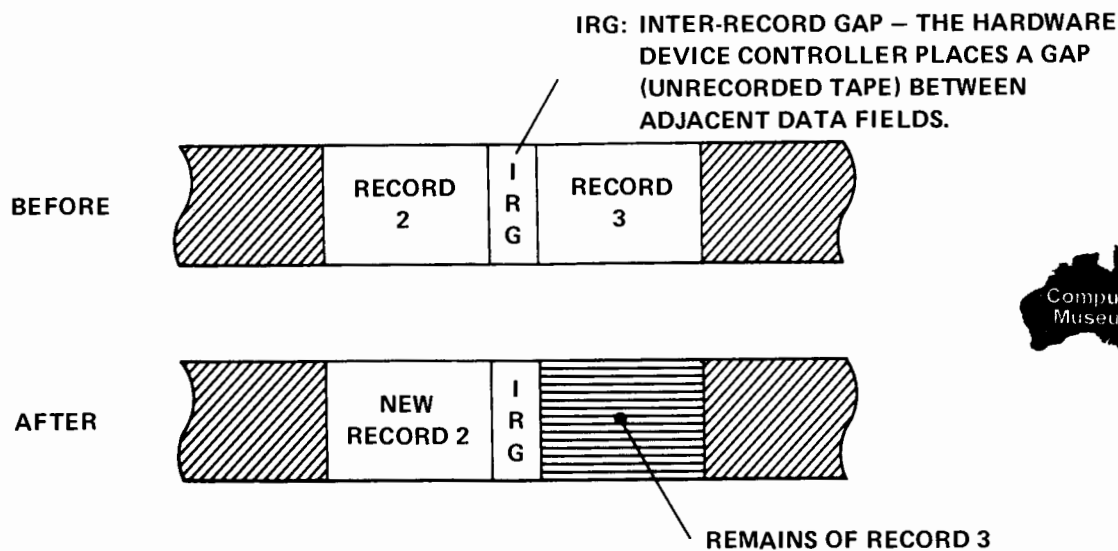
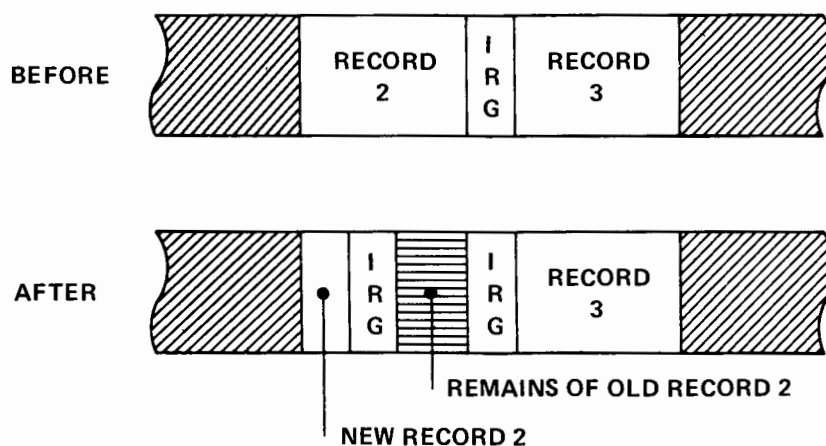


Figure 3-24. Using the FCLOSE Intrinsic with Magnetic Tape Files

If the replacement differed at all in size from the original record, the result would not simply be an update of the record. A replacement record of greater length than the original record would overwrite (destroy) a portion of the next record on the tape, as shown below.



On the other hand, if the length of the replacement record is less than that of the original record, a portion of the original record will still remain on the tape as shown below.



In either of the two cases shown, the partial records remaining would cause magnetic tape read errors and would create problems in subsequent processing of the tape file.

Even with replacement records of the same size as the original records, errors can still result. Mechanical and timing variations from one magnetic tape drive to another can create substantial differences in the actual length of tape records containing the same amount of data. Magnetic tape standards, for example, permit the inter-record gap (IRG) to vary in length from 0.5 to 0.7 inches. Similar variations may occur to a lesser extent in the spacing of the actual data bytes recorded. In short, the variation of a number of hardware factors which are beyond the user's control can affect

the physical length of the tape records written. For this reason, always update tape files during copy operations from one tape to another.

## READING AND WRITING A MAGNETIC TAPE FILE

Figure 3-25 contains a program that copies a magnetic tape file into another file on the same reel of tape.

The FOPEN intrinsic call

```
MT:=FOPEN(NAME,%201,%4,66,CLASS);
```

opens the magnetic tape file. The parameters specified are

*formal designator*                   MAGTAPE, which is contained in the byte array NAME

*foptions*                            %201, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	Binary
								2			0				1	Octal

The above bit pattern specifies the following file options:

Domain: Old permanent file. Bits (14:2) = 01.

ASCII/Binary: Binary. Bit (13:1) = 0.

Default Designator: Same as formal file designator. Bits (10:3) = 000.

Record Format: Undefined length. Bits (8:2) = 10.

*aoptions*                            %4, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	Binary
															4	Octal

The above bit pattern specifies the following access options:

Access Type: Input/output access. Bits (12:4) = 0100.

*reclsize*                            66 words.

*device*                              TAPE, contained in the byte array CLASS.

All other parameters are omitted from the FOPEN intrinsic call.

```

00001000 00000 0  %CONTROL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  INTEGER MT,RECD'POSITION:=0,LGTH;
00004000 00000 1  BYTE ARRAY NAME(0:7):="MAGTAPE ";
00005000 00005 1  BYTE ARRAY CLASS(0:4):="TAPE ";
00006000 00004 1  ARRAY BUFFER(0:65);
00007000 00004 1  LOGICAL DUMMY;
00008000 00004 1
00009000 00004 1  INTRINSIC FOPEN,FREAD,FCONTROL,FSPACE,FWRITE,FCLOSE,
00010000 00004 1  PRINT'FILE'INFO,QUIT;
00011000 00004 1
00012000 00004 1  PROCEDURE FILERROR(FILENO,QUITNO);
00013000 00000 1  VALUE FILENO,QUITNO;
00014000 00000 1  INTEGER FILENO,QUITNO;
00015000 00000 1  BEGIN
00016000 00000 2  PRINT'FILE'INFO(FILENO);
00017000 00002 2  QUIT(QUITNO);
00018000 00004 2  END;
00019000 00000 1
00020000 00000 1  <<END OF DECLARATIONS>>
00021000 00000 1
00022000 00000 1  MT:=FOPEN(NAME,%201,%4,66,CLASS); <<MAG TAPE>>
00023000 00012 1  IF < THEN FILERROR(MT,1); <<CHECK FOR ERROR>>
00024000 00016 1
00025000 00016 1  COPY*LOOP;
00026000 00016 1  LGTH:=FREAD(MT,BUFFER,66); <<TAPE FILE 1>>
00027000 00024 1  IF < THEN FILERROR(MT,2); <<CHECK FOR ERROR>>
00028000 00030 1  IF > THEN GO DONE; <<CHECK FOR EOF>>
00028100 00033 1
00029000 00033 1  FCONTROL (MT,7,DUMMY); <<GO TO END FILE 1>>
00030000 00037 1  IF < THEN FILERROR(MT,3); <<CHECK FOR ERROR>>
00031000 00043 1  FSPACE(MT,RECD'POSITION); <<NEXT FILE 2 RECD>>
00032000 00046 1  IF <> THEN FILERROR(MT,4); <<CHECK FOR ERROR>>
00032100 00052 1
00033000 00052 1  FWRITE (MT,BUFFER,LGTH,0); <<TAPE FILE 2>>
00034000 00057 1  IF <> THEN FILERROR(MT,5); <<CHECK FOR ERROR>>
00035000 00063 1
00036000 00063 1  FCONTROL (MT,8,DUMMY); <<BACK TO END FILE 1>>
00037000 00067 1  IF < THEN FILERROR(MT,6); <<CHECK FOR ERROR>>
00038000 00073 1  FCONTROL (MT,8,DUMMY); <<BACK TO START FILE 1>>
00039000 00077 1  IF < THEN FILERROR(MT,7); <<CHECK FOR ERROR>>
00040000 00103 1
00041000 00103 1  RECD'POSITION:=RECD'POSITION+1; <<INCR RECORD CNTR>>
00042000 00104 1
00043000 00104 1  FSPACE (MT,RECD'POSITION); <<NEXT FILE 1 RECD>>
00044000 00107 1  IF <> THEN FILERROR(MT,8); <<CHECK FOR ERROR>>
00045000 00113 1  GO COPY*LOOP; <<CONTINUE COPYING>>
00047000 00115 1
00048000 00115 1  DONE;
00049000 00115 1  FCONTROL (MT,5,DUMMY); <<REWIND TAPE>>
00050000 00121 1  IF < THEN FILERROR(MT,9); <<CHECK FOR ERROR>>
00051000 00125 1
00052000 00125 1  FCLOSE (MT,0,0); <<MAG TAPE>>
00053000 00130 1  IF < THEN FILERROR(MT,10); <<CHECK FOR ERROR>>
00054000 00134 1
00055000 00134 1  END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00111

```

Figure 3-25. Magnetic Tape Example

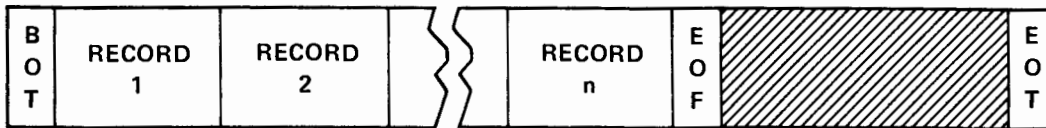
Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable MT.

The statement

```
IF < THEN FILEERROR(MT,1);
```

checks the condition code and, if it is CCL, calls the error-check procedure FILEERROR. The FILEERROR procedure prints a FILE INFORMATION DISPLAY on the standard list device, enabling you to determine the error number returned by FOPEN, then aborts the program's process.

The tape format before the copy operation is started is as follows:



The statement

```
LGTH:=FREAD(MT,BUFFER,66);
```

reads a record from the file designated by MT and transfers this record to BUFFER. The statement reads up to 66 words from the record, then returns a positive value to LGTH indicating the actual length of the information transferred.

The statement

```
FCONTROL(MT,7,DUMMY);
```

spaces forward to the EOF tape mark (the end of the file). As you recall from the paragraph "SPACING FILE MARKS", the recording head actually is positioned slightly beyond the EOF file mark. Now the statement

```
FSPACE(MT,RECD'POSITION);
```

spaces the tape to the point where the first record (RECD'POSITION = 0, see statement number 3 in the program) of the second file is to begin. The statement

```
FWRITE(MT,BUFFER,LGTH,0);
```

writes the record contained in the array BUFFER into this record.

The statement

```
FCONTROL(MT,8,DUMMY);
```

spaces back to the end of file 1 (the EOF mark) and the statement

```
FCONTROL(MT,8,DUMMY);
```

then spaces back to the next tape mark (the start of file 1).

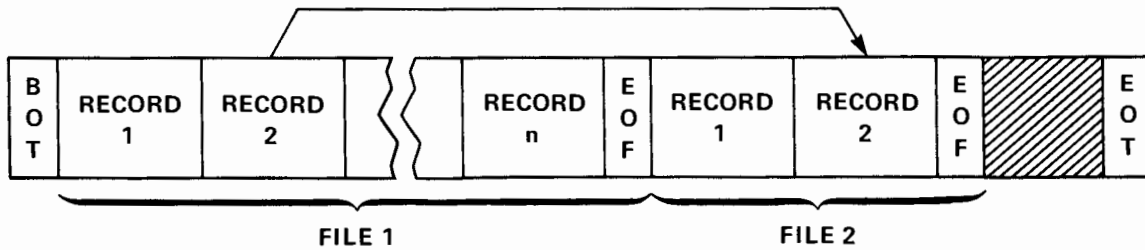
The record position is set to the next record in file 1 by incrementing RECD'POSITION with the statement

```
RECD'POSITION:=RECD'POSITION+1;
```

and spaces ahead to that record with the statement

```
FSPACE(MT,RECD'POSITION);
```

and the copy loop is repeated. After the copy loop is repeated, the tape is as follows:



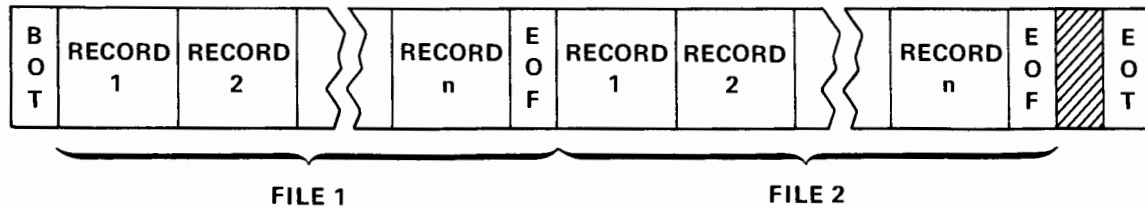
Note that the reverse tape motion after a write creates an EOF mark (see end of file 2).

The copy loop is repeated until the end of file 1 is reached, at which point program control is transferred to the statement label DONE. The tape then is rewound with the statement

```
FCONTROL(MT,5,DUMMY);
```

and closed with the same disposition (old permanent) as before.

The format of the tape at the end of the copy operation is as follows:



## SPACING ON DISC OR TAPE FILES

You can space forward or backward on a disc or tape file (containing non-variable-length records) with the FSPACE intrinsic. (This intrinsic resets the logical record pointer.)

In figure 3-25, the statement

```
FSPACE(MT,RECD'POSITION);
```

is used to space a tape file. The parameters specified are

<i>filename</i>	Supplied by MT, which was assigned the file number when the FOPEN intrinsic opened the file.
<i>displacement</i>	Specified by RECD'POSITION, which signifies the number of logical records to be spaced and the direction of displacement. (A positive value signifies forward spacing, a negative value signifies backward spacing.)

In the example, the value of RECD'POSITION is positive and the spacing is forward. RECD'POSITION is incremented by 1 each time the copy loop is executed, resulting in a sequential spacing of the file, thus enabling the program to read logical record 0, then 1, 2, and so forth.

## DIRECTING FILE CONTROL OPERATIONS

You can perform various control operations on a file (or the device on which the file resides) by issuing the FCONTROL intrinsic call. These operations include: supplying a printer or terminal carriage-control directive, verifying input/output, reading the hardware status word pertaining to the device on which the file resides, setting a terminal's time-out interval, rewinding the file, writing an end-of-file indicator, and skipping forward or backward to a tape mark. (The FCONTROL intrinsic can also be used to perform various terminal functions, such as changing the terminal speed or enabling parity checking. These applications of FCONTROL are described in Section V.)

Figure 3-25 contains four examples of FCONTROL which manipulate a tape file.

The first statement (statement number 29)

```
FCONTROL(MT,7,DUMMY);
```

spaces forward to a tape mark (EOF).

The next two statements (statement numbers 36 and 38) are both as follows:

```
FCONTROL(MT,8,DUMMY);
```

These statements space backward to tape marks. The first statement finds the EOF mark (the head was positioned beyond the EOF mark) and the second statement spaces backward again to find the beginning of the same file.

The last FCONTROL statement in the program

```
FCONTROL(MT,5,DUMMY);
```

rewinds the tape.

Note that the parameter DUMMY has no function in the application of FCONTROL in figure 3-25 and is supplied merely because all parameters of FCONTROL are required parameters.

## RESETTING THE LOGICAL RECORD POINTER

You can reset the logical record pointer for a disc file, containing only fixed-length records, to any logical record in the file with the FPOINT intrinsic. When the next FREAD or FWRITE request is issued for the file, this record will be the one read or written.

As an example, to position the logical record pointer to the 40th logical record in the file FILE1, you would use the following intrinsic call:

```
FPOINT(FILE1,39D);
```

Remember that the first logical record in a file is record zero and that the D suffix denotes a double integer value in SPL.

## DECLARING ACCESS-MODE OPTIONS

You can activate or deactivate the following access-mode options by issuing the FSETMODE intrinsic call: automatic error recovery, critical output verification, terminal control by the user, and terminal binary data mode. The access-mode options established remain in effect until another FSETMODE call is issued or until the file is closed. The FSETMODE intrinsic applies to files on all devices.

The following FSETMODE intrinsic call

```
FSETMODE(FILE1,%22);
```

establishes access-mode options as outlined below. The parameters specified are

*filenum* Designated by FILE1, which was assigned the file number by FOPEN when the file was opened.

*modeflags* %22, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Bits
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	Binary
										2		2		Octal		

The above bit pattern specifies the following access-mode options:

### Critical Output Verification:

All output to the file is to be verified as physically complete before control returns from a write intrinsic to your program. For each successful logical write operation, a condition code (CCE) is returned *immediately* to your program. Bit (14:1) = 1.



#### Tape Error Recovery:

A recovered tape error is reported with the CCE condition code. Bit (12:1) = 0.

#### Terminal Binary Data Mode:

The full 8 bits are transmitted and received through the asynchronous terminal controller. Bit (11:1) = 1.

## DETERMINING INTERACTIVE AND DUPLICATIVE FILE PAIRS

An input file and a list file are said to be *interactive* if a real-time dialogue can be established between a program and a person using the list file as a channel for programmatic requests, with appropriate responses from a person using the input file. For example, an input file and a list file opened to the same teleprinting terminal (for a session) would constitute an interactive pair. An input file and a list file are said to be *duplicative* when input from the former is duplicated automatically on the latter. For example, input from a card reader is printed on a line printer.

You can determine whether a pair of files is interactive or duplicative with the FRELATE intrinsic call. (The interactive/duplicative attributes of a file pair do not change between the time it is opened and the time it is closed.)

The FRELATE intrinsic applies to files on all devices.

To determine if the input file INFILE and the list file LISTFILE are interactive or duplicative, you could issue the following FRELATE intrinsic call.

```
ABLE:=FRELATE(INFILE,LISTFILE);
```

INFILE and LISTFILE are identifiers specifying the file numbers of the two files. (The file numbers were assigned to INFILE and LISTFILE when the FOPEN intrinsic opened the files.)

A word is returned to ABLE showing whether the files are interactive or duplicative. The word returned contains two significant bits, 0 and 15.

If bit 15 = 1, INFILE and LISTFILE form an interactive pair.

If bit 15 = 0, INFILE and LISTFILE do not form an interactive pair.

If bit 0 = 1, INFILE and LISTFILE form a duplicative pair.

If bit 0 = 0, INFILE and LISTFILE do not form a duplicative pair.

# UTILITY FUNCTIONS OF MPE INTRINSICS

SECTION

IV

MPE intrinsics allow you to perform the following utility functions.

- Manage library procedures with `LOADPROC` (see page 4-2) and `UNLOADPROC` (see page 4-3).
- Convert numbers from ASCII to binary code with `BINARY` and `DBINARY`. See page 4-13.
- Convert numbers from binary to ASCII code with `ASCII` and `DASCII`. See page 4-10.
- Convert a string of characters from EBCDIC to ASCII or from ASCII to EBCDIC with `CTranslate`. See page 4-13.
- Read input from job/session list devices with `READ` and `READX`. See page 4-16.
- Write output to the job/session list device with `PRINT`. See page 4-16.
- Write output to the Operator's Console with `PRINTOP` or write output to the Operator's Console and solicit a reply with `PRINTOPREPLY`. See page 4-18.
- Obtain system timer information with `TIMER`. See page 4-42.
- Obtain the calendar date with `CALENDAR`. See page 4-44.
- Obtain the time of day in terms of hour, minute, second, and tenth of second with `CLOCK`. See page 4-44.
- Obtain process run-time (CPU time) with `PROCTIME`. See page 4-44.
- Obtain information pertaining to your access mode and attributes with `WHO`. See page 4-10.
- Search an array for a specified name with `SEARCH`. See page 4-3.
- Format the parameters of a non-MPE command with `MYCOMMAND`. See page 4-4.
- Execute MPE commands programmatically with `COMMAND`. See page 4-9.
- Enable or disable hardware arithmetic traps with `ARITRAP`. See page 4-30.
- Enable or disable software arithmetic traps with `XARITRAP`. See page 4-32.
- Enable or disable the software library trap with `XLIBTRAP`. See page 4-35.
- Enable or disable the software system trap with `XSYSTRAP`. See page 4-36.
- Disarm the `CONTROL-Y` trap with `RESETCONTROL` (see page 4-40) or arm the `CONTROL-Y` trap with `XCONTRAP` (see page 4-41).
- Change the size of the current DL-to-DB area with `DLSIZE`. See page 4-22.
- Change the size of the current Z-to-DB area with `ZSIZE`. See page 4-27).
- Suspend the calling process with `PAUSE`. See page 4-19.
- Initiate a session break programmatically with `CAUSEBREAK`. See page 4-19.
- Programmatically terminate a process (after successful execution) with `TERMINATE`. See page 4-20.
- Programmatically abort any process within a user process structure with `QUIT`. See page 4-20.
- Abort the entire process structure (program) with `QUITPROG`. See page 4-20.
- Manage interprocess communication through the job control word with `SETJCW` (see page 4-45) and `GETJCW` (see page 4-45).

## DYNAMIC LOADING AND UNLOADING OF LIBRARY PROCEDURES

Normally, segments containing library procedures referenced by a program are attached to that program when the program is allocated in virtual memory. However, you also may dynamically attach and detach such procedures while your program is running. You might, for example, decide to do this for a large procedure used optionally and infrequently by your program, or for a procedure whose name is not known at load time. By loading this procedure only when it is required, and then unloading it, you can save the table entry. The procedures are loaded from segmented libraries, not from relocatable libraries (which are used only at program-preparation time).

### NOTE

Preparation and maintenance of segmented libraries and relocatable libraries is explained in the *Segmenter Reference Manual*.

You need not load procedures dynamically that are declared as externals to your program, because the loader will load them automatically. Dynamic loading and unloading is intended for procedures that are not declared at all.

### DYNAMIC LOADING

The LOADPROC intrinsic is used to load a library procedure, together with external procedures referenced by it.

For example, to dynamically load a procedure named PROC1, you could enter the following intrinsic call:

```
PNUM:=LOADPROC(PNAME,0,LAB);
```

The parameters specified in the above intrinsic call are

*procname* Contained in the byte array PNAME. The contents of PNAME are "PROC1". Note that the last character is a blank.

*lib* 0, signifying that only the system library should be searched.  
If 2 were specified, library searching would proceed in this order:  
Group Library  
Account Public Library  
System Library  
Specifying 1 for the *lib* parameter would cause the search to be conducted in this order:  
Account Public Library  
System Library

*plabel* LAB, a word to which the procedure's label (P label) is returned.

When the LOADPROC intrinsic executes, the procedure identity number will be returned as an integer to PNUM.

## DYNAMIC UNLOADING

The UNLOADPROC intrinsic is used to unload a procedure and its referenced external procedures.

For example, to unload the procedure that was dynamically loaded in the previous example, you could use the following UNLOADPROC intrinsic call:

```
UNLOADPROC(PNUM);
```

## SEARCHING ARRAYS

Occasionally, you may construct byte arrays whose contents you may later want to search for specified entries or names. A dictionary of user-designed commands is one such example. The searching is accomplished with the SEARCH intrinsic, which can be used with specially-formatted arrays consisting of sequential entries, each including:

- An integer specifying the length (in bytes) of the entire entry — the length includes this byte plus all the information in the following byte areas.
- An integer specifying the length of the “name” (in bytes) for which the search is performed.
- A byte string forming the name for which the search is performed — in a command dictionary, for example, this would be the command name.
- An optional byte string containing a user-supplied definition.

The entry number of the first entry in such a dictionary is *one*. The last entry is indicated by a *zero*.

As an example, consider a byte array wherein the relationship of each entry to its definition is:

Entry	Definition	
5,3,“IN”,	1,	Note: 0 = no parameter
6,4,“OUT”,	1,	1 = file name
7,5,“SKIP”,	2,	2 = numeric
7,5,“EXIT”,	0,	

You can request the search of such an array for a specified name with the SEARCH intrinsic. A simple linear search is performed, with the name, specified as a byte array, compared against the byte array forming the name in each entry. Because the search is linear, the most frequently used byte arrays should appear at the beginning of the array to promote efficient searching. If the name is found, the number of the entry containing the name is returned to the calling program. If the name is not found, a zero is returned. Optionally, you also can request the return of a pointer to the definition information for the name.

If you want to search the byte array, named DEFF, for the byte array NAME, containing the string “AN”, the following intrinsic call could be used.

```
ENUM:=SEARCH(NAME,2,DEFF,DEFADDR);
```

The length of the string in NAME is 2 bytes. The byte address of the definition sought is to be returned to the word DEFADDR. The entry number corresponding to the entry containing “AN” will be returned to the word ENUM.

When the intrinsic executes, ENUM contains 3 ("AN" is the third entry in the array) and DEFADDR contains the byte pointer to "XYZ".

## FORMATTING COMMAND PARAMETERS

You can programmatically extract and format for execution the parameters of a command defined by you (i.e., the command is *not* an MPE command) with the MYCOMMAND intrinsic. Additionally, you can have the MYCOMMAND intrinsic search a byte array for the specified command.

Figure 4-1 contains a program that checks if the user is running the program in a session and, if such is the case, performs the following:

1. Prompts the user to enter a command name from the terminal.
2. Reads the command name typed in by the user.
3. Compares this command name against entries in a byte array. If no match is found, the program displays  
    ILLEGAL ENTRY  
and prompts the user for another command.
4. Converts the parameter, entered with the command, to binary, then uses this operand to perform the calculation specified by the command.
5. Converts the result to ASCII, then displays the result on the terminal.

The statement

```
LGTH:=READ(INPUT,-72);
```

reads the command entered by the user (the arrays INPUT and COMMAND have been equivalenced, see statement 6 in the program).

The two statements

```
IF <> THEN QUIT(1);  
IF COMMAND = "END" THEN GO EXIT;
```

perform the following:

1. Check for a condition code error and execute the QUIT intrinsic (causing the process to abort if a condition code error is returned).
2. Cause program control to transfer to the statement EXIT if "END" is entered by the user.

The statement

```
COMMAND(LGTH):=%15;
```

adds a carriage-return character as the last character of the *comimage* parameter for the command entered. Note that the carriage-return character is added starting at the position in the array specified by LGTH, but does not overwrite the last position of the string entered by the user. The

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 ARRAY HEADING(0:8):="INTEGER CALCULATOR";
00004000 00011 1 ARRAY ERRMSG(0:6):="ILLEGAL ENTRY.";
00005000 00007 1 ARRAY INPUT(0:36);
00006000 00007 1 BYTE ARRAY COMMAND(*)=INPUT;
00007000 00007 1 BYTE ARRAY ANSWER(0:13):="ACCUM = ";
00008000 00010 1 ARRAY OUTPUT(*)=ANSWER;
00009000 00010 1 BYTE ARRAY TABLE(0:25):=
00010000 00001 1 5,3,"ADD", 5,3,"SUB", 5,3,"MUL",
00010100 00010 1 5,3,"DIV", 5,3,"SET", 0;
00011000 00016 1 INTEGER ARRAY PARMINFO(0:1);
00012000 00016 1 LOGICAL INTERACTIVE:=FALSE;
00013000 00016 1 INTEGER ACCUM:=0, OPERAND:=0, REQ="?",
00014000 00016 1 LGTH, INDX, PARMCNT, TYPE;
00015000 00016 1
00016000 00016 1 INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00017000 00016 1
00019000 00016 1 <<END OF DECLARATIONS>>
00020000 00016 1
00021000 00016 1 PRINT(HEADING,9,0); <<PROGRAM ID>>
00022000 00004 1 WHO(INTERACTIVE); <<LIVE USER?>>
00023000 00010 1 LOOP:
00024000 00010 1 IF INTERACTIVE THEN PRINT(REQ,1,%320); <<PROMPT USER>>
00025000 00016 1 LGTH:=READ(INPUT,-72); <<GET COMMAND>>
00026000 00023 1 IF <> THEN QUIT(1); <<CHECK FOR ERROR>>
00027000 00026 1 IF COMMAND="END" THEN GO EXIT; <<DONE - EXIT>>
00028000 00040 1 COMMAND(LGTH):=%15; <<CARRIAGE RETURN>>
00028100 00043 1
00029000 00043 1 TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART COMMAND>>
00030000 00050 1 PARMINFO,TABLE);
00031000 00056 1 IF < THEN GO ERROR; <<NO COMMAND MATCH>>
00032000 00057 1 IF PARMCNT<>1 THEN GO ERROR; <<NO PARAMETERS>>
00033000 00062 1 INDX:=PARMINFO-@COMMAND; <<SUBSCRIPT OF PARM>>
00034000 00065 1 OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARAMETER>>
00035000 00070 1 PARMINFO(1),(0:8));
00036000 00075 1 IF <> THEN GO ERROR; <<CHECK FOR ERROR>>
00036100 00076 1
00037000 00076 1 CASE (TYPE-1) OF <<SELECT OPERATION>>
00038000 00100 1 BEGIN
00039000 00106 2 ACCUM:=ACCUM+OPERAND; <<ADD COMMAND>>
00040000 00116 2 ACCUM:=ACCUM-OPERAND; <<SUB COMMAND>>
00041000 00122 2 ACCUM:=ACCUM*OPERAND; <<MUL COMMAND>>
00042000 00126 2 ACCUM:=ACCUM/OPERAND; <<DIV COMMAND>>
00043000 00133 2 ACCUM:=OPERAND; <<SET COMMAND>>
00044000 00136 2 END;
00045000 00143 1 RESULT:
00046000 00143 1 MOVE ANSWER(8):=" "; <<RESET OLD ANSWER>>
00047000 00155 1 ASCII(ACCUM,10,ANSWER(8)); <<CONVERT ACCUM>>
00048000 00163 1 PRINT(OUTPUT,7,0); <<OUTPUT NEW ANSWER>>
00049000 00170 1 GO LOOP; <<CONTINUE CALCULATION>>
00050000 00171 1 ERROR:
00051000 00171 1 PRINT(ERRMSG,7,0); <<ERROR MESSAGE>>
00052000 00175 1 IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVE USER-QUIT>>
00053000 00201 1 GO LOOP; <<CONTINUE CALCULATION>>
00054000 00202 1 EXIT: END.
PRIMARY DB STORAGE=%020; SECONDARY DB STORAGE=%00113
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:11

```



Figure 4-1. Using the MYCOMMAND Intrinsic

reason for this is that, in SPL convention, the first position in an array is 0, not 1. For example, if the user entered the command ADD 5, this command would occupy array positions 0 through 4, as follows:

```
0  1  2  3  4
A  D  D      5
```

The value returned to LGTH specifying the length of the string read, however, is 5 because the READ statement read a string 5 characters long and therefore the carriage-return character is added to position 5 of the array.

The statement

```
TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT,PARMINFO,TABLE);
```

calls the MYCOMMAND intrinsic to parse the command entered by the user. The parameters specified are

*comimage* Contained in the array COMMAND, which contains the string entered by the user. This parameter contains a user command such as ADD or SUB, a parameter consisting of an integer, and a carriage-return character added to the command by the statement

```
COMMAND(LGTH):=%15;
```

For example, the complete *comimage* parameter could be ADD 15%15, with ADD 15 entered by the user and the %15 carriage return added programmatically.

*delimiters* Omitted. The default delimiter array “comma, equal, semicolon, carriage return” is used.

*maxparms* 1, specifying that one parameter is expected in *comimage*.

*numparms* Specified by PARMCNT, which contains the actual number of parameters entered with the command.

*parms* Specified by PARMINFO, an integer array to which is returned the byte address of the parameter entered as part of *comimage*.

#### NOTE

Although this parameter is listed as a double array in the specifications for the MYCOMMAND intrinsic in Section II, it is declared as a two-word integer array in this program because it is necessary to access each of the words individually. This is more convenient than declaring a one-word double array and a two-word integer array, then equivalencing the two.

*dict* Specified by TABLE, a byte array containing 5,3,"ADD", 5,3,"SUB", 5,3,"MUL", 5,3,"DIV", 5,3,"SET", 0;

The table specified by the *dict* parameter is searched until a match is found between the command name and an entry in the table. If a match is found, the number of the entry in the table containing the matching name is returned to TYPE. The *dict* parameter specifies a specially-formatted array, or table. Each entry in the table contains:

1. An integer specifying the total number of bytes in the entry.
2. An integer specifying the total number of characters in the command portion of the entry.
3. The command portion of the entry.
4. An arbitrary user-defined definition of the entry.

For example, the first entry in the array TABLE is

5,3,ADD

which is broken down as follows:

- 5 The total number of bytes in this entry (53ADD = 5 bytes).
- 3 The total number of bytes in the command portion (ADD) of the entry.
- ADD The string comprising the command portion of the entry.

Note that a user-defined definition of the entry is not included in the entries in TABLE.

The byte array TABLE, then, consists of 26 bytes structured as follows:

5	3
A	D
D	5
3	S
U	B
5	3
M	U
L	5
3	D
I	V
5	3
S	E
T	0



The statement

```
IF < THEN GO ERROR;
```

checks the condition code and, if it is CCL, transfers program control to statement label ERROR.

The statement

```
IF PARMCNT < > 1 THEN GO ERROR;
```

checks that only one parameter was entered with the command (the parameter *maxparms* had specified that one parameter was expected). If PARMCNT does not equal 1, control is transferred to statement label ERROR.

The two statements

```
INDX:=PARMINFO-@COMMAND;  
OPERAND:=BINARY(COMMAND(INDX),PARMINFO(1).(0:8));
```

determine the byte address of the parameter entered with the command, then convert this parameter to a binary value.

The first statement above

```
INDX:=PARMINFO-@COMMAND;
```

subtracts the byte address of the first element of COMMAND from the byte address of PARMINFO to obtain the relative position of the parameter in the array COMMAND. This value is returned to INDX. For example, the command

```
ADD 5
```

would occupy positions in the array COMMAND as follows:

0	1	2	3	4
A	D	D		5

Subtracting the byte address of the first (zero) element of COMMAND from the byte address specified by PARMINFO for the first element of the parameter produces the byte address of the parameter.

The statement

```
OPERAND:=BINARY(COMMAND(INDX),PARMINFO(1).(0:8));
```

converts the ASCII characters starting in the INDX position of the array COMMAND to a binary value and returns this value to OPERAND. The number of bytes (length) of the ASCII string to be converted are specified by the first eight bits (PARMINFO(1).(0:8)) of the first word contained in PARMINFO

The statement

### CASE (TYPE-1) OF

transfers program control to one of the five statements following the BEGIN statement, depending on the value of TYPE-1. Note that -1 is necessary because the five statements are considered in the SPL numbering convention by the CASE statement (ACCUM:=ACCUM+OPERAND; is considered to be the zeroth statement following BEGIN) but the value assigned to TYPE by MYCOMMAND contains the range 1 to 5.

An example of running the program is shown below.

```
:RUN UTILY
  
INTEGER CALCULATOR
? SET 10
ACCUM = 10
? ADD 34
ACCUM = 44
? MUL .5
ILLEGAL ENTRY
? MUL 2
ACCUM = 88
? END
  
END OF PROGRAM
```

## EXECUTING MPE COMMANDS PROGRAMMATICALLY

The COMMAND intrinsic can be used to programmatically request the execution of certain MPE commands. The command image, including parameters, is passed to the intrinsic, which searches the system command dictionary for a command of the same name, and executes it. When command execution is completed, or when an error is detected during this execution, control returns to the calling process. Commands that can be executed programmatically are listed below.

:ALTSEC	:SAVE
:BUILD	:SECURE
:COMMENT	:SETDUMP
:FILE	:SETMSG
:GETRIN	:SHOWDEV
:LISTF	:SHOWIN
:PTAPE	:SHOWJOB
:PURGE	:SHOWOUT
:RELEASE	:SHOWTIME
:RENAME	:SPEED
:REPORT	:STORE
:RESET	:STREAM
:RESETDUMP	:TELL
:RESTORE	:TELLOP

See the *MPE Commands Reference Manual* for discussions of commands.

If you want to programmatically execute the command :SHOWTIME, the following intrinsic call could be used:

```
COMMAND(COMD,ECODE,EPARM);
```

All characters for the command except the prompting colon are contained in the byte array COMD. Any error code is returned to ECODE. Since the :SHOWTIME command has no parameters, no information is returned to EPARM.

When the intrinsic executes, the date and time are printed on the job/session list device.

#### NOTE

Warning messages issued by the command executor are transmitted to \$STDLIST.

## DETERMINING THE USER'S ACCESS MODE AND ATTRIBUTES

A program can obtain the access mode and attributes of the user running that program from the system tables with the WHO intrinsic.

Figure 4-2 contains a program which must determine if the user is running the program in an interactive session. The statement

```
WHO(INTERACTIVE);
```

calls the WHO intrinsic to make this determination. If the logical identifier INTERACTIVE is TRUE (bit 15 = 1) after the WHO intrinsic executes, and job/session input file and job/session list file form an interactive pair, thus the user is running the program interactively.

The statement

```
IF INTERACTIVE THEN PRINT(REQ,1,%320);
```

checks whether INTERACTIVE is TRUE or FALSE. If TRUE, the PRINT portion of the statement is executed and a prompt character (?) is displayed on the terminal to prompt the user for a command.

## CONVERTING NUMBERS FROM BINARY CODE TO ASCII STRINGS

You can convert a one-word binary number to an octal or decimal number represented as an ASCII string with the ASCII intrinsic. The length of the resulting ASCII string can be returned as an integer value.

The ASCII intrinsic call is illustrated in figure 4-3. The statement

```
ASCII(ACCUM,10,ANSWER(8));
```

converts the one-word binary number contained in ACCUM to the base 10 and places the converted value into the 8th element of the byte array ANSWER. The length of the resulting ASCII string is

```

00001000 00000 0  SCONTROL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  ARRAY HEADING(0:8):="INTEGER CALCULATOR";
00004000 00011 1  ARRAY FRRMSG(0:6):="ILLEGAL ENTRY.";
00005000 00007 1  ARRAY INPUT(0:36);
00006000 00007 1  BYTE ARRAY COMMAND(*)=INPUT;
00007000 00007 1  BYTE ARRAY ANSWER(0:13):="ACCUM =      ";
00008000 00010 1  ARRAY OUTPUT(*)=ANSWER;
00009000 00010 1  BYTE ARRAY TABLE(0:25):=
00010000 00001 1      5,3,"ADD",      5,3,"SUB",      5,3,"MUL",
00010100 00010 1      5,3,"DIV",      5,3,"SET",      0;
00011000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00012000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00013000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00014000 00016 1      LGTH,      INDX,      PARMCNT,      TYPE;
00015000 00016 1
00016000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00017000 00016 1
00019000 00016 1  <<END OF DECLARATIONS>>
00020000 00016 1
00021000 00016 1      PRINT(HEADING,9,0);      <<PROGRAM ID>>
00022000 00004 1      WHO(INTERACTIVE);      <<LIVE USER?>>
00023000 00010 1  LOOP:
00024000 00010 1      IF INTERACTIVE THEN PRINT(REQ,1,%320); <<PROMPT USER>>
00025000 00016 1      LGTH:=READ(INPUT,-72);      <<GET COMMAND>>
00026000 00023 1      IF <> THEN QUIT(1);      <<CHECK FOR ERROR>>
00027000 00026 1      IF COMMAND="END" THEN GO EXIT;      <<DONE - EXIT>>
00028000 00040 1      COMMAND(LGTH):=%15;      <<CARRIAGE RETURN>>
00028100 00043 1
00029000 00043 1      TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT,      <<TAKE APART COMMAND>>
00030000 00050 1      PARMINFO,TABLE);
00031000 00056 1      IF < THEN GO ERROR;      <<NO COMMAND MATCH>>
00032000 00057 1      IF PARMCNT<>] THEN GO ERROR;      <<NO PARAMETERS>>
00033000 00062 1      INDX:=PARMINFO=@COMMAND;      <<SUBSCRIPT OF PARM>>
00034000 00065 1      OPERAND:=BINARY(COMMAND(INDX),      <<CONVERT PARAMETER>>
00035000 00070 1      PARMINFO(1).(0:8));
00036000 00075 1      IF <> THEN GO ERROR;      <<CHECK FOR ERROR>>
00036100 00076 1
00037000 00076 1  CASE (TYPE-1) OF      <<SELECT OPERATION>>
00038000 00100 1  BEGIN
00039000 00106 2      ACCUM:=ACCUM+OPERAND;      <<ADD COMMAND>>
00040000 00116 2      ACCUM:=ACCUM-OPERAND;      <<SUB COMMAND>>
00041000 00122 2      ACCUM:=ACCUM*OPERAND;      <<MUL COMMAND>>
00042000 00126 2      ACCUM:=ACCUM/OPERAND;      <<DIV COMMAND>>
00043000 00133 2      ACCUM:=OPERAND;      <<SET COMMAND>>
00044000 00136 2  END;
00045000 00143 1  RESULT:
00046000 00143 1      MOVE ANSWER(8):="      ";      <<RESET OLD ANSWER>>
00047000 00155 1      ASCII(ACCUM,10,ANSWER(8));      <<CONVERT ACCUM>>
00048000 00163 1      PRINT(OUTPUT,7,0);      <<OUTPUT NEW ANSWER>>
00049000 00170 1      GO LOOP;      <<CONTINUE CALCULATION>>
00050000 00171 1  ERROR:
00051000 00171 1      PRINT(ERRMSG,7,0);      <<ERROR MESSAGE>>
00052000 00175 1      IF NOT INTERACTIVE THEN QUIT(2);      <<NO LIVE USER-QUIT>>
00053000 00201 1      GO LOOP;      <<CONTINUE CALCULATION>>
00054000 00202 1  EXIT: END.
PRIMARY DB STORAGE=%020; SECONDARY DB STORAGE=%00113
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:11

```

Figure 4-2. Using the WHO Intrinsic

```

00001000 00000 0  SCONTROL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  ARRAY HEADING(0:8):="INTEGER CALCULATOR";
00004000 00011 1  ARRAY ERRMSG(0:6):="ILLEGAL ENTRY.";
00005000 00007 1  ARRAY INPUT(0:36);
00006000 00007 1  BYTE ARRAY COMMAND(*)=INPUT;
00007000 00007 1  BYTE ARRAY ANSWER(0:13):="ACCUM =      ";
00008000 00010 1  ARRAY OUTPUT(*)=ANSWER;
00009000 00010 1  BYTE ARRAY TABLE(0:25):=
00010000 00001 1      5,3,"ADD",      5,3,"SUB",      5,3,"MUL",
00010100 00010 1      5,3,"DIV",      5,3,"SET",      0;
00011000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00012000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00013000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00014000 00016 1      LGTH,      INDX,      PARMCNT,      TYPE;
00015000 00016 1
00016000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00017000 00016 1
00019000 00016 1  <<END OF DECLARATIONS>>
00020000 00016 1
00021000 00016 1  PRINT(HEADING,9,0); <<PROGRAM ID>>
00022000 00004 1  WHO(INTERACTIVE); <<LIVE USER?>>
00023000 00010 1  LOOP:
00024000 00010 1  IF INTERACTIVE THEN PRINT(REQ,1,%320); <<PROMPT USER>>
00025000 00016 1  LGTH:=READ(INPUT,-72); <<GET COMMAND>>
00026000 00023 1  IF <> THEN QUIT(1); <<CHECK FOR ERROR>>
00027000 00026 1  IF COMMAND="END" THEN GO EXIT; <<DONE - EXIT>>
00028000 00040 1  COMMAND(LGTH):=%15; <<CARRIAGE RETURN>>
00028100 00043 1
00029000 00043 1  TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART COMMAND>>
00030000 00050 1      PARMINFO,TABLE);
00031000 00056 1  IF < THEN GO ERROR; <<NO COMMAND MATCH>>
00032000 00057 1  IF PARMCNT<>1 THEN GO ERROR; <<NO PARAMETERS>>
00033000 00062 1  INDX:=PARMINFO-@COMMAND; <<SUBSCRIPT OF PARM>>
00034000 00065 1  OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARAMETER>>
00035000 00070 1      PARMINFO(1).(0:8));
00036000 00075 1  IF <> THEN GO ERROR; <<CHECK FOR ERROR>>
00036100 00076 1
00037000 00076 1  CASE (TYPE-1) OF <<SELECT OPERATION>>
00038000 00100 1  BEGIN
00039000 00106 2  ACCUM:=ACCUM+OPERAND; <<ADD COMMAND>>
00040000 00116 2  ACCUM:=ACCUM-OPERAND; <<SUB COMMAND>>
00041000 00122 2  ACCUM:=ACCUM*OPERAND; <<MUL COMMAND>>
00042000 00126 2  ACCUM:=ACCUM/OPERAND; <<DIV COMMAND>>
00043000 00133 2  ACCUM:=OPERAND; <<SET COMMAND>>
00044000 00136 2  END;
00045000 00143 1  RESULT:
00046000 00143 1  MOVE ANSWER(8):="      "; <<RSET OLD ANSWER>>
00047000 00155 1  ASCII(ACCUM,10,ANSWER(8)); <<CONVERT ACCUM>>
00048000 00163 1  PRINT(OUTPUT,7,0); <<OUTPUT NEW ANSWER>>
00049000 00170 1  GO LOOP; <<CONTINUE CALCULATION>>
00050000 00171 1  ERROR:
00051000 00171 1  PRINT(ERRMSG,7,0); <<ERROR MESSAGE>>
00052000 00175 1  IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVE USER=QUIT>>
00053000 00201 1  GO LOOP; <<CONTINUE CALCULATION>>
00054000 00202 1  EXIT: END.
PRIMARY DB STORAGE=%020; SECONDARY DB STORAGE=%00113
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:11

```

Figure 4-3. Using the ASCII Intrinsic

unimportant in this application and therefore no variable is provided in the intrinsic call for this return. If the length were desired, the intrinsic call could have had the form

```
LGTH:=ASCII(ACCUM,10,ANSWER(8));
```

The DASCII intrinsic, which converts a double-word (32-bit) binary number to an octal or decimal number represented as an ASCII string, is shown in figure 4-4.

The statement

```
LGTH:=DASCII(CNTR,10,BMSG(20));
```

converts the 32-bit binary number contained in CNTR to the base 10 and places the converted decimal value in the 20th element of byte array BMSG. The length (number of characters) of the converted value is returned to LGTH.

The value is converted from binary to ASCII so that it can be printed by the PRINT statement.

## **CONVERTING NUMBERS FROM AN ASCII NUMERIC STRING TO A BINARY CODED VALUE**

The BINARY intrinsic converts an ASCII numeric string to its equivalent binary value. The converted value is returned to the calling program.

The BINARY intrinsic call is illustrated in figure 4-5. The statement

```
OPERAND:=BINARY(COMMAND(INDX),PARMINFO(1).(0:8));
```

converts the ASCII numeric string contained in the element specified by INDX of the array COMMAND to its binary equivalent. The length of the ASCII string is specified by the first 8 bits of the first word of the array PARMINFO. The resulting binary value is stored in the word OPERAND.

To convert a number from an ASCII string to a double-word (32-bit) binary value, the DBINARY intrinsic is used. A DBINARY intrinsic call could be of the form

```
DVAL:=DBINARY(STRING,LENGTH);
```

where STRING contains the octal or decimal number to be converted and LENGTH is an integer representing the length of the string containing the ASCII-coded value. The converted double-word value is returned to DVAL.

## **TRANSLATING CHARACTERS FROM EBCDIC TO ASCII OR FROM ASCII TO EBCDIC**

You can convert a string of characters represented in EBCDIC to its ASCII equivalent, and vice versa, with the CTRANSLATE intrinsic. In addition, you can use CTRANSLATE to convert to other codes by defining your own translation table.

```

00001000 00000 0  SCONTROL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  ARRAY HEADING(0:10):="CONTROL Y TRAP EXAMPLE";
00004000 00013 1  ARRAY MSG(0:15):="COUNTER CURRENTLY =          ";
00005000 00020 1  BYTE ARRAY HMSG(*)=MSG;
00006000 00020 1  DOUBLE CNTR:=0D;
00007000 00020 1  INTEGER DUMMY,LGTH;
00008000 00020 1
00009000 00020 1  INTRINSIC PRINT,XCONTRAP,QUIT,DASCII,RESETCONTROL;
00010000 00020 1
00011000 00020 1  PROCEDURE CONTROLY;
00012000 00000 1  BEGIN
00013000 00000 2  INTEGER SDEC=Q+1;
00014000 00000 2
00015000 00000 2  LGTH:=DASCII(CNTR,10,HMSG(20)); <<CONVERT COUNTER>>
00016000 00007 2  PRINT(MSG,16,0); <<OUTPUT COUNTER>>
00017000 00013 2  RESETCONTROL; <<PEARM CONTROL Y TRAP>>
00018000 00014 2  TOS:=%31400+SDEC; <<BUILD EXIT INSTRUCTION>>
00019000 00016 2  ASSEMBLE(XFQ 0); <<EXECUTE EXIT>>
00020000 00017 2  END;
00021000 00000 1
00022000 00000 1  <<END OF DECLARATIONS>>
00023000 00000 1
00024000 00000 1  PRINT(HEADING,11,0); <<PROGRAM ID>>
00025000 00004 1  XCONTRAP(@CONTROLY,DUMMY); <<ARM CONTROL Y TRAP>>
00026000 00007 1  IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00027000 00012 1  LOOP:
00028000 00012 1  CNTR:=CNTR+1D; <<DOUBLE INCREMENT>>
00029000 00023 1  IF CNTR<3000000D THEN GO LOOP; <<CONTINUOUS LOOP>>
00030000 00027 1  END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00033
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:02; ELAPSED TIME=0:00:26

```

Figure 4-4. Using the DASCII Intrinsic

As an example of converting from EBCDIC to ASCII, suppose the byte array ESTRING contains the EBCDIC characters "JOB 2". You want to convert this string to its ASCII equivalent and store it in the byte array ASTRING. The following intrinsic call could be used:

```
CTRANSLATE(1,ESTRING,ASTRING,5);
```

The parameters specified in the above intrinsic call are:

- code* 1, which specifies the EBCDIC-to-ASCII table. A 0 for this parameter specifies a user-defined translation table, and a 2 specifies the ASCII-to-EBCDIC table.
- instring* ESTRING, a byte array containing the string to be converted.
- outstring* ASTRING, a byte array which will contain the ASCII characters for "JOB 2" when the intrinsic is executed.

```

00001000 00000 0  SCONTPOL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  ARRAY HEADING(0:8):="INTEGER CALCULATOR";
00004000 00011 1  ARRAY ERRMSG(0:6):="ILLEGAL ENTRY.";
00005000 00007 1  ARRAY INPUT(0:36);
00006000 00007 1  BYTE ARRAY COMMAND(*)=INPUT;
00007000 00007 1  BYTE ARRAY ANSWER(0:13):="ACCUM =      ";
00008000 00010 1  ARRAY OUTPUT(*)=ANSWER;
00009000 00010 1  BYTE ARRAY TABLE(0:25):=
00010000 00001 1      5,3,"ADD",      5,3,"SUB",      5,3,"MUL",
00010100 00010 1      5,3,"DIV",      5,3,"SET",      0;
00011000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00012000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00013000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REG:="? ",
00014000 00016 1      LGTH,      INDX,      PARMCNT,      TYPE;
00015000 00016 1
00016000 00016 1  INTRINSIC ASCII,BINARY,HEAD,PRINT,MYCOMMAND,QUIT,WHO;
00017000 00016 1
00019000 00016 1  <<END OF DECLARATIONS>>
00020000 00016 1
00021000 00016 1      PRINT(HEADING,9,0);      <<PROGRAM ID>>
00022000 00004 1      WHO(INTERACTIVE);      <<LIVE USER?>>
00023000 00010 1  LOOP:
00024000 00010 1      IF INTERACTIVE THEN PRINT(REG,1,%320); <<PROMPT USER>>
00025000 00016 1      LGTH:=READ(INPUT,-72); <<GET COMMAND>>
00026000 00023 1      IF <> THEN QUIT(1); <<CHECK FOR ERROR>>
00027000 00026 1      IF COMMAND="END" THEN GO EXIT; <<DONE - EXIT>>
00028000 00040 1      COMMAND(LGTH):=%15; <<CARRIAGE RETURN>>
00028100 00043 1
00029000 00043 1      TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART COMMAND>>
00030000 00050 1          PARMINFO,TABLE);
00031000 00056 1      IF < THEN GO ERROR; <<NO COMMAND MATCH>>
00032000 00057 1      IF PARMCNT<>1 THEN GO ERROR; <<NO PARAMETERS>>
00033000 00062 1      INDX:=PARMINFO-@COMMAND; <<SUBSCRIPT OF PARM>>
00034000 00065 1      OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARAMETER>>
00035000 00070 1          PARMINFO(1).(0:8));
00036000 00075 1      IF <> THEN GO ERROR; <<CHECK FOR ERROR>>
00036100 00076 1
00037000 00076 1      CASE (TYPE-1) OF <<SELECT OPERATION>>
00038000 00100 1          BEGIN
00039000 00106 2              ACCUM:=ACCUM+OPERAND; <<ADD COMMAND>>
00040000 00116 2              ACCUM:=ACCUM-OPERAND; <<SUB COMMAND>>
00041000 00122 2              ACCUM:=ACCUM*OPERAND; <<MUL COMMAND>>
00042000 00126 2              ACCUM:=ACCUM/OPERAND; <<DIV COMMAND>>
00043000 00133 2              ACCUM:=OPERAND; <<SET COMMAND>>
00044000 00136 2          END;
00045000 00143 1  RESULT:
00046000 00143 1      MOVE ANSWER(8):="      "; <<RESET OLD ANSWER>>
00047000 00155 1      ASCII(ACCUM,10,ANSWER(8)); <<CONVERT ACCUM>>
00048000 00163 1      PRINT(OUTPUT,7,0); <<OUTPUT NEW ANSWER>>
00049000 00170 1      GO LOOP; <<CONTINUE CALCULATION>>
00050000 00171 1  ERROR:
00051000 00171 1      PRINT(ERRMSG,7,0); <<ERROR MESSAGE>>
00052000 00175 1      IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVE USER=QUIT>>
00053000 00201 1      GO LOOP; <<CONTINUE CALCULATION>>
00054000 00202 1  EXIT: END.
PRIMARY DB STORAGE=%020; SECONDARY DB STORAGE=%00113
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:11

```



Figure 4-5. Using the BINARY Intrinsic



*stringlength*

5, which specifies the length, in bytes, of the string "JOB 2".

*table*

Omitted. This parameter, if specified, consists of a byte array containing a user-defined table to be used in the translation.

## TRANSMITTING PROGRAM INPUT/OUTPUT FROM JOB/SESSION INPUT/OUTPUT DEVICES

In addition to the FREAD and FWRITE intrinsics discussed in Section III, MPE provides three other intrinsics that allow you to read information from the job/session input device (READ and READX intrinsics) or write information to the job/session list device (PRINT intrinsic). Additionally, two other intrinsics allow you to transmit a message to the Operator's Console (PRINTOP intrinsic), or transmit a message to the Operator's Console and solicit a reply (PRINTOREPLY intrinsic).

Please bear in mind that the READ, READX, and PRINT intrinsics are limited in their usefulness. The reason for this is that :FILE commands are not allowed with these intrinsics and the *filenum* parameter, obtained from the FOPEN intrinsic, is not available for use with these intrinsics. Usually, therefore, you will find it to be more convenient and better programming practice to use the FOPEN intrinsic to open the files \$STDIN and \$STDLIST, and then issue FREAD's and FWRITE's against these files.

### READING INPUT FROM THE JOB/SESSION INPUT DEVICE

The job/session input device is the source of all MPE commands relating to a job or session, and is the primary source of all ASCII information input to the job or session. Normally, the input device is a terminal for sessions and a card reader for jobs.

You can read a string of ASCII characters from the job/session input device into an array in your program with the READ and READX intrinsics. The READ and READX intrinsics are identical except that the READX intrinsic reads input from \$STDINX instead of \$STDIN. (\$STDINX is equivalent to \$STDIN except that records with a colon in column 1 indicate the end of data to \$STDIN and only the commands :EOD, :EOF, :JOB, :EOJ, and :DATA indicate the end of data for \$STDINX.)

The READ intrinsic call is illustrated in figure 4-6.

The statement

```
LGTH:=READ(INPUT,-72);
```

reads an entry from the terminal and transfers this string to the array INPUT. The maximum length of the string to be read is specified as 72 bytes (-72). The actual length of the string read is returned and stored in the word LGTH when the intrinsic executes.

The statement

```
IF < > THEN QUIT(1);
```

```

00001000 00000 0 sCONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 ARRAY HEADING(0:8):="INTEGER CALCULATOR";
00004000 00011 1 ARRAY ERRMSG(0:6):="ILLEGAL ENTRY.";
00005000 00007 1 ARRAY INPUT(0:36);
00006000 00007 1 BYTE ARRAY COMMAND(*)=INPUT;
00007000 00007 1 BYTE ARRAY ANSWER(0:13):="ACCUM = ";
00008000 00010 1 ARRAY OUTPUT(*)=ANSWER;
00009000 00010 1 BYTE ARRAY TABLE(0:25):=
00010000 00001 1 5,3,"ADD", 5,3,"SUB", 5,3,"MUL",
00010100 00010 1 5,3,"DIV", 5,3,"SET", 0;
00011000 00016 1 INTEGER ARRAY PARMINFO(0:1);
00012000 00016 1 LOGICAL INTERACTIVE:=FALSE;
00013000 00016 1 INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00014000 00016 1 LGTH, INDX, PARMCNT, TYPE;
00015000 00016 1
00016000 00016 1 INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00017000 00016 1
00019000 00016 1 <<END OF DECLARATIONS>>
00020000 00016 1
00021000 00016 1 PRINT(HEADING,9,0); <<PROGRAM ID>>
00022000 00004 1 WHO(INTERACTIVE); <<LIVE USER?>>
00023000 00010 1 LOOP:
00024000 00010 1 IF INTERACTIVE THEN PRINT(REQ,1,%320); <<PROMPT USER>>
00025000 00016 1 LGTH:=READ(INPUT,-72); <<GET COMMAND>>
00026000 00023 1 IF <> THEN QUIT(1); <<CHECK FOR ERROR>>
00027000 00026 1 IF COMMAND="END" THEN GO EXIT; <<DONE - EXIT>>
00028000 00040 1 COMMAND(LGTH):=%15; <<CARRIAGE RETURN>>
00028100 00043 1
00029000 00043 1 TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART COMMAND>>
00030000 00050 1 PARMINFO,TABLE);
00031000 00056 1 IF < THEN GO ERROR; <<NO COMMAND MATCH>>
00032000 00057 1 IF PARMCNT<>1 THEN GO ERROR; <<NO PARAMETERS>>
00033000 00062 1 INDX:=PARMINFO-@COMMAND; <<SUBSCRIPT OF PARM>>
00034000 00065 1 OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARAMETER>>
00035000 00070 1 PARMINFO(1).(0:8));
00036000 00075 1 IF <> THEN GO ERROR; <<CHECK FOR ERROR>>
00036100 00076 1
00037000 00076 1 CASE (TYPE-1) OF <<SELECT OPERATION>>
00038000 00100 1 BEGIN
00039000 00106 2 ACCUM:=ACCUM+OPERAND; <<ADD COMMAND>>
00040000 00116 2 ACCUM:=ACCUM-OPERAND; <<SUB COMMAND>>
00041000 00122 2 ACCUM:=ACCUM*OPERAND; <<MUL COMMAND>>
00042000 00126 2 ACCUM:=ACCUM/OPERAND; <<DIV COMMAND>>
00043000 00133 2 ACCUM:=OPERAND; <<SET COMMAND>>
00044000 00136 2 END;
00045000 00143 1 RESULT:
00046000 00143 1 MOVE ANSWER(8):=" "; <<RESET OLD ANSWER>>
00047000 00155 1 ASCII(ACCUM,10,ANSWER(8)); <<CONVERT ACCUM>>
00048000 00163 1 PRINT(OUTPUT,7,0); <<OUTPUT NEW ANSWER>>
00049000 00170 1 GO LOOP; <<CONTINUE CALCULATION>>
00050000 00171 1 ERROR:
00051000 00171 1 PRINT(ERRMSG,7,0); <<ERROR MESSAGE>>
00052000 00175 1 IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVE USER=QUIT>>
00053000 00201 1 GO LOOP; <<CONTINUE CALCULATION>>
00054000 00202 1 EXIT: END.
PRIMARY DB STORAGE=%020; SECONDARY DB STORAGE=%00113
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:11

```

Figure 4-6. Using the PRINT and READ Intrinsics

checks for a “not equal” condition code and, if CCG or CCL is returned, the QUIT intrinsic is executed and the process is aborted. The (1) parameter is an arbitrary user-supplied value that is displayed as part of the abort message.

## WRITING OUTPUT TO THE JOB/SESSION LIST DEVICE

Normally, the list device is a line printer for jobs and a terminal for sessions. You can write a string of ASCII characters from an array in your program to this list device with the PRINT intrinsic.

In figure 4-6, the statement

```
PRINT(HEADING,9,0);
```

transmits the string “INTEGER CALCULATOR” from the array HEADING (see statement number 3 in figure 4-6). The *length* parameter is specified as 9, which means that the string to be transmitted is 9 words long (a negative value would specify bytes). The *control* parameter is 0, signifying that the full record is to be printed, up to 132 characters per line, using single spacing.

## WRITING OUTPUT TO THE OPERATOR’S CONSOLE

The PRINTOP intrinsic can be used to transmit an ASCII string from an array in your program to the Operator’s Console. The ASCII string to be transmitted is limited to 56 characters.

The PRINTOP intrinsic could be called as follows:

```
PRINTOP(MESSAGE,10,%60);
```

The character string to be transmitted is contained in the array MESSAGE. The parameter 10 signifies that the message is 10 words long. A negative value for this parameter would specify bytes. If *zero* is specified for the *length* parameter, only the standard message prefix is written on the Operator’s Console; the string contained in MESSAGE would not be transmitted in this case. The parameter %60 is the carriage-control code for double space.

## WRITING OUTPUT TO THE OPERATOR’S CONSOLE AND REQUESTING A REPLY

The PRINTOPREPLY intrinsic can be used to transmit an ASCII string from an array in your program to the Operator’s Console and to request that a reply be returned. For example, a program could ask the operator if the line printer contains a certain type of form. If the response is affirmative, the program then could write information on these forms.

A PRINTOPREPLY intrinsic call could be as follows:

```
REPLGTH:=PRINTOPREPLY(MESSAGE,18,%320,REPLY,-3);
```

The following parameters were specified in the above call:

<i>message</i>	An ASCII string contained in the array MESSAGE.
<i>length</i>	18 words. A negative value would specify bytes. If <i>zero</i> is specified for the length parameter, only the standard message prefix is written on the

Operator's Console; the string contained in MESSAGE would not be transmitted in this case.

*control* %320, which signifies no space and no carriage return. The operator will respond on the same line, then press RETURN on the console.

*reply* The operator's reply will be returned to the array REPLY.

*expectedl* -3, signifying that the maximum expected length of the reply is 3 bytes. A positive value would specify words.

The actual length of the operator's reply is returned to REPLGTH. This is a positive value representing a byte count in this case because *expectedl* is negative (-3). If *expectedl* is positive, then the length returned represents words.

## SUSPENDING THE CALLING PROCESS

The calling process can be suspended with the PAUSE intrinsic. A PAUSE intrinsic call could be as follows:

```
PAUSE(INT);
```

INT is a real variable that specifies the amount of time, in seconds, that the process is suspended. The maximum interval allowed is approximately 2,147,484 seconds.

When INT seconds have elapsed, control is returned to the calling process and execution resumes at the statement following the PAUSE intrinsic call.

## REQUESTING A PROCESS BREAK

During a session, you can initiate a break programmatically with the CAUSEBREAK intrinsic. Using this intrinsic is the programmatic equivalent of using the BREAK key in a session, and allows you to enter certain MPE commands to perform functions such as creating a file or transmitting an informal message. The MPE commands permitted during a break are listed below:

:ABORT	:RESUME
:ALTSEC	:SAVE
:BUILD	:SECURE
:COMMENT	:SETDUMP
:FILE	:SETMSG
:GETRIN	:SHOWDEV
:LISTF	:SHOWIN
:PTAPE	:SHOWJOB
:PURGE	:SHOWOUT
:RELEASE	:SHOWTIME
:RENAME	:SPEED
:REPORT	:STORE
:RESET	:STREAM
:RESETDUMP	:TELL
:RESTORE	:TELLOP

See the *MPE Commands Reference Manual* for discussions of commands.

The form of the CAUSEBREAK intrinsic call is

```
CAUSEBREAK;
```

Execution of the process can be resumed where the interruption occurred by entering the command

```
:RESUME
```

The CAUSEBREAK intrinsic is not valid in a job.

## TERMINATING A PROCESS

You can programatically terminate a process with the TERMINATE intrinsic. The process and all of its descendants, including any extra data segments belonging to them, are deleted.

All files still open by the process are closed and assigned the same disposition they had when opened.

The form of the TERMINATE intrinsic call is

```
TERMINATE;
```

## ABORTING A PROCESS

If called from within any process in a user-process structure, the QUIT intrinsic aborts that process.

The QUIT intrinsic sets the job/session in an error state and transmits a Type 2 abort message to the calling process' list device. In a session, the process is aborted but the session remains active when the entire program finishes. In a batch job, the job terminates when the entire program finishes unless the :CONTINUE command (see the *MPE Commands Reference Manual*) has been included as part of the job.

Figure 4-7 shows the QUIT intrinsic being called if a READ statement did not execute properly. The abort message resulting from the QUIT intrinsic execution is shown below.

```
ABORT :SPROG.PUB.TECHPUBS.%0.%26: PROCESS QUIT P=1
```

The statement

```
IF < > THEN QUIT(1);
```

checks for a "not equal" condition code and, if CCG or CCL is returned, the QUIT intrinsic is called. The process is aborted and the abort message is printed. The QUIT parameter (1) is an arbitrary number supplied by the user and can be used to identify a specific QUIT intrinsic call in case of multiple possible QUIT intrinsic calls. This number, 1 in this case, is printed at the end of the abort message (P=1).

## ABORTING A PROGRAM

You can programmatically abort the entire user-process structure (program) with the QUITPROG intrinsic. This intrinsic destroys all processes up to, but not including, the job/session main process.

```

00001000 00000 0  SCONTROL USLIMIT
00002000 00000 0  BEGIN
00003000 00000 1  ARRAY HEADING(0:8):="INTEGER CALCULATOR";
00004000 00011 1  ARRAY ERRMSG(0:6):="ILLEGAL ENTRY.";
00005000 00007 1  ARRAY INPUT(0:36);
00006000 00007 1  BYTE ARRAY COMMAND(*)=INPUT;
00007000 00007 1  BYTE ARRAY ANSWER(0:13):="ACCUM =      ";
00008000 00010 1  ARRAY OUTPUT(*)=ANSWER;
00009000 00010 1  BYTE ARRAY TABLE(0:25):=
00010000 00001 1  5,3,"ADD",      5,3,"SUB",      5,3,"MUL",
00010100 00010 1  5,3,"DIV",      5,3,"SET",      0;
00011000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00012000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00013000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00014000 00016 1  LGTH,      INDX,      PARMCNT,      TYPE;
00015000 00016 1
00016000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00017000 00016 1
00019000 00016 1  <<END OF DECLARATIONS>>
00020000 00016 1
00021000 00016 1  PRINT(HEADING,9,0); <<PROGRAM ID>>
00022000 00004 1  WHO(INTERACTIVE); <<LIVE USER?>>
00023000 00010 1  LOOP:
00024000 00010 1  IF INTERACTIVE THEN PRINT(REQ,1,%320); <<PROMPT USER>>
00025000 00016 1  LGTH:=READ(INPUT,-72); <<GET COMMAND>>
00026000 00023 1  IF <> THEN QUIT(1); <<CHECK FOR ERROR>>
00027000 00026 1  IF COMMAND="END" THEN GO EXIT; <<DONE - EXIT>>
00028000 00040 1  COMMAND(LGTH):=%15; <<CARRIAGE RETURN>>
00028100 00043 1
00029000 00043 1  TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART COMMAND>>
00030000 00050 1  PARMINFO,TABLE);
00031000 00056 1  IF < THEN GO ERROR; <<NO COMMAND MATCH>>
00032000 00057 1  IF PARMCNT<>1 THEN GO ERROR; <<NO PARAMETERS>>
00033000 00062 1  INDX:=PARMINFO-@COMMAND; <<SUBSCRIPT OF PARM>>
00034000 00065 1  OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARAMETER>>
00035000 00070 1  PARMINFO(1).(0:8));
00036000 00075 1  IF <> THEN GO ERROR; <<CHECK FOR ERROR>>
00036100 00076 1
00037000 00076 1  CASE (TYPE-1) OF <<SELECT OPERATION>>
00038000 00100 1  BEGIN
00039000 00106 2  ACCUM:=ACCUM+OPERAND; <<ADD COMMAND>>
00040000 00116 2  ACCUM:=ACCUM-OPERAND; <<SUB COMMAND>>
00041000 00122 2  ACCUM:=ACCUM*OPERAND; <<MUL COMMAND>>
00042000 00126 2  ACCUM:=ACCUM/OPERAND; <<DIV COMMAND>>
00043000 00133 2  ACCUM:=OPERAND; <<SET COMMAND>>
00044000 00136 2  END;
00045000 00143 1  RESULT:
00046000 00143 1  MOVE ANSWER(8):="      "; <<RESET OLD ANSWER>>
00047000 00155 1  ASCII(ACCUM,10,ANSWER(8)); <<CONVERT ACCUM>>
00048000 00163 1  PRINT(OUTPUT,7,0); <<OUTPUT NEW ANSWER>>
00049000 00170 1  GO LOOP; <<CONTINUE CALCULATION>>
00050000 00171 1  ERROR:
00051000 00171 1  PRINT(ERRMSG,7,0); <<ERROR MESSAGE>>
00052000 00175 1  IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVE USER-QUIT>>
00053000 00201 1  GO LOOP; <<CONTINUE CALCULATION>>
00054000 00202 1  EXIT: END.
PRIMARY DB STORAGE=%020; SECONDARY DB STORAGE=%00113
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:11

```

Figure 4-7. Using the QUIT Intrinsic

In batch jobs not containing the `:CONTINUE` command (see the *MPE Commands Reference Manual*), this terminates the job.

The form of the `QUITPROG` intrinsic call could be as follows:

```
QUITPROG(1);
```

The parameter (1) can be any user-specified number. When the `QUITPROG` intrinsic executes, this number is printed as part of the abort message.

## CHANGING STACK SIZES

When you prepare or execute a process, you specify (or allow MPE to assign by default) the size of the stack (Z to DB) area and the user-managed (DL to DB) area within the stack segment. Once the process begins execution, you can programmatically change the size of these areas, to meet new requirements as they arise, by altering the register offsets Z to DB or DL to DB. For example, you typically expand the size of these areas when you find, during process execution, that the sizes specified initially were not sufficient for your data requirements. Conversely, you might contract the size of either of these areas should your process no longer require large amounts of space for data. (This is a good practice — it improves overall system performance.) These changes are requested with the `DLSIZE` intrinsic for the DL to DB area and the `ZSIZE` intrinsic for the Z to DB area.

If you plan to expand or contract the Z to DB or DL to DB areas programmatically, you *must* specify, at the time the stack is created, the anticipated maximum size of the stack segment. This value is used by MPE in allocating disc storage. The maximum stack size value is specified at preparation or run time with the `segsize` parameter of the `:PREP`, `:PREPRUN`, or `:RUN` command, or if you are a user with the Process-Handling Capability, after the program is running with the `maxdata` parameter of the `CREATE` intrinsic.

### NOTE

When the stack segment belonging to a process running in privileged mode is frozen in main memory (during an input/output operation, for example), either implicitly (when a user's process interfaces directly with the input/output system), or explicitly (by a direct call to system intrinsics), the intrinsics to change the register offsets DL to DB or Z to DB cannot be executed. When these intrinsics are called under such circumstances, a special `FROZEN STACK` error code is returned to the calling process, which then may attempt recovery. In general, this error implies that you should wait until the stack is unfrozen before re-issuing the intrinsic call.

## CHANGING THE DL TO DB AREA SIZE

You can expand or contract the area between DL and DB within the stack segment with the `DLSIZE` intrinsic. All current information within the DL to DB area is saved on expansion. On contraction, data within the area to be contracted is lost. See figure 4-8.

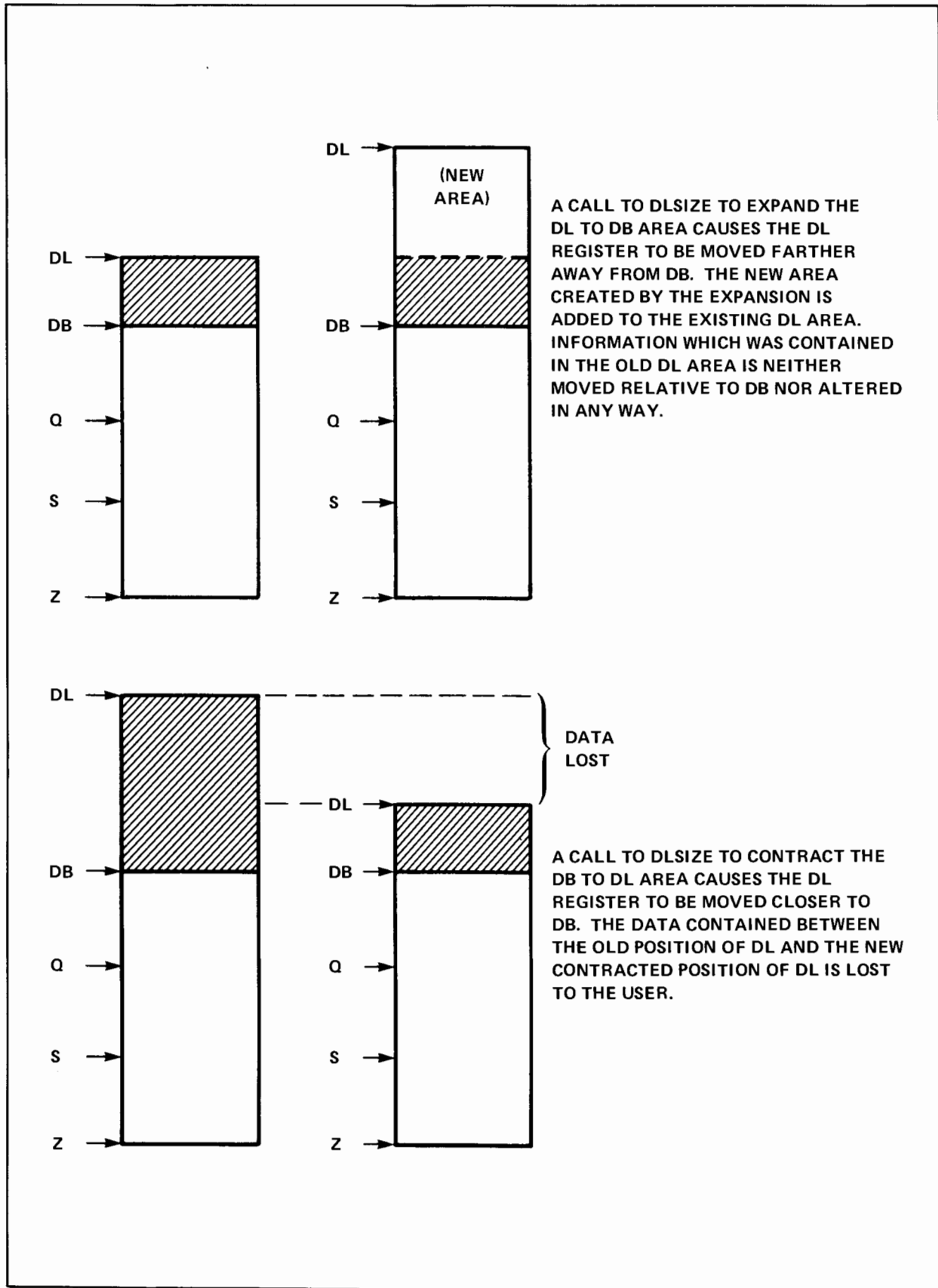


Figure 4-8. Expanding and Contracting the DL to DB Area



A request for contraction less than the initial DL size of the area causes the initial DL size to be granted and an error condition code (CCL) to be returned. If the size requested causes the stack to exceed the maximum size permitted by the stack area, Z to DL, only this maximum will be granted.

Some possible applications for the DL area are:

- Dynamically allocated I/O buffers when using the FCOPY subsystem.
- Compiler symbol tables when programming in SPL.
- Global storage area for library routines in Segmented Libraries. These routines typically have no global area storage which will retain values assigned to them between calls to the procedure. These routines also typically have no common storage where data can be shared by several procedures. If you define your conventions carefully, these library routines can use the DL area of the process which calls them for this kind of storage. Care must be taken, however, because the first 10 words of the DL area are reserved for subsystem use, and some system routines make use of the DL area for their own storage. As long as your environment is completely known and well defined, your main program or your library routines can get DL space and manage it as they choose..

Figure 4-9 contains an SPL program that expands and contracts the DL to DB area.

#### NOTE

All addressing within the DL to DB area is DB-relative *negative* indexing and SPL is the only language, at present, which can access this area for you.

The program in figure 4-9 reads data from \$STDIN and stores it in the DL area at progressively lower (DB - *n*) addresses. Additional DL space is allocated when the next buffer would lie outside the current DL area. When a null record (0 length) is read, the program outputs the data on a last-in-first-out basis. After all the records are output, the DL space is collapsed to its initial allocation and the operation begins again. The loop is terminated by entering :EOD in place of a data line.

#### NOTE

The program was PREPped with a MAXDATA = 2000 parameter.

The statement

```
TOTALDL:=DLSIZE(0);
```

sets the DL to DB area to the original value assigned when the process was created (initial DL).

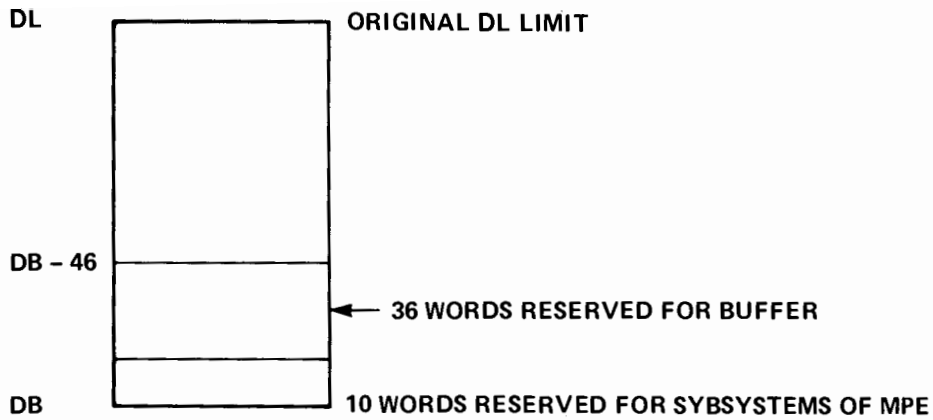
Observe the following illustration of the DL to DB area.

```

00001000 00000 0  SCONTROL USLIMIT
00002000 00000 0  BEGIN
00003000 00000 1  INTEGER IN,OUT,LGTH,
00004000 00000 1  TOTALDL:=0;
00007000 00000 1  LOGICAL PROMPT:="? ";
00008000 00000 1  LOGICAL POINTER BUFFER:=-46; <<BUFFER:36 ; RESERVED:10>>
00009000 00000 1
00010000 00000 1  INTRINSIC FOPEN,DLSIZE,FREAD,FWRITE,QUIT;
00011000 00000 1
00012000 00000 1  <<END OF DECLARATIONS>>
00013000 00000 1
00014000 00000 1  IN:=FOPEN(%44); <<SSDTIN>>
00015000 00007 1  IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00016000 00012 1
00017000 00012 1  OUT:=FOPEN(%414,1); <<SSDLIST>>
00018000 00022 1  IF < THEN QUIT(2); <<CHECK FOR ERROR>>
00018100 00025 1  RESTART;
00018200 00025 1  TOTALDL:=DLSIZE(0); <<SET DL TO INITIAL SIZE>>
00018300 00030 1  IF <> THEN QUIT(3); <<CHECK FOR ERROR>>
00019000 00033 1  LINEIN:
00020000 00033 1  PUSH(DL); <<GET CURRENT DL SETTING>>
00021000 00034 1  IF IOS>@BUFFER THEN <<NEXT BUFFER OUTSIDE DL?>>
00022000 00036 1  BEGIN
00023000 00036 2  TOTALDL:=DLSIZE(TOTALDL-128); <<GET MORE DL AREA>>
00024000 00043 2  IF <> THEN QUIT(4); <<CHECK FOR ERROR>>
00025000 00046 2  END;
00027000 00046 1  BUFFER:=" ";
00028000 00050 1  MOVE BUFFER(1):=BUFFER,(35); <<BLANK READ BUFFER>>
00029000 00055 1  FWRITE(OUT,PROMPT,1,%320); <<? PROMPT FOR INPUT>>
00030000 00062 1  IF <> THEN QUIT(5); <<CHECK FOR ERROR>>
00031000 00065 1  LGTH:=FREAD(IN,BUFFER,36); <<INPUT DATA>>
00032000 00073 1  IF < THEN QUIT(6); <<CHECK FOR ERROR>>
00033000 00076 1  IF > THEN GO EXIT; <<CHECK FOR :EOD>>
00034000 00077 1  IF LGTH=0 THEN <<NO DATA INPUT?>>
00035000 00102 1  BEGIN
00036000 00102 2  @BUFFER:=@BUFFER+36; <<ADDRESS PREVIOUS BUFFER>>
00037000 00105 2  GO LINEOUT; <<START OUTPUT PHASE>>
00038000 00113 2  END;
00039000 00113 1  @BUFFER:=@BUFFER-36; <<ADDRESS NEXT BUFFER>>
00040000 00116 1  GO LINEIN; <<CONTINUE >>
00041000 00117 1  LINEOUT:
00042000 00117 1  FWRITE(OUT,BUFFER,36,0); <<OUTPUT DATA>>
00043000 00124 1  IF <> THEN QUIT(7); <<CHECK FOR ERROR>>
00044000 00127 1  IF @BUFFER>=-46 THEN GO RESTART; <<ALL BUFFERS OUTPUT;RESTART>>
00050000 00132 1  @BUFFER:=@BUFFER+36; <<ADDRESS PREVIOUS BUFFER>>
00051000 00135 1  GO LINEOUT; <<CONTINUE OUTPUT PHASE>>
00053000 00137 1  EXIT;END.
PRIMARY DB STORAGE=%006; SECONDARY DB STORAGE=%00000
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:22

```

Figure 4-9. Using the DLSIZE Intrinsic



Statement number 8 in the program

```
LOGICAL POINTER BUFFER:=-46;
```

sets a pointer to **DB - 46**, which is the DB-relative address of the first word in **BUFFER**.

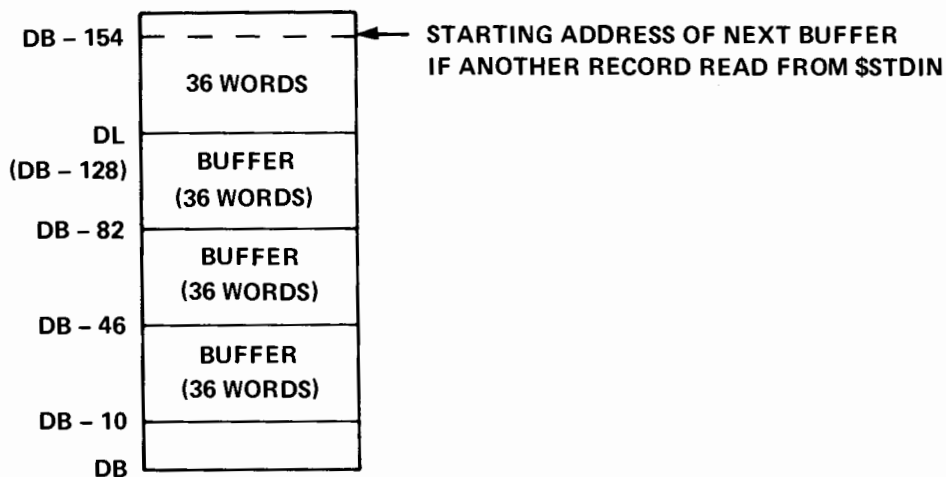
The statement

```
PUSH(DL);
```

pushes the **DL** register contents onto the top of the stack and the statement

```
IF TOS > @BUFFER THEN
```

checks if the address of the first word of **BUFFER** is outside the **DL** to **DB** area. In other words, **TOS** contains the **DL** address from the **DL** register. If this value is greater than (less negative) than the address of the first word in **BUFFER**, then **BUFFER** is outside the **DL** to **DB** area. See below.



If the DL address is DB - 128, then when only one buffer is filled, the address of the next buffer is well within the DL to DB area. When three buffers have been filled, however, the starting address of the next buffer (DB - 154) would be outside the DL to DB area (DB - 0 to DB - 128). The TOS (DB - 128) is greater than the address of the first word in the next buffer (DB - 154).

If the next buffer would lie outside the DL to DB area, the next four statements in the program

```
BEGIN
    TOTALDL:=DLSIZE(TOTALDL-128);
    IF < > THEN QUIT(4);
END;
```



add 128 more words to the DL to DB area.

The statements

```
BUFFER:=" ";
MOVE BUFFER(1):=BUFFER, (35);
```

fill BUFFER with blanks preparatory to reading the input from \$STDIN. A prompt is displayed and the user enters the next record. The length of the record is assigned to LGTH.

If LGTH = 0, signalling a carriage return (no data entered), the program addresses the previous buffer and transfers control to LINEOUT. The contents of the buffers are written on \$STDLIST on a last-in-first-out basis. When the original address is reached (DB - 46), control is returned to RESTART and the procedure is repeated.

The statement

```
TOTALDL:=DLSIZE(0);
```

contracts the DL to DB area back to its original size, destroying the contents of all buffers. An :EOD entry terminates program execution.

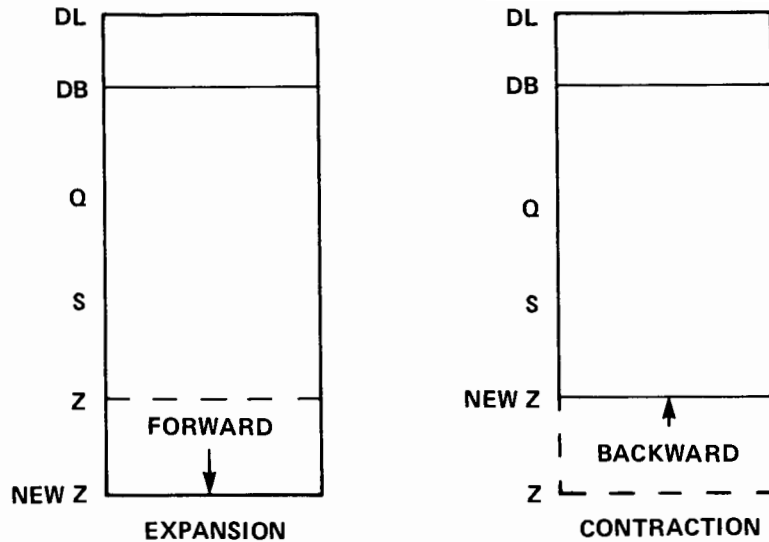
Figure 4-10 shows the results when the program is run.

## CHANGING THE Z TO DB AREA SIZE

You can alter the size of the current Z to DB area by adjusting the register offset of the Z address from the DB address with the ZSIZE intrinsic.

The ZSIZE intrinsic moves the Z address forward (expansion) or backward (contraction) as shown below.





If the Z to DB area size requested exceeds the maximum size permitted for the Z to DL (stack) area, only the maximum size allowed is granted.

All changes to the Z to DB area are made in increments or decrements of 128 words, thus the size actually granted may differ from the size requested. For example, if the present Z to DB area size is 128 words, a request for a size of 129 words would result in a size of 256 words being granted.

A ZSIZE intrinsic call could be

```
ACTSIZE:=ZSIZE(250);
```

If the maximum size for the Z to DL area permitted, an actual size granted for the Z to DB area of 256 words would be returned to ACTSIZE.

## ENABLING AND DISABLING TRAPS

Normally, whenever a major error occurs during the execution of a hardware instruction, a procedure from the System Library, or an intrinsic called by a user, the user program is aborted and an error message is output. You can, however, avoid immediate abort by enabling any of three software traps provided by MPE:

The *arithmetic trap*, for hardware instruction errors.

The *library trap*, for errors detected during execution of a system library procedure.

The *system trap*, for errors detected during execution of a system intrinsic.

When an error occurs, the corresponding trap, if enabled, suppresses output of the normal error message, transfers control to a *trap procedure* defined by you, and passes one or more parameters describing the error to this procedure. This procedure may attempt to analyze or recover from the error, or may execute some other programming path. Upon exiting from the trap procedure, control returns to the instruction following the one that activated the trap. In the case of the library trap, however, you can specify that the process be aborted when control exits from the trap procedure. Trap intrinsics can be invoked from within trap procedures.

## NOTE

The validity of a trap procedure, specified by the external-type label of the user trap procedure (*plabel*), depends on the code domain of the caller's code and executing mode (privileged or non-privileged), and on the code domain of the *plabel* and the mode (privileged or non-privileged). The code domains are:

PROG	(User Program)
GSL	(Group SL)
PSL	(Public SL)
SSL	(System SL, non-MPE Segments)
MPSSL	(System SL, MPE Segments)

If, at the time of enabling a trap procedure, the code of the caller is

1. Non-privileged in PROG, GSL, or PSL: *plabel* must be non-privileged in PROG, GSL, or PSL.
2. Privileged in PROG, GSL, or PSL *plabel* may be privileged or non-privileged in PROG, GSL, or PSL
3. Privileged or non-privileged in SSL: *plabel* may be in any non-MPSSL segment.

## ARITHMETIC TRAPS

There are two levels of arithmetic traps: the *hardware arithmetic trap set* and the *software arithmetic trap*. Each trap in the hardware trap set detects a particular type of hardware error, such as division by zero or result overflow. The software trap, if enabled, receives an internal interrupt signal from a hardware trap when an error is encountered, and transfers control to a user trap procedure.

When a user process begins execution, all hardware trap set interrupt signals are enabled automatically, but the software trap is disabled, permitting any hardware error to abort the process. Through intrinsic calls, however, you can alter the ability of the hardware trap set to send signals, and that of the software trap to receive a signal from any particular hardware trap. Only signals received and accepted by the software trap can invoke a user trap procedure.

To enable or disable the internal interrupt signals from *all* hardware arithmetic traps, you enter the ARITRAP intrinsic call, as follows:

```
ARITRAP(STATE);
```

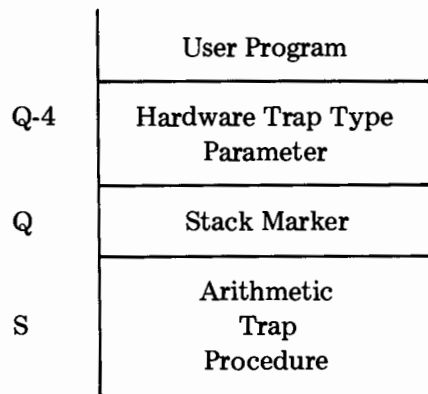
where STATE is TRUE (bit 15 = 1) to enable the signals from all hardware traps, and FALSE (bit 15 = 0) to disable these signals.

When a software arithmetic trap procedure is executed, the Index register contains the word of code being executed when the trap occurred. This information, plus, if necessary, the right stackop bit in the stacked status word, can be used to identify the offending instruction. A one-word parameter is available, in Q - 4, in which certain bits indicate the type of hardware trap invoked. The various traps leave the parameter in Q - 4 as follows.

## STANDARD TRAPS

- Bit 15 = Floating Point Divide by 0
- 14 = Integer Divide by 0
- 13 = Floating Point Underflow
- 12 = Floating Point Overflow
- 11 = Integer Overflow

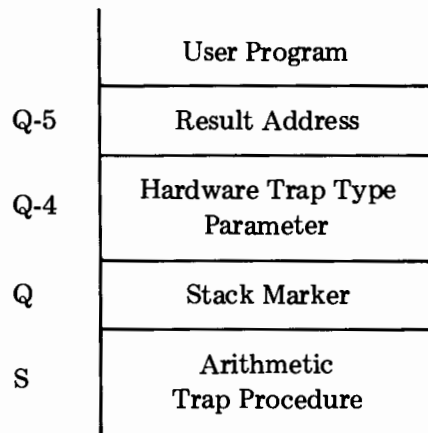
A return from the trap procedure (through an (EXIT1) instruction) will resume execution in the user code domain at the instruction following that which activated the trap procedure. The condition of the stack when the trap procedure is invoked is



## EXTENDED PRECISION FLOATING-POINT TRAPS

- Bit 10 = Extended Precision Overflow
- 9 = Extended Precision Underflow
- 8 = Extended Precision Divide by 0

The address of the result operand is left on the stack in Q-5. An (EXIT 2) return will resume execution in the user code domain at the instruction following the one which caused the trap. The condition of the stack when the trap procedure is invoked is

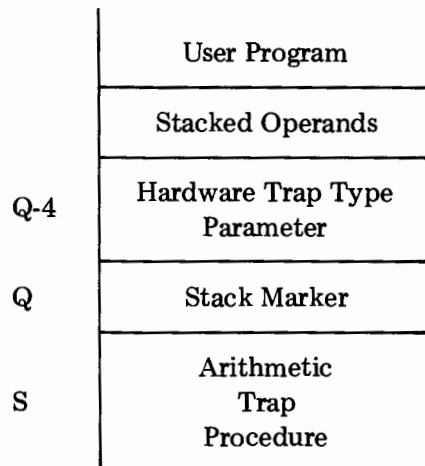




## COMMERCIAL INSTRUCTION TRAPS

- Bit 7 = Decimal Overflow
- 6 = Invalid ASCII Digit (CVAD)
- 5 = Invalid Decimal Digit
- 4 = Invalid Source Word Count (CVBD)
- 3 = Invalid Decimal Operand Length
- 2 = Decimal Divide by 0

The parameters stacked for the execution of the instruction are left on the stack below Q-4. To return properly the trap handler must examine the opcode (found in the Index Register) to determine the proper stack decrement to use on exit. The condition of the stack when the trap procedure is invoked is



An arithmetic trap procedure is shown in figure 4-11. The procedure `FDIVZRO` is a trap procedure to which control is passed if a floating point divide by zero software trap is enabled *and* a program attempts an operation to divide by 0.

The statement

```
XZRITRAP(%1,@FDIVZRO,DUMMY1,DUMMY2);
```

enables the floating point divide by 0 trap. The parameter `%1` (bit 15 = 1) enables only the floating point divide by 0; the `@FDIVZRO` passes the trap procedure as a parameter; `DUMMY1` and `DUMMY2` are dummy parameters.

The statement

```
RESULT:=NUM/DENOM;
```

attempts a floating point divide by 0 operation and, since the floating point divide by 0 trap is enabled, control is transferred to procedure `FDIVZRO`. The condition of the stack at this point is

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 REAL NUM:=1.,
00004000 00000 1 DENOM:=0.,
00005000 00000 1 RESULT;
00006000 00000 1 INTEGER DUMMY1,DUMMY2;
00007000 00000 1 ARRAY DIVMSG(0:7):="DIVIDE OPERATION";
00008000 00010 1 ARRAY ADDMSG(0:6):="ADD OPERATION ";
00009000 00007 1
00010000 00007 1 PROCEDURE FDIVZRO(QUOTIENT,TRAP);
00011000 00000 1 VALUE QUOTIENT,TRAP;
00012000 00000 1 REAL QUOTIENT;
00013000 00000 1 LOGICAL TRAP;
00014000 00000 1 BEGIN
00015000 00000 2 IF QUOTIENT=0. THEN GO EXIT; <<LEAVE UNCHANGED>>
00016000 00004 2 IF QUOTIENT<0. <<CHECK SIGN OF ANSWER>>
00017000 00006 2 THEN QUOTIENT:=%377777777777D << - LARGEST VALUE>>
00018000 00011 2 ELSE QUOTIENT:=%177777777777D; << + LARGEST VALUE>>
00019000 00023 2 EXIT:
00020000 00023 2 RETURN 1; <<DELETE TRAP PARM ONLY>>
00021000 00026 2 END;
00022000 00000 1
00023000 00000 1 INTRINSIC XARITRAP,PRINT,QUIT;
00024000 00000 1
00025000 00000 1 <<END OF DECLARATIONS>>
00026000 00000 1
00027000 00000 1 XARITRAP(%1,FDIVZRO,DUMMY1,DUMMY2); <<ARM FP/O. TRAP>>
00027100 00005 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00028000 00010 1
00029000 00010 1 PRINT(DIVMSG,8,0); <<DIVIDE HEADING>>
00030000 00014 1 RESULT:=NUM/DENOM; <<DIVIDE>>
00031000 00020 1
00032000 00020 1 PRINT(ADDMSG,7,0); <<ADD HEADING>>
00033000 00024 1 RESULT:=RESULT+RESULT; <<ADD>>
00034000 00030 1 END.
PRIMARY DB STORAGE=%012; SECONDARY DB STORAGE=%00017
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:02; ELAPSED TIME=0:00:11

```

Figure 4-11. Using the XARITRAP Intrinsic

	User Program
Q-5	RESULT
Q-4	Hardware Trap Type Parameter (%1)
Q	Stack Marker
S	Arithmetic Trap Procedure

The address of RESULT has been left at Q - 5 and the FDIVZRO procedure uses this for QUOTIENT.

If QUOTIENT = 0 (0 divided by 0), no action is taken and the procedure is exited, transferring control back to the user program.

If QUOTIENT is less than 0, then the largest possible negative value is assigned to QUOTIENT.

If QUOTIENT is not less than 0, the largest possible positive value is assigned.

The statement

```
RETURN 1;
```

deletes only one word from the stack (the TRAP parameter at Q - 4) and returns to the program leaving the address of QUOTIENT (whose value is either %3777777777D or %1777777777D) at stack location Q - 5.

When the statement

```
RESULT:=RESULT+RESULT;
```

tries to add the large value contained in RESULT to itself, the floating point overflow hardware trap aborts the process. The floating point overflow error was deliberately caused in this example program by assigning one of two largest possible values to RESULT and then attempting an add (RESULT + RESULT) which could not succeed. In a practical program, of course, such trap recovery (causing another intentional error) would not be used. The result of running the example program is shown below.

```
:RUN ATRAP
```

```
DIVIDE OPERATION  
ADD OPERATION
```

```
ABORT :ATRAP.PUB.SUPPORT.%Ø.%26: FL.PT. OVERFLOW
```

```
ERR 2  
ABNORMAL PROGRAM TERMINATION  
:
```

## LIBRARY TRAP

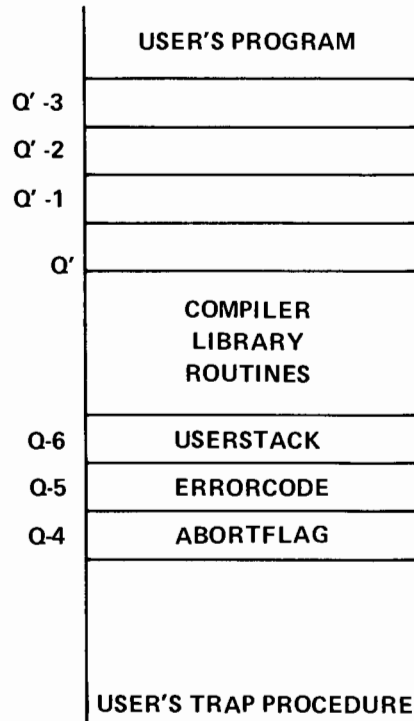
The software library trap reacts to major errors that occur during execution of procedures from the System Library. When a user program begins execution, this trap is disabled automatically. You can enable (or disable) it with the XLIBTRAP intrinsic. When enabled, the library trap passes control to a trap procedure in the event of an error. This procedure, in turn, returns to the user program four words containing the stack marker created when the library procedure was called by the user program. In addition, the trap procedure returns an integer representing the error number. When the procedure is completed, it either transfers control to the instruction following that which caused the error or aborts the process at your option. The trap procedure is defined by you, but it must conform to the special format discussed in the *HP 3000 Compiler Library Manual*.

The XLIBTRAP intrinsic call could be as follows:

```
XLIBTRAP(PLABEL,OLDPLABEL);
```

where PLABEL is the external-type label of your trap procedure. If the value of this parameter is 0, the trap is disabled. OLDLABEL is a word in which the previous *plabel* is returned to the use program. If no *plabel* existed previously, 0 is returned.

When a library trap procedure is invoked, the condition of the stack is:



**USERSTACK**

A word pointer to the base of the stack marker placed on the stack when the user program called the compiler library.

**ERRORCODE**

A number indicating the type of compiler library error, described in the *HP 3000 Compiler Library Manual*.

**ABORTFLAG**

A value set before the user exists from the trap procedure. If TRUE, the compiler library aborts the program with the standard error message (just as if no trap procedure had been executed). If FALSE, the compiler library does not abort the program and no error message is printed; in this case, the compiler library attempts error-recovery.

**SYSTEM TRAP**

The software system trap reacts to errors occurring in intrinsics called by user programs. Typical errors are

- Illegal access. An attempt by a user to access an intrinsic for which he does not have access capability.
- Illegal parameters. The passing to an intrinsic of parameters that are not defined for the user's environment.
- Illegal environment. The DB register is not currently pointing to the user's stack area.
- Resource violation. The resource requested by a user is either illegal or outside the constraints imposed by MPE.

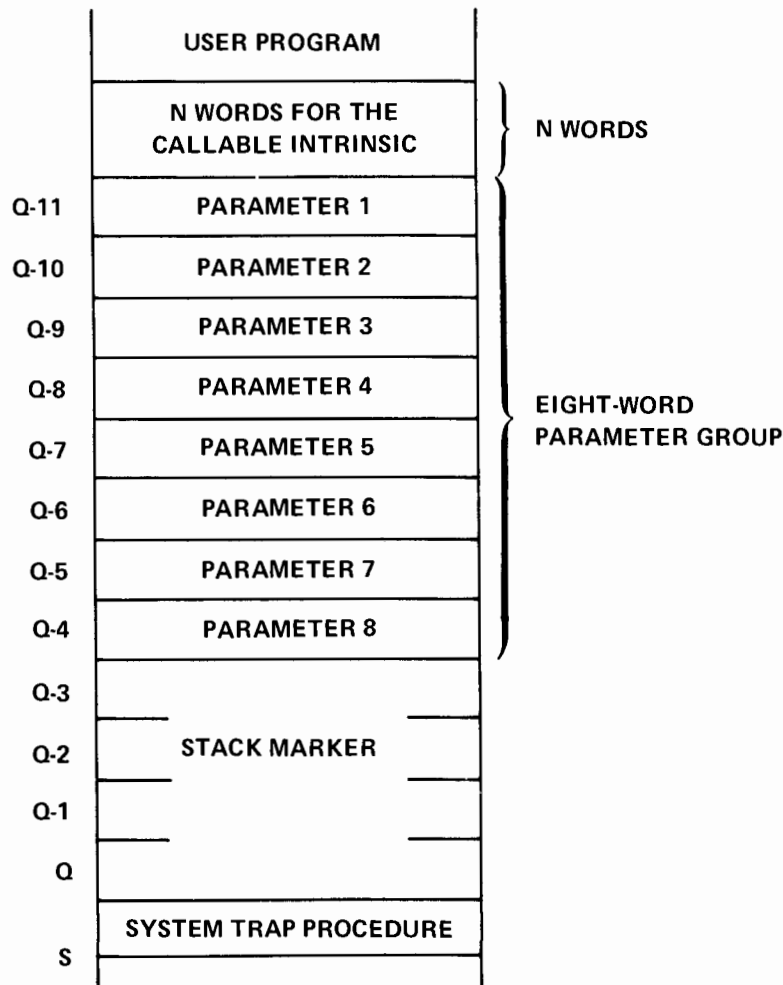
When a user program begins execution, the system trap is disabled automatically. When enabled by the XSYSTRAP intrinsic call and subsequently activated by an error, the trap transfers control to a trap procedure.

The system trap is enabled or disabled by a XSYSTRAP intrinsic call, as follows:

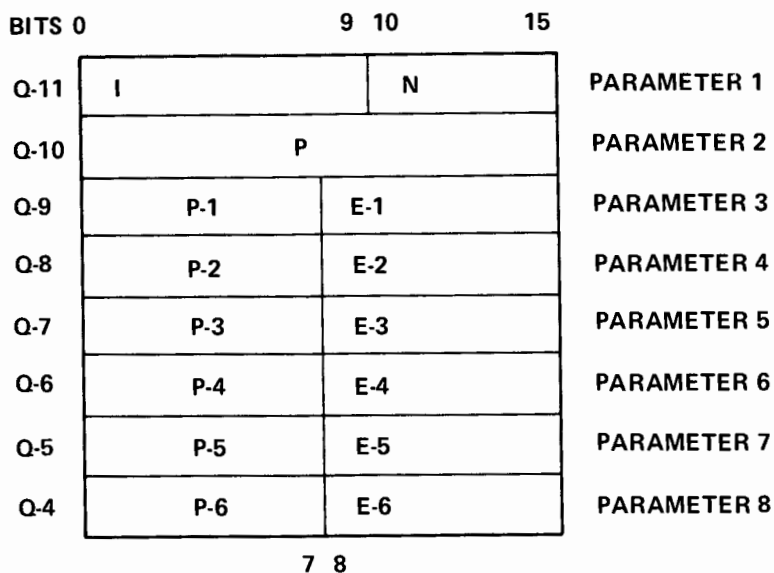
```
XSYSTRAP(PLABEL,OLDPLABEL);
```

where PLABEL is the external-type label of your trap procedure. If the value of this parameter is 0, the software trap is disabled. OLDPLABEL is a word to which the previous *plabel* is returned to your program. If no *plabel* existed previously, 0 is returned.

When a system trap procedure is executed because of an abort condition arising in a system intrinsic, the stack is readjusted to provide an eight-word parameter group between the intrinsic parameters and the stack marker.



The format of the eight-word parameter group in Q-4 through Q-11 is



- I                    Intrinsic number.
- N                    Number of callable intrinsic parameters. (To resume execution in the user code domain, an EXIT N+8 instruction should be executed.)
- P                    Additional parameter information.
- P-1 through P-6    Parameters modifying the error bytes, described below. If no modifying parameter is present, the corresponding parameter byte is set to zero.
- E-1 through E-6    Error bytes, containing the error codes noted in Section X. The last error code present is delimited by the value of zero in the following error byte.

With these parameters, the trap procedure may take any recovery action necessary — write messages, produce selective dumps, set error-indication flags, or allow interactive debugging. Finally, the procedure may either call the TERMINATE intrinsic or issue an (EXIT N+8) instruction to return to the user program (at the location following that where the trap was invoked), with appropriate error indications.

A sample declaration for a system trap procedure, and an example of how you might issue an EXIT N+8 instruction follow:

```
PROCEDURE        SYSTEMTRAP    (PARAMETER1,PARAMETER2,PARAMETER3,
                                 PARAMETER4,PARAMETER5,PARAMETER6,
                                 PARAMETER7,PARAMETER8);
```

```

VALUE      PARAMETER1,PARAMETER2,PARAMETER3,PARAMETER4,
           PARAMETER5,PARAMETER6,PARAMETER7,PARAMETER8;

LOGICAL    PARAMETER1,PARAMETER2,PARAMETER3,PARAMETER4,
           PARAMETER5,PARAMETER6,PARAMETER7,PARAMETER8;

BEGIN

    INTEGER N;
    .
    .
    .
    << USER MAY OUTPUT MESSAGES >>
    .
    .
    .
    N:=PARAMETER1  LAND%37; << N=NUMBER OF PARAMETERS
                           PASSED TO CALLABLE
                           INTRINSIC >>

    TOS:=N+%31410;      << PUT "EXIT N+8" ON TOP
                           OF STACK >>

    ASSEMBLE (XEQ 0);   << EXECUTE "EXIT N+8" ON
                           TOP OF STACK >>

END;

```

## CONTROL-Y TRAPS

If you are running a program in an interactive session, you can enable a special trap that transfers control from the currently-executing program to a trap procedure whenever a CONTROL-Y subsystem break signal is entered from the terminal. On most terminals, the CONTROL-Y signal is transmitted by pressing the Y key while holding the CONTROL key down.

When more than one process is currently running within your process' tree structure, the CONTROL-Y signal interrupts the last process to enable the trap.

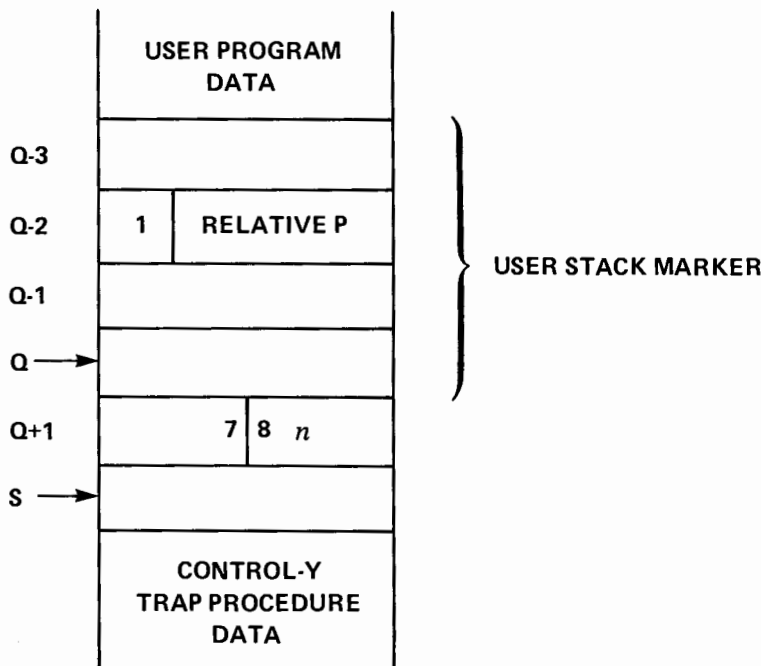
When a process is interrupted by a CONTROL-Y signal, the following occurs:

1. The input/output transactions pending between the process and the terminal are halted and flagged as though all were completed successfully.
2. Control is transferred to the trap procedure defined by you, with which you can now interact. The trap procedure executes in the same mode (privileged or non-privileged) as the user program that was interrupted.
3. Control returns from the trap procedure to the interrupted program or procedure. If the interrupted program or procedure was awaiting completion of input/output (reading from or writing to the terminal) when the CONTROL-Y signal was received, the FREAD or

FWRITE intrinsic that was being executed is flagged as successfully completed when control returns from the trap procedure. If the CONTROL-Y signal was received during reading, the number of characters typed in *before* this signal is returned to you as the value of FREAD. The carriage position is unchanged.

If you send another CONTROL-Y signal, it is ignored unless a call to the RESETCONTROL intrinsic was issued at some point prior to the signal.

If you send a CONTROL-Y signal while MPE system code is executing on your behalf, no interrupt occurs until the process resumes execution of your code; then control transfers to the trap procedure. This protects MPE operation. An EXIT N instruction later returns control to the instruction in the user code domain following the last instruction executed before the CONTROL-Y trap procedure was invoked. When the trap procedure is invoked, the condition of the stack is as follows:



When the first instruction in the trap procedure is executed, the Q register points to the user stack marker and the S register points to Q + 2. The trap procedure should not write data in the rightmost byte of the word Q + 1, because it contains the number of words (which were passed as parameters) in the stack, plus the stack marker, which are to be deleted from the stack when the EXIT instruction executes. The operating system searches back to the last user stack marker and sets a bit to 1 (bit 0 of Relative P) in that marker. An EXIT instruction through this marker causes a trap. The trap is recognized, a marker is built, and an exit is executed to the trap handler. On return, the trap handler must know the value contained in Q + 1, and this value is passed as the N parameter of the EXIT N instruction. The EXIT N instruction must be placed on the stack as follows:

TOS:=%31400 + N

The EXIT N instruction then is executed by an XEQ instruction.



## NOTE

If you are a user with the Privileged Mode Capability, you should be aware of the following:

1. If your interrupted code was executing in privileged mode, your trap procedure also must be executed in privileged mode, and therefore must have privileged mode capability.
2. When your process is executing in privileged mode, and a CONTROL-Y signal invokes a trap procedure, the trap procedure is entered with the same DB register setting in effect when the signal was received. Thus, if the DB register is pointing to an extra data segment rather than the user stack when a CONTROL-Y signal is received, it will continue to point to that extra data segment when the trap procedure is entered.

Figure 4-12 shows a program containing a CONTROL-Y trap procedure. The statements

```
LOOP:
    CNTR:=CNTR+1D;
    IF CNTR < 3000000D THEN GO LOOP;
```

increment a double-word counter by 1D each time the loop is executed. When the counter reaches the value 3000000D, program execution terminates.

The CONTROL-Y trap procedure, beginning with the statement

```
PROCEDURE CONTROLY;
```

assumes control whenever CONTROL-Y is entered from the terminal. The trap procedure executes, then control passes back to the loop.

The statement

```
INTEGER SDEC = Q + 1;
```

equivalences SDEC to Q + 1. The rightmost byte of Q + 1 contains the number of words on the stack to be deleted when the exit instruction executes. This value is passed to EXIT as the N parameter.

The counter value is converted to an ASCII string by calling the DASCII intrinsic and the PRINT intrinsic call displays this ASCII string on the terminal.

The RESETCONTROL intrinsic call enables the CONTROL-Y trap. To take effect, this intrinsic must be called from within the trap procedure. An EXIT instruction must be built and the statement

```
TOS:=%31400 + SDEC;
```

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 ARRAY HEADING(0:10):="CONTROL Y TRAP EXAMPLE";
00004000 00013 1 ARRAY MSG(0:15):="COUNTER CURRENTLY = ";
00005000 00020 1 BYTE ARRAY RMSG(*)=MSG;
00006000 00020 1 DOUBLE CNTR:=0D;
00007000 00020 1 INTEGER DUMMY, LGTH;
00008000 00020 1
00009000 00020 1 INTRINSIC PRINT, XCONTRAP, QUIT, DASCII, RESETCONTROL;
00010000 00020 1
00011000 00020 1
00012000 00000 1 PROCEDURE CONTROLY;
00013000 00000 2 BEGIN
00014000 00000 2 INTEGER SDEC=Q+1;
00015000 00000 2 LGTH:=DASCII(CNTR,10,RMSG(20)); <<CONVERT COUNTER>>
00016000 00007 2 PRINT(MSG,16,0); <<OUTPUT COUNTER>>
00017000 00013 2 RESETCONTROL; <<PEARM CONTROL Y TRAP>>
00018000 00014 2 TQS:=%31400+SDEC; <<BUILD EXIT INSTRUCTION>>
00019000 00016 2 ASSEMBLE(XEQ 0); <<EXECUTE EXIT>>
00020000 00017 2 END;
00021000 00000 1
00022000 00000 1 <<END OF DECLARATIONS>>
00023000 00000 1
00024000 00000 1 PRINT(HEADING,11,0); <<PROGRAM ID>>
00025000 00004 1 XCONTRAP(@CONTROLY,DUMMY); <<ARM CONTROL Y TRAP>>
00026000 00007 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00027000 00012 1 LOOP:
00028000 00012 1 CNTR:=CNTR+1D; <<DOUBLE INCREMENT>>
00029000 00023 1 IF CNTR<3000000D THEN GO LOOP; <<CONTINUOUS LOOP>>
00030000 00027 1 END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00033
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:02; ELAPSED TIME=0:00:26

```

Figure 4-12. Using the XCONTRAP Intrinsic

accomplishes this, loading the octal value %31400 plus the value of SDCE (Q + 1) onto the top of the stack. The statement

```
ASSEMBLE(XEQ 0);
```

executes the statement on the top of the stack, which in this case is the EXIT instruction placed there by the previous statement.

The CONTROL-Y trap is enabled by the statement

```
XCONTRAP(@CONTROLY,DUMMY);
```

The @CONTROLY parameter informs the system that a procedure (CONTROLY) is being passed as a parameter.

The results of executing the program are shown below.


```

: RUN CONTY

CCNTROL Y TRAP EXAMPLE
COUNTER CURRENTLY = 125153
COUNTER CURRENTLY = 1093423
COUNTER CURRENTLY = 1860957
COUNTER CURRENTLY = 2700949

END OF PROGRAM
:

```



## TIME AND DATE INTRINSICS

You can programmatically request the return of system timer information with the `TIMER` intrinsic; the time of day with the `CLOCK` intrinsic; the calendar date with the `CALENDAR` intrinsic; and the duration, in milliseconds, that a process has been running with the `PROCTIME` intrinsic.

### OBTAINING SYSTEM TIMER INFORMATION

A 31-bit logical quantity representing the current system timer count can be returned to your program with the `TIMER` intrinsic. This information can be used in routines that generate random numbers, or in measuring the real time elapsed between two events. The resolution of the system timer is one millisecond, thus readings taken within a one-millisecond period may be identical.

Figure 4-13 contains a program that uses the system timer bit count to generate a random octal number. This number then is converted to one of the ASCII characters ; , < , = , > , ? , @ , or A through Z.

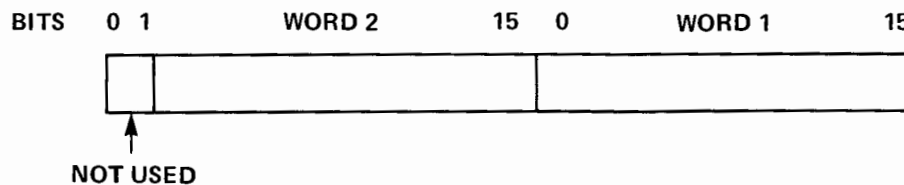
The statement

```

CBUF(5):=INTEGER(TIMER).(11:5)+%73;

```

calls the `TIMER` intrinsic to obtain the timer count. A double-word quantity is returned as follows:



The `INTEGER` function strips word 2 from this quantity, leaving a 16-bit integer value. The low-order five bits of course change value most rapidly, and these bits are used to obtain a decimal number from 0 to 32.

The octal codes for ASCII characters ; , < , = , > , ? , @ , and A through Z range from 000073 to 000132, or decimal values 58 through 90 (a difference of 32 decimal). Thus, by adding `%73` to the value obtained from the low-order five bits of the system timer information, one of the above ASCII characters is generated by the foregoing statement and assigned to the 6th (`CBUF(5)`) position of byte array `CBUF`. The `FWRITE` displays this character, and the string "TYPE" on the terminal (`CBUF` and `BUFR` have been equivalenced, see statements 6 and 7).

```

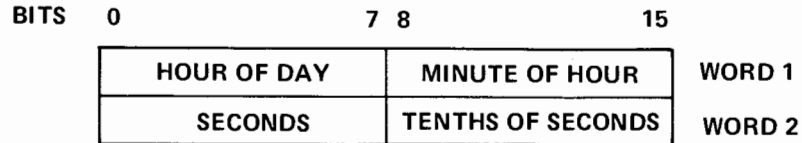
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INNAME(0:5):="INPUT ";
00004000 00004 1 BYTE ARRAY OUTNAME(0:6):="OUTPJT ";
00005000 00005 1 INTEGER IN,OUT,LGTH,DUMMY,TIME,TIMEOUT:=10;
00006000 00005 1 ARRAY BUFR(0:3):="TYPE X",0;
00007000 00004 1 BYTE ARRAY CBUF(*)=BUFR;
00008000 00004 1 ARRAY INSTRUCTIONS(0:34):="REACTION TIMER: ",%6412,
00009000 00011 1 "TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN. ";
00010000 00043 1 ARRAY MSG(0:24):="TRY AGAIN? (Y/N)","WRONG CHARACTER.",
00011000 00020 1 %6412,"YOU'RE TOO SLOW!";
00012000 00031 1 ARRAY RESPONSE(0:16):="REACTION TIME: MILLISECONDS";
00013000 00021 1 BYTE ARRAY CRESP(*)=RESPONSE;
00014000 00021 1
00015000 00021 1 INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,ASCII,TIMER,QUIT;
00016000 00021 1
00017000 00021 1 <<END OF DECLARATIONS>>
00018000 00021 1
00019000 00021 1 IN:=FOPEN(INNAME,%45); <<$STDIN>>
00020000 00007 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00021000 00012 1 OUT:=FOPEN(OUTNAME,%414,%1); <<$STDLIST>>
00022000 00022 1 IF < THEN QUIT(2); <<CHECK FOR ERROR>>
00023000 00025 1 FWRITE(OUT,INSTRUCTIONS,35,0); <<USER DIRECTIONS>>
00024000 00032 1 IF < THEN QUIT(3); <<CHECK FOR ERROR>>
00025000 00035 1 LOOP:
00026000 00035 1 FCONTROL(IN,21,DUMMY); <<ENABLE TIMER READ>>
00027000 00041 1 IF < THEN QUIT(4); <<CHECK FOR ERROR>>
00028000 00044 1 FCONTROL(IN,4,TIMEOUT); <<ENABLE TIMEOUT>>
00029000 00050 1 IF < THEN QUIT(5); <<CHECK FOR ERROR>>
00030000 00053 1 CBUF(5):=INTEGER(TIMER,(11:5)+%73; <<GENERATE A CHARACTER>>
00031000 00062 1 FWRITE(OUT,BUFR,3,%320); <<REQUEST USER INPUT>>
00032000 00067 1 IF < THEN QUIT(6); <<CHECK FOR ERROR>>
00033000 00072 1 LGTH:=FREAD(IN,BUFR(3),-1); <<READ CHARACTER>>
00034000 00101 1 IF < THEN <<TIMEOUT OCCURRED>>
00035000 00102 1 BEGIN
00036000 00102 2 FWRITE(OUT,MSG(16),9,0); <<TOO SLOW MESSAGE>>
00037000 00110 2 IF < THEN QUIT(7) ELSE GO NEXT; <<CHECK FOR ERROR>>
00038000 00120 2 END;
00039000 00120 1 IF CBUF(5)<>CBUF(6) THEN <<INCORRECT CHARACTER>>
00040000 00126 1 BEGIN
00041000 00126 2 FWRITE(OUT,MSG(8),8,0); <<WRONG CHARACTER MESSAGE>>
00042000 00134 2 IF < THEN QUIT(8) ELSE GO NEXT; <<CHECK FOR ERROR>>
00043000 00141 2 END;
00044000 00141 1 MOVE RESPONSE(7):=" "; <<RESET RESPONSE TIME>>
00045000 00153 1 FCONTROL(IN,22,TIME); <<READ INPUT TIME>>
00046000 00157 1 IF <> THEN QUIT(9); <<CHECK FOR ERROR>>
00047000 00162 1 ASCII(TIME*10,10,CRESP(15)); <<CONVERT TIME>>
00048000 00171 1 FWRITE(OUT,RESPONSE,17,0); <<REACTION TIME>>
00049000 00177 1 IF < THEN QUIT(10); <<CHECK FOR ERROR>>
00050000 00202 1 NEXT:
00051000 00202 1 FWRITE(OUT,MSG(8),%320); <<CONTINUE TEST?>>
00052000 00207 1 IF < THEN QUIT(11); <<CHECK FOR ERROR>>
00053000 00212 1 FREAD(IN,BUFR(3),-1); <<GET Y/N ANSWER>>
00054000 00220 1 IF < THEN QUIT(12); <<CHECK FOR ERROR>>
00055000 00224 1 IF CBUF(6)="Y" THEN GO LOOP; <<Y-CONTINUE TEST>>
00056000 00232 1 END;
PRIMARY DB STORAGE=%016; SECONDARY DB STORAGE=%00130
NO. ERRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:03; ELAPSED TIME=0:00:10

```

Figure 4-13. Using the TIMER Intrinsic

## OBTAINING THE CURRENT TIME

The `CLOCK` intrinsic returns the actual time (wall time) as a double word. The first word contains the hour of the day and the minute of the hour, the second word contains seconds and tenths of seconds, as follows:

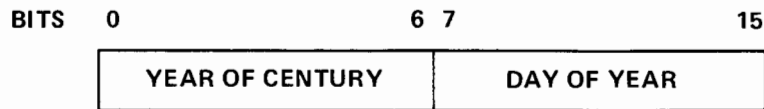


In the following intrinsic call, the above information would be returned to the double-word identifier `TIME`.

```
TIME:=CLOCK;
```

## OBTAINING THE CALENDAR DATE

The `CALENDAR` intrinsic returns a logical value representing the year and day as follows:



In the following intrinsic call, the day and year information would be returned to the logical identifier `DATE`.

```
DATE:=CALENDAR;
```

## OBTAINING PROCESS RUN TIME (USE OF THE CENTRAL PROCESSOR)

The `PROCTIME` intrinsic returns a double integer value representing the duration, in milliseconds, that a process has been running (CPU time).

In the intrinsic call shown below, the process run time would be returned to `TIME`.

```
TIME:=PROCTIME;
```

## INTERPROCESS COMMUNICATION

You can arrange for two processes belonging to the same job/session to communicate with each other through a *job control word*. This word is used by systems programmers to enable a subsystem process to return information to the command executor that initiated that process. Such a communication mechanism is used by the command executors for `:RUN` and various subsystem commands. However, you may find this control word helpful in other applications.

The SETJCW intrinsic is used to set the bits in the job control word. A SETJCW intrinsic call could be

```
SETJCW(WORD);
```

where WORD is a 16-bit logical word whose bits are set by you. Bit zero is reserved for MPE and must be set to 0. If you set this bit to 1, the system displays the following message when your program terminates, either normally or due to an error:

```
ERR 2 (ABNORMAL PROGRAM TERMINATION)
```

Bits 1 through 15 may be set to any pattern.

#### NOTE

In batch mode, the job is terminated unless the :CONTINUE command is used. See the *MPE Commands Reference Manual* for a discussion of :CONTINUE.

A job control word can be requested by a process with the GETJCW intrinsic. The form of the GETJCW intrinsic call is

```
JCW:=GETJCW;
```

The job control word would be returned to JCW in the above intrinsic call.

As an example, consider a job where two processes pass information to each other through the job control word. In one process, you transmit the contents of the word PROCLNK to the job control word. Bit 0 of this word is reserved for MPE and must be set to 0, but the meaning of the remaining 15 bits can be established in any pattern desired. Process A sets the job control word to PROCLNK as follows:

```
SETJCW(PROCLNK);
```

When process B is executed, it obtains this current job control word through the GETJCW intrinsic. In this case, the contents of the job control word are returned to the word STORELNK.

```
STORELNK:=GETJCW;
```



MPE intrinsics can be used to alter certain aspects of device operation. Before any of these intrinsics can be issued against a device, however, the device file must be opened with the FOPEN intrinsic (see Section II, page 2-58).

With the FCONTROL intrinsic, you can

- Change terminal speed. See page 5-10.
- Change input echo facility. See page 5-11.
- Enable and disable the system break function. See page 5-13.
- Enable and disable subsystem break requests. See page 5-14.
- Enable and disable parity checking. See page 5-14.
- Enable and disable tape-mode option. See page 5-15.
- Enable and disable the terminal timer. See page 5-15.
- Read the result from the terminal input timer. See page 5-18.
- Define line-termination characters for terminal input. See page 5-19.
- Control the operation of a primary reader/punch. See page 5-4.

In addition, you can programmatically read paper tapes not containing the X-OFF control character with the PTAPE intrinsic. See page 5-20.

## DEVICE CHARACTERISTICS

### PAPER TAPE READER

The paper tape reader driver is capable of reading tapes in either BINARY or ASCII format. The mode is determined by the ASCII/BINARY bit in the *foptions* parameter of the FOPEN intrinsic. The default condition for this *foption* is ASCII; however, this default condition can be altered with the FCONTROL intrinsic (see page 2-44).

**BINARY MODE.** In binary mode, the device will read the number of words/bytes requested without regard to any special characters present on the tape. Tape leaders are skipped automatically by the hardware. For example, whenever the roller is raised to load a new tape, the device sets an internal flag which causes it to ignore all leading null characters on the next read. After this first read, the flag is reset. Thus, trailers and embedded nulls are as valid as any other characters. The full 8-bit character is read, and characters are packed two characters per word. The end-of-record character is not recognized, so the read is terminated only when the word/byte count is satisfied.

If a time-out occurs (for example, for a tape bind, broken tape, or an attempt to read beyond the end of the tape), an error (CCL) is returned to the calling program.

**ASCII MODE.** In ASCII mode, the following conditions apply by default:

- Bit 8 (parity bit) is set to zero.



- The carriage-return character (%15) is recognized as the end-of-record character. In other words, data transfer stops when the word/byte count is satisfied or when the end-of-record character is detected. (However, if word/byte count is satisfied before the end-of-record character is found, the tape motion continues until end-of-record is detected, with the extra characters being ignored.)
- Data characters are packed automatically with two characters per computer word.

The following characters are not transferred as data and result in the following action:

Carriage-return (%15)	End-of-record.
Line-feed (%12)	Ignored.
X-ON (%21)	Ignored.
X-OFF (%23)	Ignored.
Rub-out (%177)	Ignored.
Control H (%10)	Previous character deleted from data buffer.
Control X (%30)	All data in current record are deleted. The tape is advanced to the next record and the read is restarted.
Control Y (%31)	Ignored.
Null (% 0)	Consecutive nulls are counted to determine end-of-tape.

- Any number of leading null characters are allowed. However, after the first non-zero character in a record, 20 consecutive null characters are treated as the end-of-tape condition and result in the following actions:
  - a. Tape motion is stopped.
  - b. Any characters already read are deleted.
  - c. The message

LDEV #nn NOT READY

is output to the system console. The operator should insert the next tape to be read into the paper tape reader.

- d. When the READY interrupt is detected, the READ request is restarted and operation continues as before.

The entire sequence is invisible to the calling program, except for the delay required to change tapes, so that multiple tapes may be read as if they were a single tape.

#### NOTE

There should always be at least one non-null character, such as a line feed, before the TRAILER to allow it to be distinguished from the LEADER.

All job control cards (i.e., :JOB, :DATA, :EOD, :EOJ, etc.) are recognized in the standard way to allow batch input from paper tape.



## PAPER TAPE PUNCH

The paper tape punch driver is capable of punching tapes in either BINARY or ASCII format. The mode is determined by the ASCII/BINARY bit in the *foptions* parameter of the FOPEN intrinsic. The default condition of ASCII mode may be altered with the FCONTROL intrinsic.

**BINARY MODE.** In BINARY mode, all characters are punched exactly as they appear in the buffer. No characters are deleted or added. Data characters are unpacked automatically, assuming two characters per computer word. In other words, no end-of-record marks such as carriage return are punched unless they are in the buffer. All bit patterns are considered valid and none have any significance as far as the driver is concerned. If you want end-of-record marks on the tape, you must provide them. Note, however, that the paper tape reader driver also attaches no significance to the end-of-record marks.

**ASCII MODE.** In ASCII mode, the following conditions apply by default:

- Trailing blank characters (%40) are punched on the tape. This feature saves paper tape on short records, and also speeds up the net transfer rate for output and for input when the tape is read.
- All other characters, including leading and embedded blanks, are transferred as data. No special characters are recognized.
- A record termination sequence, consisting of X-OFF (%23), a carriage return (%15), followed by a line feed (%12), is appended to the end of each record.
- Data characters are unpacked automatically, assuming two characters per computer word.

## CARD READER

The card reader is a unit record device. The data is read in ASCII mode; that is, two columns are converted to ASCII and packed into the left and right byte of one word. If the read request specifies 80 or more bytes, 80 bytes will be transmitted independent of the data on the card.

## LINE PRINTER

The line printer is a print and space device (post space). The prespace operation is simulated by performing a print operation and then filling the line printer buffer.

Table 5-1 describes the differences between the three subtypes of line printers.

The HP 2610/2614 printers use a 6-bit space count that allows vertical spacing of up to 63 lines when carriage control codes of %200 to %277 are used.

The HP 2607/2613/2618 printers have only a 4-bit space count that imposes a maximum vertical spacing of 15 lines. MPE will not simulate vertical spacing of more than 15 lines for these printers. Carriage control codes %217 to %277 space 15 lines on these printers. User program and

Table 5-1. Line Printer Differences

SUBTYPE	HP PRODUCT NO.	SUPPRESS SPACE	VERTICAL SPACING	CHANNELS 9-12
0	2610	YES	63	NO**
0	2614	YES	63	NO**
1	2607	NO*	15	NO**
2	2613	YES	15	YES
2	2618	YES	15	YES

\*A suppress space request to subtype 1 will result in a single space.

\*\*A request to skip to channels 9-12 (carriage control codes %310-%313) will result in a single space on subtypes 0 and 1.

subsystems such as RPG which permit vertical spacing of up to 112 lines should always perform vertical spacing of more than 15 lines by issuing successive 15-line spaces. This technique will permit such programs to be device independent. Also, it is impossible for such programs to accurately determine the printer subtype under certain conditions when the file is spooled. See the *devtype* parameter of the FGETINFO intrinsic in Section II.

### MAGNETIC TAPE

The magnetic tape unit reads and writes variable length records in packed binary mode. Each word of data is represented by two tape characters. On read requests, the amount of data transferred is the lesser of the read request length and the tape record length.

After write operations, when the end of tape reflective marker is detected, an EOT indication is returned. A request initiated before the EOT marker was detected is completed but an EOT indication is returned.

### PRINTING READER/PUNCH

The HP 30119A printing reader/punch is supported in three ways by MPE:

1. As a card reader which, from a user program's viewpoint, behaves exactly like the HP 30106A/30107A card readers. This mode of operation prevails when the device is opened by device class name and the device class access type is input only. In this mode, default is select primary hopper and primary stacker.
2. As a card punch. This mode of operation prevails when the device is opened by device class name and the device class access type is output only. In this mode, default is select secondary hopper and secondary stacker.
3. As a printing reader/punch with two input hoppers and two output stackers over which the user has complete control. This mode of operation prevails when the device is opened by logical device number or by device class name when the device class access type is input/output.

In the mode of operation described under 3 above, you can control all aspects of the device with the intrinsic call

```
FCONTROL(filenum,0,param);
```

where the bit settings of *param* signify the following:

- Bits (0:6) = Reserved for MPE. These bits should be set to zero.
- Bit (6:1) = 0. Select no inhibit feed on writes.  
= 1. Select inhibit feed on writes.
- Bit (7:1) = 0. Select punch on writes.  
= 1. Select no punch on writes.
- Bit (8:1) = 0. Select print on writes.  
= 1. Select no print on writes.
- Bit (9:1) = 0. Select print and punch same data on writes.  
= 1. Select print and punch separate data on writes.
- Bit (10:1) = 0. Select primary stacker.  
= 1. Select secondary stacker.
- Bit (11:1) = 0. Select primary hopper.  
= 1. Select secondary hopper.
- Bits (12:4) = Reserved for MPE. These bits should be set to zero.

When the device is opened for the first time, all of above parameter selections assume a default value of zero. Subsequent opens, however, do not necessarily yield these default values; the parameter selections for such opens assume the values established by previous opens.

The FREAD and FWRITE intrinsics perform the following actions for the printing reader/punch.

#### FREAD

- a. Feeds a card from the hopper selected.
- b. Moves card from wait station (if present) to stacker last selected (no punch or print performed).
- c. Reads data from (a) and transfers it to caller's buffer.
- d. The mode (ASCII or column binary) of the read is specified on each read/write.
- e. The following parameter selections have no effect:
  - same/separate print data;
  - print/no print;
  - punch/no punch;
  - inhibit input feed.

#### FWRITE

- a. Moves card from the wait station to the stacker last selected.
- b. Prints and/or punches card (1) using data in caller's buffer.
- c. If inhibit input feed has been selected, no card is fed from hoppers. If inhibit feed has not been selected, card is fed from hopper last selected; any data on that card is lost.
- d. If separate print data has been selected, a double buffer is expected and punch data is extracted from the first part, print data from the second part. No print and no punch selections are still honored. If no print or no punch is on, a single length buffer suffices. If separate print data has not been selected, then the same data is printed and punched, unless no print or no punch has been selected.

- e. The mode (ASCII or column binary) used will be the one last selected.
- f. If no print is selected, printing is inhibited.
- g. If no punch is selected, punching is inhibited.

The FCONTROL *param* selections (bits 6 through 11) remain in effect until changed by another FCONTROL intrinsic call.

## LINE PRINTER AND TERMINAL CARRIAGE-CONTROL CODES

Line printer and terminal carriage-control codes are shown in table 5-2. All of the carriage-control codes shown in table 5-2 may be used as the value of the *param* parameter of FCONTROL (when *controlcode* = 1) regardless of whether the file is opened with CCTL or NOCCTL specified in the FOPEN intrinsic. When the file is opened with CCTL, the carriage-control codes may be used in either of the following ways via FWRITE:

1. As the value of the *control* parameter.
2. When *control* = 1, as the first byte of the *target* array.

Carriage-control codes greater than %403 cause an error return with no operation performed.

The default mode controls are post spacing with automatic page eject.

## END-OF-FILE INDICATION

An end-of-file indication is returned by the card reader and tape drivers under conditions specified by the initiators of read requests. The types of requests and the end-of-file classes are as follows:

Type	Class of end-of-file
A	All records that begin with a colon (:).
B	All records that contain, starting in the first byte, :EOD, :EOJ, :JOB and :DATA (See Note.)
E	Hardware-sensed end-of-file

### NOTE

*If the word count is less than 3 or the byte count is less than 6, then Type B reads are converted to Type A reads.*

In utilizing the card/tape devices as files via the File System, the following types are assigned.

File Specified	Type
\$STDIN	Type A.
\$STDINX	Type B.
Dev=CARD/TAPE	Type B, if device accepts jobs or data. Type E, if device does not accept jobs or data.

Table 5-2. Line Printer and Terminal Carriage-Control Codes

Octal Code	ASCII Symbol	Carriage Action
%40	" "	*Single-space.
%60	"0"	*Double-space.
%61	"1"	Page-eject (form-feed).
%53	"+"	No space, return (next printing at column 1).
%2nn		Space nn lines. (No automatic page eject.)
(where n is any digit from 0 through 7)		
%300		Page-eject (Tape Channel 1).
%301		Skip to bottom of form (Tape Channel 2).
%302		Single-spacing (with automatic page eject). (Tape Channel 3.)
%303		Single-space on next odd-numbered line (with automatic page eject). (Tape Channel 4.)
%304		Triple-space (with automatic page eject). (Tape Channel 5.)
%305		Space 1/2 page (with automatic page eject). (Tape Channel 6.)
%306		Space 1/4 page (with automatic page eject). (Tape Channel 7.)
%307		Space 1/6 page (with automatic page eject). (Tape Channel 8.)
%310		Space to bottom of form. (Tape Channel 9.)
%311		Skip to Tape Channel 10. (User option.)
%312		Skip to Tape Channel 11. (User option.)
%313		Skip to Tape Channel 12. (User option.)
%320		No space, no return. (Next printing physically follows this.)
%0 – %37	}	Same as %40
%41 – %52		
%54 – %57		
%62 – %77		
%314 – %317		
%321 – %377		
%400 or %100		Set post-space movement option; this first prints, then spaces. If previous option set was pre-space movement option, the driver outputs a line (and suppresses spacing) to clear the buffer.
%401 or %101		Set pre-space movement option; this first spaces, then prints.
%402 or %102		Set single-space option, with automatic page eject (60 lines per page).
%403 or %103		Set single-space option <i>without</i> automatic page eject (66 lines per page).
* Spacing with or without automatic page eject can be selected.		

Any subsequent requests to the driver after an end-of-file condition is sensed, are rejected with an end-of-file indication.

When reading from an unlabeled tape file, a request encountering a tape mark responds with an end-of-file indication but succeeding requests are allowed to continue reading data past the tape mark. Under these conditions, it is the responsibility of the caller to protect against the occurrence of data beyond an end-of-file and to prevent reading off the end of the reel.

## TERMINALS

Terminals are supported by MPE through the following controller:

Terminal controller (each controls up to 16 terminals).

The terminal controller supports 103A and 202A modems, and hardwired terminals.

**TERMINAL TYPES.** The following terminals can be connected to an HP 3000 Computer running under MPE:

- HP 2600A or DATAPOINT 3300 Keyboard — Display Terminal (10/15/30/60/120/240 cps). Terminal type 4 for the :HELLO command.
- ASR-33 EIA-compatible (HP 2749B) Terminal (10 cps). Terminal type 0 for the :HELLO command.
- ASR-35 EIA-compatible Terminal (10 cps). Terminal type 0 for the :HELLO command.
- General Electric TermiNet 300 Data Communications Terminal, Model B (10/15/30 cps) with Paper Tape Reader/Punch, Option 2. Terminal type 6 for the :HELLO command.

### NOTE

This terminal must be equipped for “ECHO PLEX”.

- Memorex 1240 Communications Terminal (10/15/30/60/120/240 cps). Terminal type 5 for the :HELLO command.

### NOTE

This terminal must be equipped with the even parity checking option.

- Execuport 300 Data Communications Transceiver Terminal (10/15/30 cps). Terminal type 3 for the :HELLO command.
- ASR-37 Teleprinter Terminal with Paper Tape Reader/Punch (15 cps). Terminal type 1 for the :HELLO command.
- HP 2615A Terminal (10/15/30/60/120/240 cps). Terminal type 9 for the :HELLO command.

- HP 2640A Interactive Display Terminal (10 to 240 cps). Terminal type 10 for the :HELLO command.
- HP 2644A Mini DataStation (10 to 240 cps). Terminal type 10 for the :HELLO command.

Terminals equipped with the automatic linefeed feature (operator selectable) must be operated with this feature OFF.

**SPECIAL KEYS.** The following keys have special significance to MPE.

Key	Meaning
X <sup>c</sup>	Deletes (ignores) the line being typed and then reads any following characters. The system responds with a triple exclamation point (!!!) followed by a carriage-return and line-feed. (The superscript <sup>c</sup> denotes a control character. Thus, "X <sup>c</sup> " means "CONTROL-X.")
H <sup>c</sup>	Deletes the previous character. (To delete <i>n</i> characters, enter <i>n</i> H <sup>c</sup> 's.) See note below.
Q <sup>c</sup>	Places terminal in tape mode, allowing input from paper tape. When enabled, the tape-mode option inhibits the implicit line feed normally issued by MPE each time a carriage return is entered. The tape-mode option also inhibits responses to H <sup>c</sup> and X <sup>c</sup> entries. Thus, when H <sup>c</sup> is received and tape mode is in effect, no exclamation points (!!!) are sent to the terminal.
Y <sup>c</sup>	If the terminal is not in tape mode, Y <sup>c</sup> requests subsystem break (terminating program or command execution.) If the terminal is in tape mode, Y <sup>c</sup> returns it to the keyboard mode.
<i>BREAK</i>	Requests a system break.
<i>ESC:</i>	Places the terminal in the echo-on mode so that characters input are echoed to the terminal printer by MPE.
<i>ESC;</i>	Places the terminal in echo-off mode so that characters input are not echoed by MPE.
<i>LINE FEED</i>	For any terminal with a line-feed key, the logon user may strike this key and a carriage return will be echoed. The line feed character is not transmitted to the input buffer. This mechanism permits multiple lines to be entered in response to a single read request (i.e., the length of a read request from a terminal is not constrained by the carriage or line width).

The defined control characters X<sup>c</sup>, H<sup>c</sup>, Q<sup>c</sup>, and Y<sup>c</sup> are recognized even when following an ESC key entry. However, entry of ESC followed by any character (other than one of these control characters, a colon, or a semicolon) is read as a 2-character string in the user's input stream.



## NOTE

The line correction mechanism (H<sup>c</sup>) works in the following ways for all terminals including the system console:

- CRT Terminals  
All currently supported CRT terminals can physically backspace the cursor; therefore, H<sup>c</sup> causes the cursor to be backspaced one position, leaving the cursor positioned over the character to be replaced. The physical backspacing of the cursor does not erase the character from the screen, but the character has been deleted from MPE's internal buffer.
- Hardcopy Terminals
  - a. Terminals which have physical backspace capability:  
H<sup>c</sup> causes a physical backspace to occur. In addition, a line feed is performed unless the previous character also was a H<sup>c</sup>. The result is that you begin typing beneath the first character to be replaced.
  - b. Terminals with no physical backspace capability:  
No backspacing takes place. Each H<sup>c</sup> echoes a backslash (\) unless in tape mode.

**CHANGING TERMINAL CHARACTERISTICS.** Certain aspects of terminal operation can be changed with the FCONTROL and PTAPE intrinsics. Before these intrinsics can be used in a program to change terminal characteristics, however, the terminal/file must be opened with the FOPEN intrinsic.

**Changing Terminal Speed.** MPE supports terminals that run at speeds ranging from 10 to 240 characters per second (cps). You can programmatically change these speeds with the FCONTROL intrinsic. This capability allows a user running a mark sense card reader coupled to a terminal to operate the two devices at different speeds (for example, the card reader at 240 cps for input and the terminal at 10 cps for output). The FCONTROL intrinsic is not valid for terminals that operate at only one speed.

The format for this application of the FCONTROL intrinsic is

```
IV      IV      L
FCONTROL(filename,controlcode,speed);
```

The parameters are

*filename*                    *integer by value (required)*  
A word identifier supplying the file number of the terminal for which the speed is to be changed.

***controlcode***                    *integer by value (required)*  
The decimal integer 10 to change the input speed or 11 to change the output speed.

***speed***                            *logical (required)*  
A word identifier that specifies the new speed desired: 10, 14, 15, 30, 60, 120, or 240 cps. When the FCONTROL intrinsic is executed, the previous input or output speed is returned to the calling process through the *speed* parameter.

The condition codes are

- |     |  |
|-----|--|
| CCE | Request granted.   |
| CCG | Not returned by this application of FCONTROL.  |
| CCL | Request denied. The process does not own the logical device, or this device is not a terminal, or the speed entered is not acceptable. |

As an example, to change the current input speed of the terminal identified by the file number stored in the word TERMFN from 60 to 120 cps, the following call could be used. The word SPEED contains the value 120.

```
FCONTROL(TERMFN,10,SPEED);
```

After the intrinsic is executed, the word SPEED contains the integer 60 (the previous speed).

**Changing Input Echo Facility.** You can programmatically determine whether MPE transmits (echoes) input from the terminal keyboard back to the terminal printer by calling the FCONTROL intrinsic to turn the echo facility on or off.

When the echo facility is *on*, input read from the terminal is echoed to the terminal's printer by MPE. If the terminal is operating in full-duplex mode, the echoed information appears as normal printed lines. If the terminal is in half-duplex mode, however, the echoed printing is illegible — as you enter input on such terminals, it is simultaneously printed by the terminal itself and subsequently overwritten by the echoed information. Where a terminal can operate in either full- or half-duplex mode, the mode is selected by a switch on the terminal. When you log on, all terminals are assumed to have the echo facility *on*.

When the echo facility is *off*, input read from the terminal is not echoed to the terminal's printer by MPE. If the terminal is operating in full-duplex mode, no printing appears. If the terminal is in half-duplex mode, the input is copied by the terminal itself, and appears as normal, printed lines. Bear in mind that the only way printing can be suppressed is with the echo facility *off* and the terminal in full-duplex mode, as illustrated in figure 5-1.

In addition to the FCONTROL intrinsic, the echo facility also can be switched on and off by entering the characters:

- ESC: to turn the echo facility on.
- ESC; to turn the echo facility off.

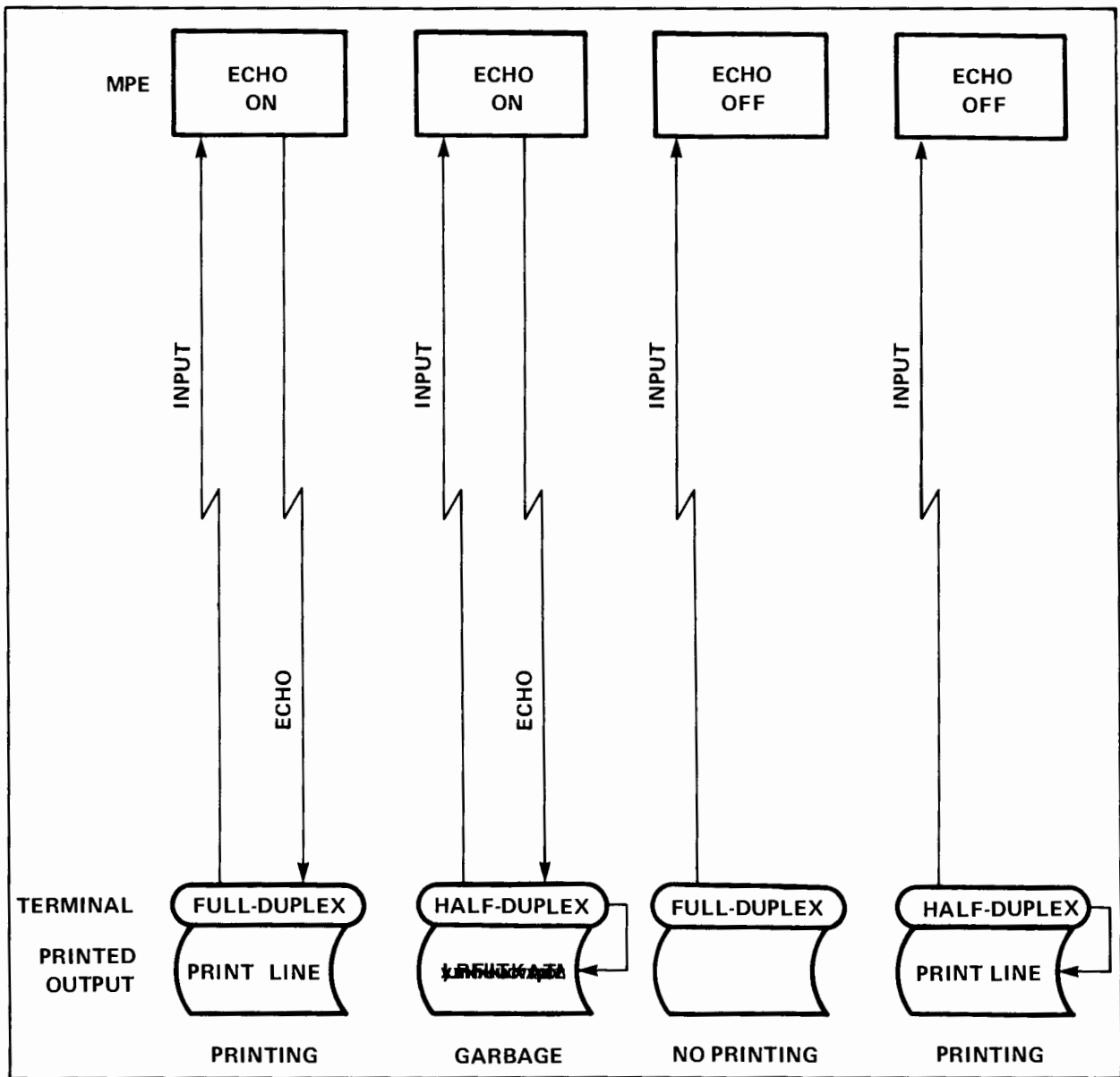


Figure 5-1. Echo Facility vs Duplex Mode

The format for this application of the FCONTROL intrinsic is

```

IV      IV L
FCONTROL(filenum,controlcode,last);

```

The parameters are

***filenum***                    *integer by value (required)*  
A word identifier supplying the file number of the terminal.

***controlcode***                    *integer by value (required)*  
The integer 12 to turn the echo facility on, or 13 to turn it off.

***last***                                *logical (required)*  
A word identifier to which the previous echo facility is returned, where  
0 = echo on  
1 = echo off.

The condition codes are

CCE                                Request granted.

CCG                                Not returned by this application of FCONTROL.

CCL                                Request denied because the file number specified did not belong to this process or this device is not a terminal.

As an example, to turn the echo facility off, the following intrinsic call could be used:

```
FCONTROL(TERMFN,13,LAST);
```

After the intrinsic is executed, the word LAST contains the value 0 or 1 to reflect the previous echo facility status.

**Enabling and Disabling System Break Function.** You can programmatically suspend or enable a terminal's ability to react to a system break request with the FCONTROL intrinsic. System break requests are initialized by pressing the BREAK key or by calling the CAUSEBREAK intrinsic.

The format for this application of the FCONTROL intrinsic is

```
                  IV          IV          L  
FCONTROL(filenum,controlcode,anyinfo);
```

The parameters are

***filenum***                            *integer by value (required)*  
A word identifier supplying the file number of the terminal.

***controlcode***                        *integer by value (required)*  
The integer 15 to enable the break function, or 14 to disable the break function.

***anyinfo***                            *logical (required)*  
Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE                                Request granted.

CCG Not returned by this application of FCONTROL.  
CCL Request denied because the file number specified did not belong to this process or the device is not a terminal.

As an example, to enable the break function, the following intrinsic call could be used:

```
FCONTROL(TERMFN,15,DUMMY);
```

**Enabling and Disabling Subsystem Break Function.** All terminals are initially set to disable (not accept) subsystem break requests, generated by entering CONTROL-Y during a session. You can, however, programmatically enable and again disable a terminal's ability to react to subsystem break requests with the FCONTROL intrinsic.

The format for this application of the FCONTROL intrinsic is

```
IV IV L  
FCONTROL(filenum,controlcode,anyinfo);
```

The parameters are

*filenum* integer by value (required)  
A word identifier supplying the file number of the terminal.

*controlcode* integer by value (required)  
The integer 17 to enable the subsystem break function, or 16 to disable the subsystem break function.

*anyinfo* logical (required)  
Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE Request granted.  
CCG Not returned by this application of FCONTROL.  
CCL Request denied because the file number specified did not belong to this process or the device is not a terminal.

As an example, to enable the subsystem break function, the following intrinsic call could be used:

```
FCONTROL(TERMFN,17,DUMMY);
```

**Enabling and Disabling Parity Checking.** All terminals and mark-sense card readers are initially set to disable parity checking during read operation. They may, however, be programmatically enabled for parity checking with the FCONTROL intrinsic. Then, the parity of the data received is checked against the parity computed by the asynchronous channel multiplexer. If a parity error is detected,

an error code is made available through the FCHECK intrinsic. When you are running a card reader coupled to a terminal, the ability to enable/disable parity checking allows you to obtain optimum utilization of the card reader by running it at 240 characters per second.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
FCONTROL(filename,controlcode,anyinfo);
```

The parameters are



<i>filename</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<i>controlcode</i>	<i>integer by value (required)</i> The integer 24 to enable parity checking, or 23 to disable parity checking.
<i>anyinfo</i>	Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirement of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because the file number specified did not belong to this process or the device is not a terminal.

As an example, to enable parity checking, the following intrinsic call could be used:

```
FCONTROL(TERMFN,24,DUMMY);
```

**Enabling and Disabling Tape-Mode Option.** You can programmatically enable or disable the tape-mode option for a terminal with the FCONTROL intrinsic. When enabled, the tape-mode option inhibits the implicit line feed normally issued by MPE each time a carriage return is entered. The tape mode option also inhibits responses to H<sup>c</sup> and X<sup>c</sup> entries. Thus, when H<sup>c</sup> is received and tape mode is in effect, no exclamation points (!!!) are sent to the terminal.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
FCONTROL(filename,controlcode,anyinfo);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 19 to enable tape mode, or 18 to disable tape mode.
<b><i>anyinfo</i></b>	<i>logical (required)</i> Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves to other purpose and is not modified by the intrinsic.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because the file number specified did not belong to this process or the device is not a terminal.

As an example, to enable tape mode, the following intrinsic call could be used:

```
FCONTROL(TERMFN,19,DUMMY);
```

**Enabling and Disabling The Terminal Input Timer.** The terminal input timer records the time required to satisfy an input request on the terminal, from the time the input is requested until it is completed. This applies only to unbuffered, serial terminal input requests. You can program-matically enable or disable the terminal input timer with the FCONTROL intrinsic.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L  
FCONTROL(filenum,controlcode,anyinfo);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 21 to enable the time, or 20 to disable the timer.
<b><i>anyinfo</i></b>	<i>logical (required)</i> Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because the file number specified did not belong to this process or the device is not a terminal.

Figure 5-2 contains a program that generates an ASCII character, instructs the user to enter this character on the terminal, then measures and displays the reaction time of the user.

The statement

```
FCONTROL(IN,21,DUMMY);
```

enables the terminal input timer so that the reaction time of the user can be measured. The parameter IN supplies the file number of the terminal and was obtained through the FOPEN intrinsic call (see statement 19 in the program).

#### NOTE

The statement

```
FCONTROL(IN,4,TIMEOUT);
```

is used to apply a time-out interval to the terminal. This application of FCONTROL is used in conjunction with FREAD intrinsic calls issued against the terminal. The time-out interval specified in this case is 10 seconds (see statement number 5 in the program). If there is no response to the FREAD intrinsic call (see statement number 33) within 10 seconds, a CCL condition code is returned and the program displays the message

```
YOU'RE TOO SLOW
```



```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INNAME(0:5):="INPUT ";
00004000 00004 1 BYTE ARRAY OUTNAME(0:6):="OUTPJT ";
00005000 00005 1 INTEGER IN,OUT,LGTH,DUMMY,TIME,TIMEOUT:=10;
00006000 00005 1 ARRAY BUFR(0:3):="TYPE X",0;
00007000 00004 1 BYTE ARRAY CBUF(*)=BUFR;
00008000 00004 1 ARRAY INSTRUCTIONS(0:34):="REACTION TIMER: ",%6412,
00009000 00011 1 "TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN, ";
00010000 00043 1 ARRAY MSG(0:24):="TRY AGAIN? (Y/N)","WRONG CHARACTER.",
00011000 00020 1 %6412,"YOU'RE TOO SLOW!";
00012000 00031 1 ARRAY RESPONSE(0:16):="REACTION TIME: MILLISECONDS";
00013000 00021 1 BYTE ARRAY CRESP(*)=RESPONSE;
00014000 00021 1
00015000 00021 1 INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,ASCII,TIMER,QUIT;
00016000 00021 1
00017000 00021 1 <<END OF DECLARATIONS>>
00018000 00021 1
00019000 00021 1 IN:=FOPEN(INNAME,%45); <<STDIN>>
00020000 00007 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00021000 00012 1 OUT:=FOPEN(OUTNAME,%414,%1); <<STDLIST>>
00022000 00022 1 IF < THEN QUIT(2); <<CHECK FOR ERROR>>
00023000 00025 1 FWRITE(OUT,INSTRUCTIONS,35,0); <<USER DIRECTIONS>>
00024000 00032 1 IF < THEN QUIT(3); <<CHECK FOR ERROR>>
00025000 00035 1 LOOP:
00026000 00035 1 FCONTROL(IN,21,DUMMY); <<ENABLE TIMER READ>>
00027000 00041 1 IF < THEN QUIT(4); <<CHECK FOR ERROR>>
00028000 00044 1 FCONTROL(IN,4,TIMEOUT); <<ENABLE TIMEOUT>>
00029000 00050 1 IF < THEN QUIT(5); <<CHECK FOR ERROR>>
00030000 00053 1 CBUF(5):=INTEGER(TIMER).(11:5)+%73; <<GENERATE A CHARACTER>>
00031000 00062 1 FWRITE(OUT,BUFR,3,%320); <<REQUEST USER INPUT>>
00032000 00067 1 IF < THEN QUIT(6); <<CHECK FOR ERROR>>
00033000 00072 1 LGTH:=FREAD(IN,BUFR(3),-1); <<READ CHARACTER>>
00034000 00101 1 IF < THEN <<TIMEOUT OCCURRED>>
00035000 00102 1 BEGIN
00036000 00102 2 FWRITE(OUT,MSG(16),9,0); <<TOO SLOW MESSAGE>>
00037000 00110 2 IF < THEN QUIT(7) ELSE GO NEXT; <<CHECK FOR ERROR>>
00038000 00120 2 END;
00039000 00120 1 IF CBUF(5)<>CBUF(6) THEN <<INCORRECT CHARACTER>>
00040000 00126 1 BEGIN
00041000 00126 2 FWRITE(OUT,MSG(8),8,0); <<WRONG CHARACTER MESSAGE>>
00042000 00134 2 IF < THEN QUIT(8) ELSE GO NEXT; <<CHECK FOR ERROR>>
00043000 00141 2 END;
00044000 00141 1 MOVE RESPONSE(7):=" "; <<RESET RESPONSE TIME>>
00045000 00153 1 FCONTROL(IN,22,TIME); <<READ INPUT TIME>>
00046000 00157 1 IF <> THEN QUIT(9); <<CHECK FOR ERROR>>
00047000 00162 1 ASCII(TIME*10,10,CRESP(15)); <<CONVERT TIME>>
00048000 00171 1 FWRITE(OUT,RESPONSE,17,0); <<REACTION TIME>>
00049000 00177 1 IF < THEN QUIT(10); <<CHECK FOR ERROR>>
00050000 00202 1 NEXT:
00051000 00202 1 FWRITE(OUT,MSG,8,%320); <<CONTINUE TEST?>>
00052000 00207 1 IF < THEN QUIT(11); <<CHECK FOR ERROR>>
00053000 00212 1 FREAD(IN,BUFR(3),-1); <<GET Y/N ANSWER>>
00054000 00220 1 IF < THEN QUIT(12); <<CHECK FOR ERROR>>
00055000 00224 1 IF CBUF(6)="Y" THEN GO LOOP; <<Y-CONTINUE TEST>>
00056000 00232 1 END.
PRIMARY DB STORAGE=%016; SECONDARY DB STORAGE=%00130
NO. ERRORS=000; NO. WARNINGS=000
PROFESSOR TIME=0:00:03; ELAPSED TIME=0:00:10

```

Figure 5-2. Using the FCONTROL Intrinsic to Enable and Read the Terminal Input Timer

The results of running the program of figure 5-2 are shown below:

```
:RUN TIME
```

```
REACTION TIMER:  
TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN.  
TYPE M  
YOU'RE TOO SLOW!  
TRY AGAIN? (Y/N)Y  
TYPE >>  
REACTION TIME: 9670  MILLISECONDS  
TRY AGAIN? (Y/N)Y  
TYPE BB  
REACTION TIME: 4090  MILLISECONDS  
TRY AGAIN? (Y/N)Y  
TYPE UU  
REACTION TIME: 1790  MILLISECONDS  
TRY AGAIN? (Y/N)Y  
TYPE IO  
WRONG CHARACTER.  
TRY AGAIN? (Y/N)N  
  
END OF PROGRAM  
:
```

**Reading The Terminal Input Timer.** You can read the result from the terminal input timer with the FCONTROL intrinsic. The result will be valid only if the terminal input was preceded by a call to enable the terminal input timer. If valid, the result is the time, in hundredths of seconds, required for the last direct, unbuffered serial input on the terminal.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L  
FCONTROL(filenum,controlcode,inputtime);
```

The parameters are

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<i>controlcode</i>	<i>integer by value (required)</i> The integer 22.



***controlcode***                    *integer by value (required)*  
The integer 25.

***character***                    *logical (required)*  
A word identifier supplying (in the right byte) the character to be used as a line terminator. The left byte of this word can contain any information — it is ignored by the intrinsic. If zero is specified in the *character* parameter, the terminal reverts to its normal line-control operation.

The condition codes are

CCE                                Request granted.

CCG                                Not returned by this application of FCONTROL.

CCL                                Request denied. The character specified is not allowed.

The following characters are not allowed as line-terminating characters in the *character* parameter.

ASCII Character	Octal Code
CONTROL-Y (Y <sup>c</sup> )	31
Carriage return	15
Line feed	12
ESC	33
CONTROL-X (X <sup>c</sup> )	30
CONTROL-H (H <sup>c</sup> )	10
CONTROL-Q (Q <sup>c</sup> )	21
X-OFF	23

If the value of *character* equals zero, the terminal reverts to normal line-control operation.

As an example, to specify a period as the standard line terminator for a terminal, the following intrinsic call could be used:

```
FCONTROL(TERMFN,25,CHAR);
```

The word CHAR contains the octal value %55 (indicating a period) in the right byte. (The left byte can be specified as any value.)

**Enabling and Disabling Binary Transfers.** Binary transfers can be enabled or disabled with the FCONTROL intrinsic. (Binary transfers are disabled in normal MPE operation.)

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L  
FCONTROL(filenum,controlcode,anyinfo);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 26 to disable binary transfers, or 27 to enable binary transfers.
<b><i>anyinfo</i></b>	<i>logical (required)</i> Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because an error occurred.

**Enabling and Disabling User Block Transfers.** User mode block transfers can be enabled or disabled with the FCONTROL intrinsic. (User mode block transfers are disabled in normal MPE operation.)

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
FCONTROL(filenum,controlcode,anyinfo);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 28 to disable user mode block transfers, or 29 to enable user mode block transfers.
<b><i>anyinfo</i></b>	<i>logical (required)</i> Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because an error occurred.

Enabling and Disabling Line Deletion Echo Suppression. In normal MPE operation, Control-X is interpreted as a line deletion operation and the character string “!!!” is echoed on the terminal. You can suppress the line deletion echo, so that the character string is not displayed on the terminal, with the FCONTROL intrinsic.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
FCONTROL(filenum,controlcode,anyinfo);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 34 to disable the line deletion echo, or 35 to enable the line deletion echo.
<b><i>anyinfo</i></b>	<i>logical (required)</i> Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic; however, it serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because an error occurred.

Setting Parity. The FCONTROL intrinsic can be used to specify the parity, if any, to be used in transmitting data to a terminal. Parity is generated on the right seven bits or the full eight bits of a character.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
FCONTROL(filenum,controlcode,param);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 36.

## WRITING A USER FILE LABEL

When a disc file is created, MPE automatically supplies a file label in the first sector of the first extent occupied by that file. User-supplied labels are located in the sectors immediately following the MPE file label. The number of records allowed for user-supplied labels for any file must be specified in the *userlabels* parameter of the FOPEN intrinsic call that creates the file.

***param***                                    *logical (required)*  
A logical word, as follows:  
0 - No parity generated. All eight bits are transmitted.  
1 - No parity generated. Bit 8 is always set to 1.  
2 - Even parity generated if bit 8 is 0; odd parity generated if bit 8 is 1.  
3 - Odd parity generated on seven bits. (This is the normal mode of operation.)

The condition codes are

CCE                                        Request granted.  
CCG                                        Not returned by this application of FCONTROL  
CCL                                        Request denied because an error occurred.

**Allocating a Terminal.** A terminal can be removed from speed-sensing mode, initialized according to the type and speed specified by the FCONTROL intrinsic, and set on line. (The terminal cannot be configured as :JOB or :DATA accepting.)

The format for this application of the FCONTROL intrinsic is

```
                  IV      IV      L  
FCONTROL(filename,controlcode,param);
```

The parameters are

***filename***                                *integer by value (required)*  
A word identifier supplying the file number of the terminal.

***controlcode***                            *integer by value (required)*  
The integer 37.

***param***                                    *logical (required)*  
A logical word, as follows:  
Bits (11:5) - Terminal type (see page 5-8).  
Bits (0:11) - Speed in characters per second.

If *param* is set to zero, the speed and terminal type specified when the system was configured will be used to initialize the device.

The condition codes are

CCE                                        Request granted.  
CCG                                        Not returned by this application of FCONTROL.  
CCL                                        Request denied because an error occurred.

**Setting Terminal Type.** The terminal type can be set with the FCONTROL intrinsic.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
    FCONTROL(filenum,controlcode,param);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 38.
<b><i>param</i></b>	<i>logical (required)</i> A logical word specifying the terminal type (see page 5-8).

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because an error occurred.

**Obtaining Terminal Type Information.** The terminal type can be determined with the FCONTROL intrinsic.

This application of FCONTROL may be used before a terminal is allocated (with FCONTROL *controlcode* parameter 37, see page 5-24) to return the terminal type specified when the system was configured. A value of 31 is returned in *param* if no terminal type was specified at configuration time.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
    FCONTROL(filenum,controlcode,param);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 39.
<b><i>param</i></b>	<i>logical (required)</i> A logical identifier to which is returned the terminal type (see page 5-8) as specified at log-on time or when the terminal was allocated.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because an error occurred.



**Obtaining Terminal output Speed.** The terminal output speed can be determined with the FCONTROL intrinsic.

This application of FCONTROL may be used before a terminal is allocated (with FCONTROL *controlcode* parameter 37, see page 5-24) to return the speed specified when the system was configured. A value of zero is returned in *param* if no speed was specified at configuration time.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
FCONTROL(filenum,controlcode,param);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 40.
<b><i>param</i></b>	<i>logical (required)</i> A logical identifier to which the terminal output speed in characters per second is returned.

The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because an error occurred.

**Setting Unedited Terminal Mode.** The terminal can be set in unedited mode with the FCONTROL intrinsic. In unedited mode, all characters, except Attention and end-of-record, are passed to the terminal. Note that only seven bits of the characters are passed, the parity bit is stripped.

The end-of-record character terminates input from the terminal in unedited mode (as a carriage return does in normal mode).

The Attention character terminates input and causes a Subsystem Break in unedited mode (as a Control-Y does in normal mode).

No automatic line feed is output to the terminal when input terminates in unedited mode.

The unedited mode is reset to normal when an FCLOSE intrinsic call is issued against the terminal, or when the *chars* parameter of FCONTROL equals zero. (See below.)

The unedited mode is disabled while the terminal is in Break or Console mode.

The format for this application of the FCONTROL intrinsic is

```
          IV      IV      L
FCONTROL(filenum,controlcode,chars);
```

The parameters are

<b><i>filenum</i></b>	<i>integer by value (required)</i> A word identifier supplying the file number of the terminal.
<b><i>controlcode</i></b>	<i>integer by value (required)</i> The integer 41.
<b><i>chars</i></b>	<i>logical (required)</i> A logical word, as follows: Bits (0:8) - Attention character. Bits (8:8) - End-of-record character. If <i>chars</i> = 0, the unedited mode is reset to normal.



The condition codes are

CCE	Request granted.
CCG	Not returned by this application of FCONTROL.
CCL	Request denied because an error occurred.

**Reading Paper Tapes Without X-OFF Control.** The X-OFF control character, written by pressing the X-OFF key on a teletype terminal, is used to delimit data input on paper tape. When a teletype tape reader encounters this character while reading a tape, reading halts until the program requests more input data.

You can programmatically read data from paper tapes not containing the X-OFF control character, or from tapes input through terminals not recognizing this character with the PTAPE intrinsic. In the latter case, the X-OFF characters are stripped from the tape. Tape input terminates when Y<sup>c</sup> is encountered, returning control to the terminal. Prior to calling the PTAPE intrinsic, you must be sure to position the end-of-file pointer to the proper position in the file. If you are reading more than one tape, you should specify, in the FOPEN intrinsic call that opens the file, the *append-only* access type, and a *variable-length* record format before the first PTAPE intrinsic call. Additionally, you should set the end-of-file pointer to zero, if necessary, before issuing the first PTAPE intrinsic call.

A PTAPE intrinsic call such as

```
PTAPE(TERMFN,DISCFL);
```

could be used to read a paper tape not containing the X-OFF control character, or to read a paper tape input through a terminal that does not recognize this character. The data would be stored in the disc file whose file number is specified by DISCFL.

## USING THE FCARD INTRINSIC TO OPERATE THE HP 7260A OPTICAL MARK READER

The FCARD intrinsic allows you to control the operation of the HP 7260A Optical Mark Reader (OMR) programmatically. This is achieved through passing a parameter value (*recode*), corresponding to the function of FCARD desired, from a program to FCARD. FCARD returns to the calling program parameter values which indicate the success or the cause of failure of execution, the status of the 7260A, the file number of the 7260A/terminal file for which the function has been performed and the number of columns read at the completion of a read request.

The program shown in figure 5-3 performs the following:

1. Opens the 7260A/terminal file.
2. Displays operator instructions.
3. Temporarily suspends program operation awaiting the depression of the 7260A READY switch.
4. Reads ten cards in the ASCII reading format.
5. Displays the number of columns read from each card.
6. Examines *status* for empty input hopper status.
7. Examines output *recode* values of each request.
8. Closes the 7260A/terminal file.

Under the label OPENFILE, the program requests that a 7260A/terminal file be opened for access and that the file number of this file be returned to the program in the parameter *filenum* by assigning to *recode* a value of 0 and calling FCARD as illustrated. When process control is returned from FCARD, the program verifies that the call was successful (*recode*=0) and continues at the label DISPINST. Under this label, operator instructions are displayed on the \$STDLIST device. If the call to FCARD was unsuccessful (*recode*≠0), then the error message "CAN NOT OPEN FILE — PROGRAM WILL TERMINATE" is displayed and the program goes to the label FINIS and terminates.

Under the label RDYWAIT, a display instructing the operator to press the READY switch is given and the request for a temporary suspension of the program awaiting the depression of the READY switch is made by setting *recode* equal to 4 and calling FCARD as illustrated. The program, upon regaining process control, checks for unsuccessful execution of the request (checks for *recode*≠0). If the execution was unsuccessful, the program goes to the label FINIS and terminates. (NOTE: The program could have branched to an error correcting or displaying instruction set if desired by the programmer). If the execution was successful, the program continues with the next statement which is under the label READ'.

Under the label READ', the program requests the reading of ten cards by setting *recode* equal to 1 and calling FCARD as illustrated. Upon return of the process control from FCARD, the program checks for an unsuccessful execution (*recode*≠0). If the execution was unsuccessful the program goes to the label READ'ERR.

Under the label READ'ERR, the program determines the value of *recode* returned after the read request and initiates corrective action and/or displays an appropriate error message or terminates itself, depending on the value of *recode* detected.

If the execution was successful, the program checks *status* for an empty input or full output hopper condition and if this status condition is detected, the program goes to the label HOPPERS under which corrective steps are initiated. If this status condition is not detected, the program calls a procedure (DISP'COUNT) which displays the number of columns read from the previous card. After the DISP'COUNT procedure is completed, the program goes to the label CLOSE'F.

Under the label CLOSE'F, the program requests that the 7260A be put in the not READY state and that the 7260A/terminal file be closed by setting *recode* equal to 10 and calling FCARD and by setting *recode* equal to 20 and calling FCARD, respectively. In both cases, the value of *recode* returned from FCARD is examined for an indication of successful execution as illustrated.

## ASCII AND COLUMN IMAGE READING FORMATS

In the ASCII mode (also called the Hollerith mode) the OMR recognizes 128 character Hollerith codes and transmits one 7 bit serial ASCII character plus an even parity bit per card column. FCARD packs two ASCII characters (two columns of data) into each buffer word in *bufadr*. The data from the first column of the card is stored in the upper byte of the first word of the buffer, as illustrated below.

1st word	1st column data	2nd column data
2nd word	3rd column data	4th column data

In the column image mode, the OMR transmits a 12-bit data string, representing the twelve rows of one card column. FCARD packs the first 12-bit data string (the first column of data) into the first buffer word in *bufadr*, as illustrated below.

	—	—	—	—	12	11	0	1	2	3	4	5	6	7	8	9	column row no.
Buffer Word	0	0	0	0	X	X	X	X	X	X	X	X	X	X	X	X	data
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	bit no.

X — indicates that bit may be either logical 1 or 0.

```

$CONTROL USLINIT
BEGIN
INTEGER ARRAY BUFADR(0:99);
BYTE ARRAY TOO(0:72);
POINTER HERE;
INTEGER RECODE,A,I;
INTRINSIC QUIT,PRINT;
INTEGER COUNT,FILENUM,STATUS;
INTRINSIC .PRINT'FILE'INFO;

PROCEDURE DISP*COUNT(COUNT);
INTEGER COUNT;

BEGIN
ARRAY OUT(0:11);
BYTE ARRAY ROUT(*)=OUT;
INTRINSIC PRINT,ASCII;
INTEGER A1,A2;

MOVE ROUT:="NO. OF COLUMNS READ= ";
A1:=ASCII(COUNT,10,ROUT(21));
A2:= -21-A1;
PRINT(OUT,A2,X40);

END;

PROCEDURE FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
INTEGER ARRAY BUFADR;
INTEGER RECODE,FILENUM,COUNT,STATUS;
OPTION EXTERNAL;

@HERE:=@TON & LSR(1);
OPENFILE:
<<GET FILE NUMBER FOR LOGICAL DEV EQUAL TO THE TERMINAL>>
RECODE:=0;
FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
IF RECODE =0 THEN GO DISPINST;
MOVE TOO:="CAN NOT OPEN FILE-PROGRAM WILL TERMINATE";
PRINT(HERE,-40,0);
GO FINIS;

DISPINST:
MOVE TOO:=(X15,X12);
PRINT(HERE,-2,0);
MOVE TOO:="SET THE 7260A FOR CLOCK ON DATA.";
PRINT(HERE,-32,0);
MOVE TOO:="PUSH IN THE FULL/HALF SWITCH TO ITS FULL POSITION.";
PRINT(HERE,-49,0);
MOVE TOO:="UNMUTE THE TERMINAL.";
PRINT(HERE,-20,0);
MOVE TOO:="LOAD 30 CLOCK ON DATA CARDS IN THE INPUT HOPPER.";
PRINT(HERE,-48,0);

```

Figure 5-3. FCARD Intrinsic Example (1 of 3)

```

RDYWAIT:
MOVE T00:="NOW, PRESS THE READY SWITCH.";
PRINT(HERE,-20,0);
RECODE:=4;
FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
A:=0;I:=0;
IF RECODE <>0 THEN GO FINIS;

READ*:
DO BEGIN
RECODE:=1;
FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
IF RECODE <> 0 THEN GO READ*ERR;
IF STATUS = %07 THEN GO HOPPERS;
DISP*COUNT(COUNT);
I:=I+1;
END
UNTIL I=10;
GO CLOSE*F;

HOPPERS:
RECODE:=10; <<MAKE BMR NOT READY>>
FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
IF RECODE <> 0 THEN BEGIN
A:=A+1;
IF A<5 THEN GO HOPPERS;
PRINT*FILE*INFO(FILENUM);
QUIT,RECODE);
END;
MOVE T00:="INPUT HOPPER EMPTY OR OUTPUT HOPPER FULL";
PRINT(HERE,-40,0);
MOVE T00:="CORRECT HOPPER CONDITION AND PRESS READY";
PRINT(HERE,-40,0);
IF RECODE <>0 THEN GO FINIS ELSE
GO READ*;

CLOSE*F:
RECODE:=10; <<MAKE CR NOT READY>>
FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
IF RECODE <> 0 THEN BEGIN
I:=I+1;
IF I < 16 THEN GO CLOSE*F;
PRINT*FILE*INFO(FILENUM);
END;
RECODE:=20;
FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
IF RECODE =0 THEN GO FINIS ELSE BEGIN
MOVE T00:="UNABLE TO CLOSE THE TERMINAL FILE";
PRINT(HERE,-33,0);
GO FINIS; END;

READ*ERR:
IF RECODE =8 THEN GO RFRANS;
IF RECODE =4 THEN BEGIN

```

Figure 5-3. FCARD Intrinsic Example (2 of 3)

```

MOVE TOO:="FREAD OR FWRITE ERROR-PROGRAM WILL ABORT";
PRINT(HERE,-40,0);
QUIT(RECODE); END;

IF RECODE #6 THEN BEGIN
MOVE TOO:=":EOJ, :FOD, :DATA, OR :JOB FOUND IN INPUT.";
PRINT(HERE,-42,0);
MOVE TOO:="CHECK CARD VALIDITY-PROGRAM WILL RESTART";
PRINT(HERE,-40,0);
GO DISPINST; END;
MOVE TOO:="UNINTERPRETED ERROR-PROGRAM WILL ABORT";
PRINT(HERE,-37,0);
QUIT(RECODE);

RETRANS:
RECODE:=3;
FCARD(RECODE,EILENUM,BUFADR,COUNT,STATUS);
IF RECODE <> 0 THEN BEGIN
MOVE TOO:="UNSUCCESSFUL RETRANSMIT-PROGRAM WILL ABORT";
PRINT(HERE,-42,0);
QUIT(RECODE); END;

IF STATUS #0 THEN GO READ;
MOVE TOO:="UNSUCCESSFUL RETRANSMIT-PROGRAM WILL ABORT";
PRINT(HERE,-42,0);
QUIT(RECODE);

FINIS:
END.

```

Figure 5-3. FCARD Intrinsic Example (3 of 3)

# RESOURCE MANAGEMENT

SECTION

VI

Within MPE, any element that can be accessed by your program is regarded as a *resource*. Thus, a resource can be an input/output device, file, program, subroutine, procedure, code segment, or the data stack.

Occasionally, you may want to manage a specific resource shared by a particular set of jobs or processes, so that no two of these jobs or processes can use the resource at the same time. To accomplish this type of resource management on either the inter-job (or session) or inter-process level, the jobs or processes involved must mutually cooperate. For example, if job B must not access a particular file when job A is using it, both jobs should include provisions for a hand-shaking arrangement overseen by MPE when these jobs are being executed concurrently. Under this arrangement, when job A has exclusive access to the file and job B attempts to access the same file, this access will be denied. Job B will be suspended until job A releases its exclusive access. Then, job B can resume execution and access the file.

## NOTE

It is important to realize that as long as job B is suspended, it not only cannot access the file — it cannot perform *any* operations.

On either the inter-job or inter-process level, the hand-shaking arrangement is based upon an arbitrary *resource identification number* (RIN) made available to users (at the inter-job level) or assigned to the job (at the inter-process level). Within their jobs (or processes), the cooperating programmers relate a RIN to a particular resource through the structure of the statements making up each job (or process). When a job (or process) seeks exclusive access to a resource, it requests MPE to lock the resource associated with this RIN. This request is granted only if no other job or process has already locked the RIN. Otherwise, the requesting process is suspended until the RIN is released. When it is finished with the resource, the job (or process) requests MPE to unlock the RIN so that other jobs or processes can lock it.

A RIN is *not* a *physical entity*. Furthermore, it is not *logically* assigned to any resource. The association between a RIN and a resource is accomplished only by the structure of the statements within the job or process using the RIN. The resource identification *number* is always known to MPE, but its meaning (the resource with which it is associated) is not. For this reason, all cooperating programs must specify what RIN is associated with what resource through their statements.

Processes run by users having only the *Standard MPE Capabilities* can lock only one RIN at a time. But processes run by users having the *Multiple RIN Optional Capability* can lock more than one RIN at a time. In doing so, however, such users must be careful to avoid deadlocking, where two or more suspended processes cannot be resumed because they are mutually blocked.



## INTER-JOB LEVEL (GLOBAL) RIN'S

The RIN's used at the inter-job level are called *global* RIN's. Global RIN's are used to exclude simultaneous access of a resource by two or more jobs. Each global RIN is a positive integer unique within MPE. Global RIN's are *acquired* and *released* through MPE commands, and *locked* and *unlocked* through MPE intrinsics.

### ACQUIRING GLOBAL RIN'S

Before any users can mutually engage in resource management through a RIN, one of these users must request the RIN and assign it a RIN password that enables all who know the password to lock the RIN. This is done by entering the :GETRIN command:

```
:GETRIN rinpassword
```

where

*rinpassword* is a password required in the intrinsic that locks the RIN. It is a string of up to eight alphanumeric characters beginning with a letter. (Required parameter.)

The :GETRIN command is typically entered during a session. As a result of the command, MPE makes a RIN available for use and displays the RIN number in this format:

```
RIN: rin
```

where

*rin* is the RIN number.

The user who entered the :GETRIN command can use the RIN number to lock and unlock the RIN in the current session, or in future jobs and sessions. The RIN number and password also are passed on to other users to permit them to lock and unlock the RIN in their jobs and sessions. All users pass the RIN number to the intrinsics that lock and unlock the RIN, as a reference parameter in the intrinsic calls. These users can continue to use the RIN until the user who issued the :GETRIN command for this RIN releases the RIN.

#### NOTE

MPE regards the user who issued the :GETRIN command as the *owner* of the RIN assigned. This means that only this user may release the RIN.

The total number of RIN's that MPE can allocate is specified when the system is configured, and in no case can exceed 1024.

See the *MPE Commands Reference Manual* for a further discussion of the :GETRIN command.

## RELEASING GLOBAL RIN'S

The owner of a global RIN (the user who issued the :GETRIN command to acquire the RIN) can de-allocate the RIN, returning it to the RIN pool managed by MPE. Only the *owner* can de-allocate the RIN.

The RIN is de-allocated with the :FREERIN command

```
:FREERIN rin
```

where

*rin* is the number of the RIN to be de-allocated. (Required parameter.)

See the *MPE Commands Reference Manual* for a further discussion of the :FREERIN command.

## LOCKING AND UNLOCKING GLOBAL RIN'S

Any global RIN assigned to a group of users can be locked by one job at a time when the LOCKGLORIN intrinsic. Once a RIN is locked, any other jobs that attempt to lock this RIN are suspended.

In order to lock a global RIN, you must know both the RIN number returned by MPE when the RIN was acquired with the :GETRIN command, and the password which was specified in the *rinpassword* parameter of the :GETRIN command. If you are a user with only the Standard MPE Capability, you can lock only one global RIN at a time.

The LOCKGLORIN intrinsic is useful in applications where locking an entire file is not desirable because it may inconvenience other users. For example, if several users are trying to access and update a large file simultaneously, any one user who succeeds in locking the file suspends the other users' processes until the file is unlocked. The LOCKGLORIN intrinsic, however, can be used to lock a *portion* of such a file so that the chance of inconveniencing the other users is lessened.

Figure 6-1 contains a program which uses the LOCKGLORIN and UNLOCKGLORIN intrinsics. The program allows a user to lock four records, as a RIN, in a file so that a record can be updated without any chance of some other user updating the same record simultaneously. Additionally, the other users are not suspended when attempting to access and update records elsewhere in the file.

The file used in the example (see below) contains 20 records and therefore five contiguous RIN's have to be acquired (there are four records per RIN) before the program is run. This is accomplished by entering :GETRIN commands as follows:

```
:GETRIN BOOKRIN
```

where BOOKRIN is specified as the *rinpassword* parameter. BOOKRIN is the password which is used in the program to lock the RIN (see statements 6 and 36 in figure 6-1).

```

00001000 00000 0  SCONTROL USLIMIT
00002000 00000 0  BEGIN
00003000 00000 1    BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1    BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1    BYTE ARRAY NAME(0:8):="BOOKFILE ";
00006000 00006 1    BYTE ARRAY PASSWD(0:7):="BOOKRIN ";
00007000 00005 1    INTEGER IN,OUT,BOOK,LGTH,ACCNO,RIN;
00008000 00005 1    LOGICAL DUMMY,COND:=TRUE;
00009000 00005 1    ARRAY BUFR(0:35);
00010000 00005 1    BYTE ARRAY BBUFR(*)=BUFR;
00011000 00005 1    ARRAY HEAD(0:13):="LIBRARY INFORMATION PROGRAM.";
00012000 00016 1    ARRAY REQUEST(0:7):=%6412,"ACCESSION NO: ";
00013000 00010 1    ARRAY CHANGE(0:9):="      NEW LOCATION: ";
00014000 00012 1    EQUATE RINBASE=2, RECD$PER=RIN=4, MAXRIN=6;
00015000 00012 1    DEFINE CCL =IF < THEN QUIT#,
00016000 00012 1    CCNE=IF <> THEN QUIT#;
00017000 00012 1
00018000 00012 1    INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,FREADDIR,FWRITEDIR,
00019000 00012 1    LOCKGLORIN,UNLOCKGLORIN,QUIT,BINARY;
00020000 00012 1
00021000 00012 1    <<END OF DECLARATIONS>>
00022000 00012 1
00023000 00012 1    IN:=FOPEN(INPUT,%45); CCL(1);          <<$STDIN>>
00024000 00012 1    OUT:=FOPEN(OUTPUT,%414); CCL(2);        <<$STDLIST>>
00025000 00024 1    BOOK:=FOPEN(NAME,%5,%304); CCL(3);     <<OLD DISC FILE>>
00026000 00037 1    FWRITE(OUT,HEAD,14,0); CCNE(4);        <<PROGRAM ID>>
00027000 00047 1  LOOP:
00028000 00047 1    FWRITE(OUT,REQUEST,8,%320); CCNE(5);    <<REQST BOOK NUMBR>>
00029000 00057 1    LGTH:=FREAD(IN,BUFR,-10); CCNE(6);    <<INPUT NUMBER>>
00030000 00070 1    IF LGTH=0 THEN GO EXIT;              <<NO INPUT-EXIT>>
00031000 00073 1    ACCNO:=BINARY(BBUFR,LGTH);          <<CONVERT NUMBER>>
00032000 00100 1    IF <> THEN GO LOOP;                  <<IF BAD TRY AGAIN>>
00033000 00101 1
00034000 00101 1    RIN:=RINBASE+(ACCNO/RECD$PER*RIN);    <<COMPUTE RIN NO.>>
00035000 00105 1    IF NOT(RINBASE<=RIN<=MAXRIN) THEN GO LOOP; <<BOUNDS CHECK RIN>>
00036000 00120 1    LOCKGLORIN(RIN,COND,PASSWD);        <<LOCK FILE SUBSET>>
00037000 00124 1
00038000 00124 1    FREADDIR(BOOK,BUFR,36,DOUBLE(ACCNO)); CCL(7); <<READ BOOK DATA>>
00039000 00136 1    IF > THEN GO AGAIN;                <<EOF - TRY AGAIN>>
00040000 00137 1    FWRITE(OUT,BUFR,36,0); CCNE(8);        <<DISPLAY DATA>>
00041000 00147 1    FWRITE(OUT,CHANGE,10,%320); CCNE(9);    <<REQST A CHANGE>>
00042000 00157 1
00043000 00157 1    BUFR(19):=" ";
00044000 00162 1    MOVE BUFR(20):=BUFR(19),(16);        <<BLANK OLD LOCN>>
00045000 00170 1    LGTH:=FREAD(IN,BUFR(19),17); CCNE(10);    <<READ NEW LOCN>>
00046000 00202 1    IF LGTH>0 THEN                      <<NEW LOCN ENTERED>>
00047000 00205 1    BEGIN
00048000 00205 2    FWRITEDIR(BOOK,BUFR,36,DOUBLE(ACCNO)); <<MODIFY THE FILE>>
00049000 00214 2    CCNE(11);          <<CHECK FOR ERROR>>
00050000 00217 2    END;
00051000 00217 1    FCONTROL(BOOK,2,DUMMY); CCL(12);    <<FORCE RECD POST>>
00052000 00226 1  AGAIN:
00053000 00226 1    UNLOCKGLORIN(RIN); CCNE(13);    <<UNLOCK SUBSET>>
00054000 00233 1    GO LOOP;          <<CONTINUE>>
00055000 00235 1  EXIT:END.
PRIMARY DB STORAGE=%021;  SECONDARY DB STORAGE=%00124
NO, ERRORS=000;          NO, WARNINGS=000
PROCESSOR TIME=0:00:03;  ELAPSED TIME=0:00:10

```

Figure 6-1. Using the LOCKGLORIN and UNLOCKGLORIN Intrinsics

TITLE: THE BORROWERS	LOCN: AVAILABLE
TITLE: ALICE IN WONDERLAND	LOCN: AVAILABLE
TITLE: PETER PAN	LOCN: AVAILABLE
TITLE: JUNGLE BOOK	LOCN: AVAILABLE
TITLE: MARY POPPINS	LOCN: AVAILABLE
TITLE: TOM SAWYER	LOCN: AVAILABLE
TITLE: TREASURE ISLAND	LOCN: AVAILABLE
TITLE: A CHRISTMAS CAROL	LOCN: AVAILABLE
TITLE: HOUSE AT POOH CORNER	LOCN: AVAILABLE
TITLE: THE WIZARD OF OZ	LOCN: AVAILABLE
TITLE: SLEEPING BEAUTY	LOCN: AVAILABLE
TITLE: TALES OF MOTHER GOOSE	LOCN: AVAILABLE
TITLE: AESOP'S FABLES	LOCN: AVAILABLE
TITLE: KIDNAPPED	LOCN: AVAILABLE
TITLE: OLIVER TWIST	LOCN: AVAILABLE
TITLE: DR. DOLITTLE	LOCN: AVAILABLE
TITLE: WHEN WE WERE VERY YOUNG	LOCN: AVAILABLE
TITLE: H.M.S. PINAFORE	LOCN: AVAILABLE
TITLE: WORLD BOOK ENCYCLOPEDIA	LOCN: AVAILABLE
TITLE: COLLEGIATE DICTIONARY	LOCN: AVAILABLE

The program in figure 6-1 establishes the RIN number limits 2 and 6 (see statement number 14), thus using only RIN numbers 2, 3, 4, 5, and 6. MPE returns the RIN number assigned each time the :GETRIN command is entered. Because MPE does not always assign RIN numbers in sequence, however, it may be necessary to enter more than five :GETRIN commands in order to acquire the five contiguous RIN's 2, 3, 4, 5, and 6. Extra RIN's can be released with the :FREERIN command.

The statements

```
FWRITE(OUT,REQUEST,8,%320); CCNE(5);
```

request a book number from the user and perform a condition code check. Note that in statement number 16, CCNE has been defined as

```
IF <> THEN QUIT#;
```

This eliminates the need to repeat the entire statement at every point in the program where such a condition code check is required. Instead, the statement CCNE and an arbitrary number, (5) in this case, can be used.

The book number is read with the statement

```
LGTH:=FREAD(IN,BUFR,-10);
```

and converted to a binary value with the statement

```
ACCNO:=BINARY(BBUFR,LGTH);
```

The RIN number to be locked is computed with the statement

```
RIN:=RINBASE+(ACCNO/RECDS'PER'RIN);
```

RINBASE and RECDSPER'RIN have been equated to 2 and 4, respectively (see statement number 14). Thus, if book number 3 is entered by the user, the RIN number to be locked would be computed as RIN number 2, as follows:

$$\begin{aligned} \text{RIN} &= 2 + (3/4) \\ &= 2 + 0 \text{ (integer division)} \end{aligned}$$

The record specified by the book number is displayed for the user and the change ("NEW LOCATION: ") is requested. The existing location information is filled with blanks with the statements

```
BUFR(19):=" ";  
MOVE BUFR(20):=BUFR(19),(16);
```

The new location is entered and read with the statement

```
LGTH:=FREAD(IN,BUFR(19),(17);
```

and the record is updated with the statement

```
FWRITEDIR(BOOK,BUFR,36,DOUBLE(ACCNO));
```

The statement

```
FCONTROL(BOOK,2,DUMMY);
```

is used in case the file which has been opened is a buffered file. This statement insures that the process' buffers are posted to the disc before the RIN is unlocked.

Note that in a program of this kind, it is important that the number of records per block and the number of records per RIN are the same. The RIN must contain a complete block of records.

The statement

```
UNLOCKGLORIN(RIN);
```

unlocks the RIN before the loop is repeated. When the user enters a new book number, a new RIN number will be computed and that RIN number will be locked.

When a carriage return is entered, signifying no input, the program terminates.

The results of running the program and the updated condition of the library file are shown below.

## **INTER-PROCESS (LOCAL) LEVEL RIN'S**

The RIN's used at the inter-process level are called *local* RIN's. These RIN's are used to exclude simultaneous access of a resource by two or more processes within the same job. Each RIN number is a positive integer that is significant only with respect to different processes within that job.



Local RIN's are assigned, managed, and released with the GETLOCRIN, LOCKLOCRIN, and FREELOCRIN intrinsics.

### ACQUIRING LOCAL RIN'S

Just as global RIN's must be acquired by users before they can be used in jobs, local RIN's must be acquired by a job before they can be used by processes within the job. This is done with the GETLOCRIN intrinsic. For example, the intrinsic call below would acquire six local RIN's:

```
GETLOCRIN(6);
```

### LOCKING AND UNLOCKING LOCAL RIN'S

Any local RIN assigned to a job can be locked, by one process at a time, by issuing the LOCKLOCRIN intrinsic call within that process. When this is done, other processes within the job that attempt to lock this RIN are suspended until the locked RIN is released.

For example, to lock RIN number 6 (acquired with the GETLOCRIN intrinsic) unconditionally, the following intrinsic call could be used:

```
LOCKLOCRIN(6,COND);
```

The logical word COND = TRUE for unconditional locking. If COND = FALSE, locking will take place only if the RIN is immediately available.

To unlock this same RIN, the following UNLOCKLOCRIN intrinsic call could be used:

```
UNLOCKLOCRIN(6);
```

The above call makes RIN number 6 available for locking by other processes in the job. The highest priority process suspended because this RIN was locked is now activated.

To illustrate how the LOCKLOCRIN and UNLOCKLOCRIN intrinsic calls are used, consider two processes (a father process and its son) within a job:

FATHER PROCESS	SON PROCESS
.	.
.	.
.	.
LP:=FOPEN (LIN, . . .);	LP:=FOPEN (LIN, . . .);
.	.
.	.
.	.
GETLOCRIN (3);	LOCKLOCRIN (1, TRUEVAL);
FWRITE (LP, . . .);	FWRITE (LP, . . .);
LOCKLOCRIN (1, TRUEVAL);	.
CREATE (DESCEND, . . .).	.

```
FWRITE (LP, . . .);  
.  
.  
.  
UNLOCKLOCIN (1);  
.  
.  
.
```

```
FWRITE (LP, . . .);  
.  
.  
.  
UNLOCKLOCIN (1);  
.  
.  
.
```

Suppose that the father process and its son wanted to use RIN (1) to manage a line printer (designated by LP) so that the son process could not use the printer at any time that it was being used by the father process. This could be done as shown in the above coding. When the father process first references LP, the son process is not yet created and the printer need not be locked. However, just prior to creating the son, the father process locks the RIN covering the printer. The father issues all of its print requests before unlocking the printer. Before the son process accesses the printer, it tries to lock it, fails, and is suspended. As soon as the father unlocks the printer, the son process locks it, and issues print requests.

### **FREEING LOCAL RIN'S**

To free all local RIN's currently reserved for your job, the FREELOCIN intrinsic is called, as follows:

```
FREELOCIN;
```





# PROCESS-HANDLING CAPABILITY

SECTION

VII

All user and system programs under MPE are run on the basis of *processes* — which are the basic executable entities in the operating system. Processes are invisible to a programmer with *standard capabilities* who accesses MPE. This programmer has no control over processes or their structure; for him, MPE automatically creates, handles, and deletes all processes. Users with certain *optional capabilities*, however, can interact with processes directly. One of these optional capabilities is the *Process-Handling Optional Capability*, discussed in this section.

The Process-Handling Capability, assigned and used independently of the other optional capabilities, allows you to

- Create and delete processes.
- Activate and suspend processes.
- Manage communications between processes.
- Change the scheduling of processes.
- Obtain information about existing processes.

These operations can be very useful to you. For instance, they allow you to have several independent processes running concurrently on your behalf, all communicating with one another.

## PROCESSES

A process is an independent entity that can be run within MPE: processes are run on behalf of users and on behalf of the operating system. Many processes can be running concurrently. The design of MPE is process-oriented: the system deals exclusively with processes (except for interrupt routines and some very central and specialized system functions).

A process consists of a private data area (the stack) used only by this process, a Process Control Block (PCB) that defines the process, and an instruction in a code segment that the process is to execute. Note that code segments are used by processes, not owned by them, and may, therefore, be shared by many processes.

When a user enters MPE, a process is created for him. This process is called a Job Main Process (JMP) in batch mode or a Session Main Process (SMP) in time-share mode. The process is linked into the Command Interpreter which then proceeds to handle user commands.

Every process known to MPE is identified by a number called the Process Identification Number (PIN). Most control in MPE is carried out at the process level. A process can run any kind of code (programs, procedures, private code, sharable code, and so forth) and one of the main elements needed to establish a new process is a starting address (that is a *program label*). From this address on, the life of the process follows the sequence of the code until its deletion.

The *Progenitor* is the first process established during the initialization phase of MPE. It is the responsibility of the Progenitor, using a set of configuration data specified at system configuration time, to create its *son* processes. These processes are defined as *system processes* and are used to perform parallel functions on behalf of the system. Such processes may include I/O processes, etc., and in particular the *User Controller Process* (UCOP). All these processes may, if required, have their own structure of descendents.

Whereas the Progenitor is the ancestor of all processes in MPE, including system processes, the User Controller Process is the ancestor of all User Processes currently in existence. The UCOP is thereby the root of the user process Tree Structure. The *sons* of the UCOP are called (User) main processes.

The father/son relationship between processes is used mainly to maintain control from top to bottom everywhere in the structure. Roughly speaking, a father is always held “responsible” for what happens to its son: creation, deletion and other special actions.

## ORGANIZATION OF USER PROCESSES

When you log on yourself, a main process is created for you by UCOP. According to the mode of access, the main process can be one of the two types:

- Job Main Process (JMP)
- Session Main Process (SMP)

Such a distinction results from the different kinds of control that the system provides for those two separate entities: job is associated with a batch type of access, while session is for interactive access.

As soon as a given signal is received by UCOP, a JMP or SMP is created (depending upon the origin of the signal). The starting address of the JMP or SMP is the Command Interpreter and once the user is validated, the main process is free to recognize any command.

## PROCESS SUBSTATES

During its life span (i.e., between its creation and its deletion), a process finds itself in different substates according to its past and present history as well as its present requirements. Only two of these may be controlled by users: *active substate* and *suspended substate*.

An active process is run by the CPU until it suspends itself, terminates, or is killed.

A suspended process is not run by the CPU as long as it stays in this substate. In other words, a suspended process is waiting for some kind of a signal which will activate it. When it suspends itself, a process may specify the origin of its next activation.

You can control the termination of one of your processes. The termination destroys the process and all its descendents and resets the links of the remaining processes for the session or job.

## PROCESS TO PROCESS COMMUNICATION

MPE provides a means for processes to communicate between themselves.

The sending or receiving of information is restricted to either an upward or downward path; thus such communication is allowed only between father and son, and only one transfer is allowed in either direction at any given time.

This method results from the fact that the father is solely responsible for both the existence and actions of his sons. The father created his sons and knows their identification in the process structure; he can, therefore, reference them at any time. The sons, however, only know their father by the default identification "father".

## CREATING AND ACTIVATING PROCESSES

From within any running process, you can programmatically request the creation of a son process with the CREATE intrinsic. The CREATE intrinsic loads the program to be run by the new process into virtual memory, creates the new process as the son of the calling process, initializes its data stack, schedules the process, and returns its Process Identification Number (PIN) to the calling process.

Once a process is created, it must be activated with the ACTIVATE intrinsic in order to run. When a process is activated, it may suspend the process that activated it, then run until it is suspended or deleted. A newly-created process can be activated only by its father. A father process that has been suspended when a son process was activated can be reactivated automatically when the son process' execution ends, if a bit has been set in the *flags* parameter of the CREATE intrinsic. A process that has been suspended with the SUSPEND intrinsic (see page 7-8) can be reactivated by its father or any of its sons, as specified in the *susp* parameter of the SUSPEND intrinsic.

Figure 7-1 contains a program which illustrates the CREATE and ACTIVATE intrinsics.

The statement

```
FWRITE(OUT,REQST,9,%320);
```

requests the user to enter the program file name which is to be created and activated.

The statement

```
LGTH:=FREAD(IN,NAME,-26);
```

reads the name input by the user on \$STDIN and stores the name in the array NAME. In order to be used in the CREATE intrinsic, the string in the array NAME must be specified in a byte array; thus the byte array BNAME is equivalenced to NAME in statement number 8 in the program. Additionally, the string must be terminated by a blank and the statement

```
BNAME(LGTH):=%40;
```

enters the ASCII code for a blank character to the end of the string in BNAME.

Next, the program displays the message

```
CREATE PROCESS
```

and calls the CREATE intrinsic with the statement

```
CREATE(BNAME,,PIN,,1);
```

```

00001000 00000 0 $CONTROL USLIMIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1 BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1 INTEGER IN,OUT,LGTH,PIN;
00006000 00005 1 ARRAY REQST(0:8):=%6412,"PROGRAM FILE = ";
00007000 00011 1 ARRAY NAME(0:13);
00008000 00011 1 BYTE ARRAY BNAME(*)=NAME;
00009000 00011 1 ARRAY CRMSG(0:6):="CREATE PROCESS";
00010000 00007 1 ARRAY ACTMSG(0:7):="ACTIVATE PROCESS";
00011000 00010 1 DEFINE CCL=IF < THEN QUIT#,
00012000 00010 1 CCG=IF > THEN QUIT#,
00013000 00010 1 CCNE=IF <> THEN QUIT#;
00014000 00010 1
00015000 00010 1 INTRINSIC FOPEN,FREAD,FWRITE,CREATE,ACTIVATE,QUIT;
00016000 00010 1
00017000 00010 1 <<END OF DECLARATIONS>>
00018000 00010 1
00019000 00010 1 IN:=FOPEN(INPUT,%45); <<$STDIN>>
00020000 00007 1 CCL(1); <<CHECK FOR ERROR>>
00021000 00012 1 OUT:=FOPEN(OUTPUT,%414,1); <<$STDLIST>>
00022000 00022 1 CCL(2); <<CHECK FOR ERROR>>
00023000 00025 1
00024000 00025 1 NEXT:
00025000 00025 1 FWRITE(OUT,REQST,9,%320); <<REQUEST PROGRAM FILE NAME>>
00026000 00032 1 CCNE(3); <<CHECK FOR ERROR>>
00027000 00035 1 LGTH:=FREAD(IN,NAME,-26); <<INPUT FILE NAME>>
00028000 00043 1 CCNE(4); <<CHECK FOR ERROR>>
00029000 00046 1 IF LGTH=0 THEN GO EXIT; <<IF NO NAME - EXIT>>
00030000 00053 1 BNAME(LGTH):=%40; <<SET IN TRAILING BLANK>>
00031000 00056 1
00032000 00056 1 FWRITE(OUT,CRMSG,7,0); <<CREATE MESSAGE>>
00033000 00063 1 CCNE(5); <<CHECK FOR ERROR>>
00034000 00066 1 CREATE(BNAME,,PIN,,1); <<CREATE PROCESS>>
00035000 00076 1 CCL(6); <<CHECK FOR ERROR>>
00036000 00101 1
00037000 00101 1 FWRITE(OUT,ACTMSG,8,0); <<ACTIVATE MESSAGE>>
00038000 00106 1 CCNE(7); <<CHECK FOR ERROR>>
00039000 00111 1 ACTIVATE(PIN,2); <<ACTIVATE PROCESS>>
00040000 00115 1 CCL(8); CCG(9); <<CHECK FOR ERROR>>
00041000 00123 1 GO NEXT; <<CONTINUE OPERATIONS>>
00042000 00130 1 EXIT:END.
PRIMARY DB STORAGE=%013; SECONDARY DB STORAGE=%00055
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:04; ELAPSED TIME=0:00:51

```

Figure 7-1. Using the CREATE and ACTIVATE Intrinsics

The following parameters were specified in the above intrinsic call:

- programe* Specified by BNAME, which contains the name entered by the user.
- entryname* Omitted. The primary entry point of the created process is specified by default.
- pin* The Process Identification Number (PIN), to be used by the ACTIVATE intrinsic, is returned to the word PIN.
- param* Omitted. A word filled with zeros is specified by default.
- flags* 1, which specifies that this (the father) process will be reactivated automatically by MPE when the created process' execution ends (bit 15 = 1).

All other parameters are omitted in the CREATE intrinsic call.

The statement

```
FWRITE(OUT, ACTMSG,8,0);
```

displays the message

```
ACTIVATE PROCESS
```



and the statement

```
ACTIVATE(PIN,2);
```

calls the ACTIVATE intrinsic to activate the process. The following parameters were specified:

- pin* Specified by PIN, which contains the Process Identification Number of the process to be activated, as returned to the CREATE intrinsic by the system.
- susp* 2. When *susp* is specified, the calling process is to be suspended when the called process is activated. When 2 (bit 14 = 1) is specified, as in this call, the suspended calling process expects to be reactivated automatically by MPE when this son process ends execution. If *susp* was not specified, the calling (father) process will not be suspended when the called process is activated.

Shown below is an example of running the program (named PROC) listed in figure 7-1.

```
:RUN PROC

PROGRAM FILE = SPL.PUB.SYS
CREATE PROCESS
ACTIVATE PROCESS

PAGE 0001    HP32100A.05.1

>CONTROL USLINIT
>BEGIN
> ARRAY MSG(0:12):="* TEST PROCESS EXECUTING *";
> INTRINSIC PRINT;
> PRINT(MSG,13,0);
>END.
PRIMARY DB STORAGE=%001;    SECONDARY DB STORAGE=%00015
NO. ERRORS=000;            NO. WARNINGS=000
PROCESSOR TIME=0:00:02;    ELAPSED TIME=0:13:20
```

```

PROGRAM FILE = SEGDVR.PUB.SYS
CREATE PROCESS
ACTIVATE PROCESS
SEGMENTER SUBSYSTEM (C.0)
-USL $OLDPASS
-PREPARE $NEWPASS
-EXIT

PROGRAM FILE = $OLDPASS
CREATE PROCESS
ACTIVATE PROCESS
* TEST PROCESS EXECUTING *

PROGRAM FILE = return

END OF PROGRAM
:
```

When SPL.PUB.SYS is entered in response to the PROGRAM FILE = request, the program displays

```

CREATE PROCESS

ACTIVATE PROCESS
```

then suspends itself and the SPL compiler subsystem is accessed. (This process has been created and activated because of the SPL.PUB.SYS response by the user.)

A short program is entered from the terminal and the SPL compiler is exited, reactivating PROC at the statement following the ACTIVATE intrinsic call, and causing the PROGRAM FILE = message to be displayed again.

The response

```

SEGDVR.PUB.SYS
```

causes PROC to create and activate the Segmenter Driver (a programmatic entry point to the Segmenter subsystem). The Segmenter displays

```

SEGMENTER SUBSYSTEM (C.0)
```

and a prompt character (-).

The small program written in SPL and compiled into the USL file \$OLDPASS (the default USL file since a *USLfile* parameter was not included in the SEGDVR.PUB.SYS response) is identified with the

```

-USL $OLDPASS
```

Segmenter command.

The next command

```

-PREPARE $NEWPASS
```

prepares the SPL program and the Segmenter is exited with the

```

-EXIT
```

command.

Once again, PROC is reactivated and requests a program file to be created and activated. The response (\$OLDPASS) causes the compiled and prepared program written in SPL to be created and activated.

This program executes, displays

```
*TEST PROCESS EXECUTING*
```

then ends execution, reactivating PROC.

A carriage return, signifying no input, is entered in response to the PROGRAM FILE = request and the program terminates.

The example below uses PROC to create and activate a duplicate of itself.

```
:RUN PROC

PROGRAM FILE = PROC
CREATE PROCESS
ACTIVATE PROCESS

PROGRAM FILE = PROC
CREATE PROCESS
ACTIVATE PROCESS

PROGRAM FILE = return1

PROGRAM FILE = return2

PROGRAM FILE = return3

END OF PROGRAM
:
```

When PROC is entered in response to the PROGRAM FILE = request, the calling process (PROC) creates a duplicate of itself and activates this process (for clarity, call this PROC1). This process executes and requests a file name. The user enters PROC again, causing yet another duplicate (call this one PROC2) to be created and activated. At this point, PROC is suspended: it has created and activated a duplicate process. The duplicate process (PROC1) has, in turn, created and activated a duplicate of itself (PROC2). Thus, it also is suspended. The third process (PROC2) executes and displays

```
PROGRAM FILE =
```

A carriage return (see *return1* in the example) causes this process to stop executing and control returns to PROC1.



PROC1 displays

```
PROGRAM FILE =
```

Again, a carriage return (*return2* in the example) causes this process to stop executing and control returns to PROC.

PROGRAM FILE = is displayed once more, this time by the original process. Carriage *return3* causes PROC to stop executing and control returns to the session main process, which displays

```
END OF PROGRAM
```

## SUSPENDING PROCESSES

A process can suspend itself with the SUSPEND intrinsic. When this is done, the process relinquishes its access to the central processor until reactivated by an ACTIVATE intrinsic call. When it suspends itself, the process must specify the anticipated source of this ACTIVATE call (its father or son process). When the process is reactivated, it begins execution with the instruction immediately following the SUSPEND intrinsic call. The SUSPEND intrinsic also can release a local Resource Identification Number (RIN) when the process is suspended by specifying the RIN number as a parameter in the intrinsic call.

The intrinsic call

```
SUSPEND(3,RINNUM);
```

would cause the process to suspend itself and release the local RIN specified by RINNUM. The parameter 3 (bits 14 and 15 = 11) specifies that the process expects to be reactivated by either its father or one of its sons.

## DELETING PROCESSES

A process can request the deletion of itself with the TERMINATE intrinsic or the deletion of any of its sons with the KILL intrinsic. When this is done, all code and data segments in the process, and all resources owned by the process, are released; all temporary files opened by the process are closed; and finally, the Process Identification Number (PIN) is released. When a process is deleted, MPE also automatically deletes all descendents of that process, as shown in figure 7-2. Within a process tree structure, the newest generations are deleted first. Within each generation, processes are deleted in the order of their creation.

In a job or session main process, the TERMINATE intrinsic is invoked automatically by detection of an end-of-job/session condition. This intrinsic removes the job or session from the system.

The form of the TERMINATE intrinsic call is

```
TERMINATE;
```

The form of the KILL intrinsic call is

```
KILL(PIN);
```

Where PIN contains the Process Identification Number of the son process to be deleted.

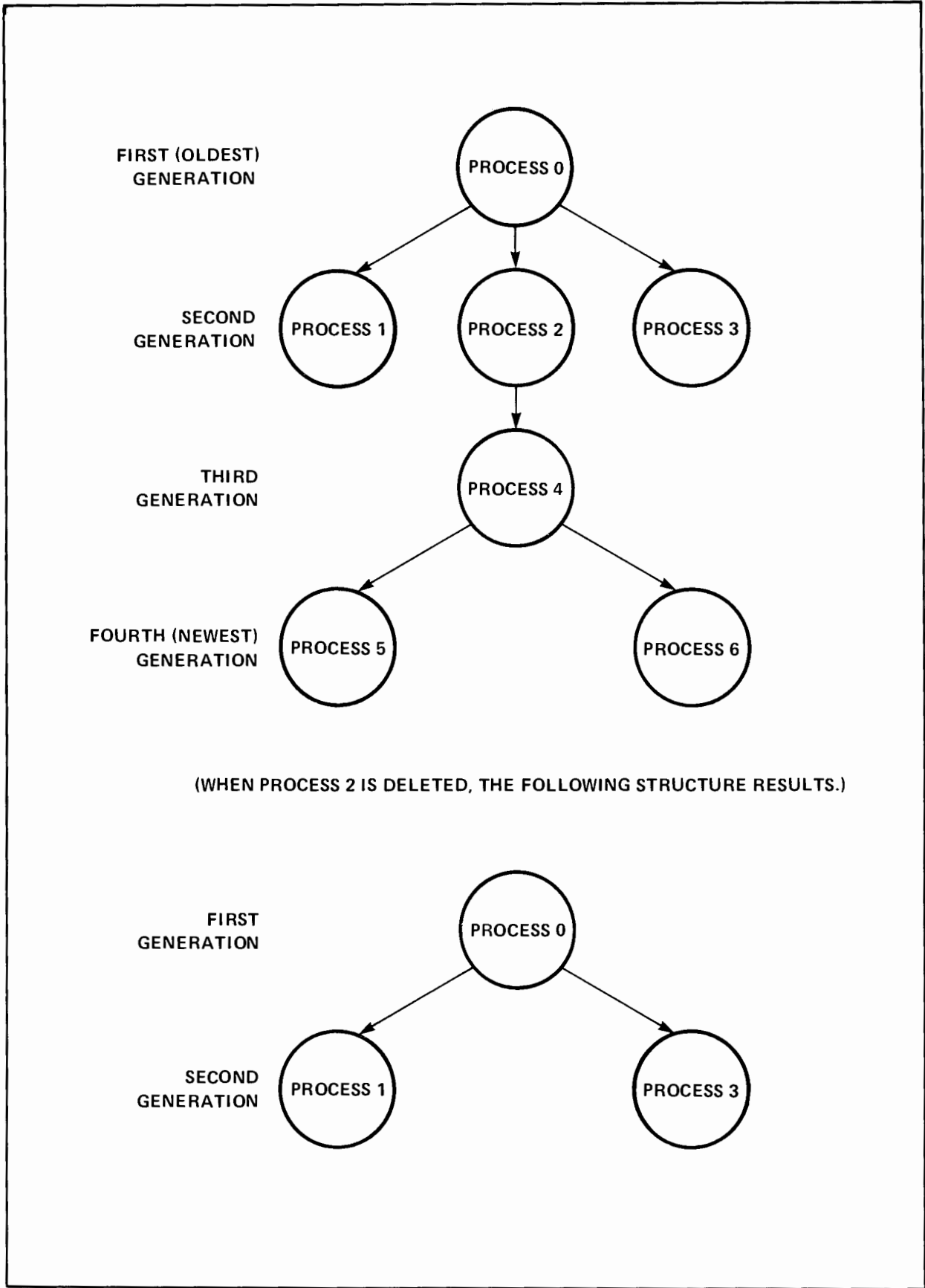


Figure 7-2. Process Deletion

## INTERPROCESS COMMUNICATION

You can direct the communication of information between processes. This information transfer, however, is restricted to upward or downward paths through the process tree structure, so that any process can communicate only with its father or sons. Between any father/son pair, only one such transfer is allowed at any particular time.

Information transferred between processes is referred to as *mail*. It is sent from one process to another through an intermediate storage area called a *mailbox*. At any given time, a mailbox can contain only one item of mail (a *message*). For any process, there are two sets of mailboxes:

- The mailbox used for communication between the process and its father. Each process has one of these.
- The set of mailboxes used for communication between the process and its sons. Each process has one of these mailboxes for each of its sons.

Even though there are two sets of mailboxes, between any two processes there is only one mailbox.

The transfer of mail is based upon a transaction between the sending and receiving processes that involves the following steps:

1. Optionally, the sending process tests the mailbox to determine its status (whether it is empty, contains a message, or is being used by the receiving process).
2. The sending process transmits the mail to the mailbox. The message transferred is a word array in the sending process' stack, defined by a starting location and word count. The smallest message allowed is a single word. MPE automatically performs a bounds check that insures that the array specified actually falls within the limits of the process' stack.
3. The receiving process optionally tests the mailbox to determine its status.
4. If the mailbox contains a message, the receiving process collects this mail. If the mail is not collected, it is overwritten by additional mail from the sending process. When the mail is collected, another bounds check is performed to validate the address given for the stack of the receiving process.

### TESTING MAILBOX STATUS

A process can determine the status of the mailbox used by its father or by a son with the MAIL intrinsic. If the mailbox contains mail that is awaiting collection by this process, the length of the message, in words, is returned to the calling process. This enables the calling process to initialize its stack in preparation for receipt of the message.

For example, to test the status of the mailbox associated with one of its son processes, the following intrinsic call could be used:

```
STATCOUNT:=MAIL(SONPIN,MCOUNT);
```

The Process Identification Number (PIN) of the son is stored in the word SONPIN. An integer count signifying the length, in words, of the incoming message will be returned to the word MCOUNT. The status returned to STATCOUNT will be one of the following values:

Status	Meaning
0	The mailbox is empty.
1	The mailbox contains previous <i>outgoing</i> mail from this calling process that has not yet been collected by the destination process.
2	The mailbox contains incoming mail awaiting collection by this calling process.
3	An error occurred because an invalid PIN was specified or a bounds check failed.
4	The mailbox is temporarily inaccessible because other intrinsics are using it in the preparation or analysis of mail.

## SENDING MAIL

A process sends mail to its father or sons with the SENDMAIL intrinsic. If the mailbox for the receiving process contains a message sent previously by the calling process but not collected by the receiving process, the action taken depends on the *waitflag* parameter specified in SENDMAIL. If the mail is being used currently by other intrinsics, the SENDMAIL intrinsic waits until the mailbox is free and then sends the mail.

For example, to send mail to its father, the following intrinsic call could be used:

```
STAT:=SENDMAIL(0,3,LOCAT,WAITSTAT);
```

The parameters specified are

<i>pin</i>	0, specifying that the mail is to be sent to the father process.
<i>count</i>	3, specifying that the length of the message is 3 words.
<i>locat</i>	LOCAT, an array in the stack containing the message to be sent.
<i>waitflag</i>	WAITSTAT, a logical word. If WAITSTAT = FALSE (bit 15 = 0), any mail sent previously will be overwritten. If WAITSTAT = TRUE (bit 15 = 1), the intrinsic will wait until the receiving process collects the previous mail before sending the current mail.

The status returned to STAT is one of the following values:

Status	Meaning
0	The mail was transmitted successfully. The mailbox contained no previous mail.
1	The mail was transmitted successfully. The mailbox contained previously-sent mail that was overwritten by the new mail, or contained previous incoming/outgoing mail that was cleared.
2	The mail was not transmitted successfully because the mailbox contained incoming mail to be collected by the sending process (regardless of the <i>waitflag</i> parameter setting).
3	An error occurred because an illegal PIN was specified, or a bounds check failed.
4	An illegal wait request would have produced a deadlock.
5	The request was rejected because the count specified in the <i>count</i> parameter exceeded the mailbox size allowed by the system.
6	The request was rejected because storage resources for the mail data segment were not available.

## RECEIVING (COLLECTING) MAIL

A process collects mail transmitted from its father or a son with the RECEIVEMAIL intrinsic. If the mailbox for the receiving process is empty, the action taken depends on the *waitflag* parameter specified in RECEIVEMAIL. If the mailbox is being used currently by other intrinsics, the RECEIVEMAIL intrinsic waits until the mailbox is free before accessing it.

To collect a message from a son process, the following intrinsic call could be used:

```
STAT:=RECEIVEMAIL(SONPIN,MDATA,WAITSTAT);
```

The parameters specified are

<i>pin</i>	SONPIN, which contains the Process Identification Number of the son process.
<i>location</i>	MDATA, an array in the stack in which the incoming mail will be stored.
<i>waitflag</i>	WAITSTAT, a logical word. If WAITSTAT = FALSE (bit 15 = 0), the intrinsic will wait until the incoming mail is ready for collection. If WAITSTAT = TRUE (bit 15 = 1), the intrinsic will return to the calling process immediately.



One of the following status codes is returned to STAT:

Status	Meaning
0	The mailbox was empty (and WAITSTAT was FALSE).
1	No message was collected because the mailbox contained outgoing mail from the receiving process.
2	The message was collected successfully.
3	An error occurred because of an illegal PIN or a bounds check failed.
4	The request was rejected because <i>waitflag</i> specified that the receiving process should wait for mail if the mailbox is empty, but the other process sharing the mailbox is already suspended, waiting for mail. If <i>both</i> processes were suspended, neither could activate the other, and they may be deadlocked.

## AVOIDING DEADLOCKS

Since the simultaneous use of mail transmission, process suspension, and RIN locking intrinsics throughout a process structure could result in a deadlock if the intrinsic calls are not synchronized properly, you should be aware of the following:

1. In a multi-process job/session, whenever a process is suspended (through the SUSPEND intrinsic, or when locking a RIN or receiving mail), MPE does not determine whether all other processes in the tree are suspended. You must exercise caution in avoiding such a situation.
2. An attempt by a process to lock a global RIN succeeds only if two conditions are met:
  - a. No other process within the job/session currently has locked this RIN — a *global* RIN cannot be used as a *local* RIN, because deadlock within the same job/session could otherwise occur.
  - b. The calling process currently has no other global RIN locked for itself. This could otherwise result in deadlock between two jobs/sessions.

## RESCHEDULING PROCESSES

When a process is created, it is scheduled on the basis of a priority class assigned by its father. After this point, its priority class can be changed at any time with the GETPRIORITY intrinsic. A process can change its own priority or that of a son but it cannot reschedule its father.

Generally, MPE schedules processes in linear or circular subqueues, as described in the *MPE General Information Manual*. The standard linear subqueues are

- The *AS* subqueue, containing processes of very high priority.
- The *BS* subqueue, containing processes of high priority.
- The *ES* subqueue, containing idle processes with low priority.

The circular subqueues are

- The *CS* subqueue recommended for interactive processes.
- The *DS* subqueue, available for general use at a lower priority than the *CS* subqueue and recommended for jobs.

The subqueue to which a process belongs determines the priority class of the process. From highest to lowest priority, these classes (named after their subqueues), are

AS  
BS  
CS  
DS  
ES

To reschedule itself with the priority class “D”, a process would make the following call:

```
GETPRIORITY (0,“DS”);
```

The 0 parameter specifies that the calling process is rescheduling itself. If the process were rescheduling a son process, the Process Identification Number of the son processes would be specified.

## **DETERMINING SOURCE OF ACTIVATION**

After a suspended process is reactivated, it can determine whether the source of the activation request was its father process or one of its son processes with the *GETORIGIN* intrinsic call.

For example, the following intrinsic call could be used:

```
SOURCE:=GETORIGIN;
```

One of the following codes would be returned to *SOURCE*:

1                      Activated by its father.

2                      Activated by a son.

## **DETERMINING FATHER PROCESS**

A process can determine the Process Identification Number of its father with the *FATHER* intrinsic.

For example, the following intrinsic call could be used:

```
PIN:=FATHER;
```

The Process Identification Number of the father is returned to *PIN*.

## DETERMINING SON PROCESSES

A process can request the return of the Process Identification Number assigned to any of its sons with the GETPROCID intrinsic.

For example, the following intrinsic call would return the Process Identification Number of the sixth existing son of the calling process to the word PINNUM:

```
PINNUM:=GETPROCID(6);
```

## DETERMINING PROCESS PRIORITY AND STATE

A process can request the return of a double-word message denoting the following information about its father or sons with the GETPROCINFO intrinsic:

Word	Bits	Meaning
1	(8:8)	The process' priority number in the master queue.
	(0:8)	Reserved for MPE. These bits are set to zero by the system.
2	(15:1)	Activity State. 1 = The process is active. 0 = The process is suspended.
	(13:2)	Suspension Condition. (Set only if bit 15 = 0) 01 = The process expects to be activated by its father. 10 = The process expects to be activated by a son.
	(9:4)	Reserved for MPE. These bits are set to zero by the system.
	(7:2)	Origin of the last ACTIVATE Call. 01 = Father. 10 = Son. 00 = MPE.
	(4:3)	Queue Characteristics. 001 = Master queue. 111 = Linear queue. 100 = Circular queue.
	(0:4)	Reserved for MPE. These bits are set to zero by the system.

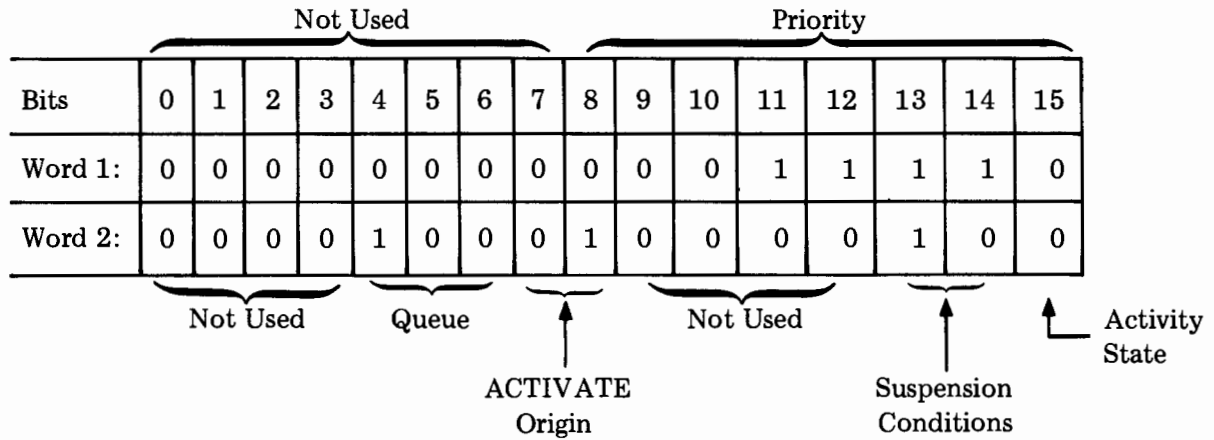
For example, to request information about its father, the following intrinsic call could be used:

```
INFO:=GETPROCINFO(0);
```

The 0 parameter specifies that the process is the father. If the process was a son process, the Process Identification Number of the son process would have been specified.



The information returned to the double-word INFO is of the following form:



The information is interpreted as follows for the father process:

Word	Bits	Value	Meaning
1	(8:8)	%36	Process has priority 32 in master queue.
	(0:8)	0	Not used.
2	(15:1)	0	Process is suspended.
	(14:1)	0	} Process to be activated by its son.
	(13:1)	1	
	(9:4)	0	Not used.
	(7:2)	1	Origin of last ACTIVATE call was father of this process.
	(4:3)	4	Circular subqueue.

# DATA SEGMENT MANAGEMENT CAPABILITY

SECTION

VIII

During execution of a user program, many processes may be created, run, and deleted. For each process in execution, one or more *code segments*, and one *data segment*, exist. The *data segment* is private to the process and contains the data generated and manipulated by that process. In a user process, this segment is referred to as the *user's stack segment*.

## NOTE

See the *MPE General Information Manual* for discussions of *segments, processes, and the stack*.

A particular program, which consists of *code segments*, can be run by many user processes simultaneously, with all user processes accessing the same body of code. The *stack segment*, however, is private to each user process and cannot be shared among others. Additionally, user processes created by a user with the standard MPE capabilities may create and access one stack segment *only*.

MPE allows users with the *Data Segment Management Capability*, however, to create and access extra data segments for their processes during a job or a session. These segments are used for temporary storage of data while the creating processes exist. Each segment is assigned an identity that either allows it to be *shared* between different processes in a job or session, or declares it *private* to the creating process. When a process terminates, all private data segments created by it are destroyed automatically. Sharable data segments are saved until explicitly deleted or until the job or session ends, at which time they are destroyed.

Extra data segments are not directly addressable by user processes. They can be accessed only through intrinsics that move data between the user's stack and the extra data segments (DMOVIN, DMOVOUT). If a process not having the *Data Segment Management Capability* attempts to call these intrinsics, that process is aborted. The *Data Segment Management Capability* is assigned to the process at :PREP time by a user with this capability (:PREP . . . ; CAP = DS).

The maximum number of extra data segments allowed per process is determined at system configuration time, and this number may not be exceeded.

If you are a user who possesses the *Data Segment Management Capability*, you can

- Create an extra data segment.
- Transfer data from an extra data segment to the stack.
- Transfer data from the stack to an extra data segment.
- Change the size of an extra data segment.
- Delete an extra data segment.

## CREATING AN EXTRA DATA SEGMENT

A process can create or acquire an extra data segment with the GETDSEG intrinsic. The number of extra data segments that can be requested, and the maximum size allowed these segments, are limited by parameters specified when the system is configured. When an extra data segment is created, the GETDSEG intrinsic returns to the calling process a *logical index number*, assigned by MPE, that allows this process to reference the segment in later intrinsic calls. The GETDSEG intrinsic also is used to assign the segment an *identity* that either allows other processes in the same job or session to share the segment, or that declares it private to the calling process. If the segment is sharable, other processes in the same job/session can obtain its logical index (through GETDSEG) and use this index to reference the segment. Thus, the logical index is a local name that identifies the segment throughout any process that obtained the index with the GETDSEG intrinsic call. The logical index need not be the same value in all processes sharing the same data segment. The GETDSEG intrinsic may return different logical index numbers to different processes, even though each process referenced the same data segment in their intrinsic calls. The *identity*, on the other hand, is a job-wide or session-wide name that allows any process to identify the data segment in order to obtain a logical index for it.

Figures 8-1, 8-2, and 8-3 contain three programs which illustrate the use of the GETDSEG, DMOVOUT, and DMOVIN intrinsics.

Together, the three programs perform the following:

1. Create an extra data segment which can be shared by all three processes.
2. Compute Julian calendar dates for any year and store these dates in an array.
3. Transfer the Julian calendar dates from the array to the extra data segment.
4. Create and activate two processes of the program shown in figure 8-3, each of which shares the same code but has its own data stack.
5. Each of the processes created in 4 above:
  - a. Opens a terminal for input/output and, once a `:DATA filename` command is entered on the terminal, requests month and day information from the user.
  - b. Moves the Julian dates, for the month entered by the user, from the extra data segment to its own stack.
  - c. Computes the Julian date based on the day of the month entered by the user and displays this information on the terminal.

The program in figure 8-1, called DSINIT, creates an extra data segment 372 words long, fills an array with values representing Julian calendar dates for a particular year entered by a user, then transfers this information from its stack to the extra data segment.

The program in figure 8-2 called DSBOSS, creates and activates two processes. Each of the two processes created is a process to run the program shown in figure 8-3, thus each process shares the same code but has its own data stack.

```

00001000 00000 0 SCONTROL USLINI1
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1 BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1 INTEGER IN,OUT,LGTH,MONTH,DAY,YEAR,DATE:=1,DSLGH:=372;
00006000 00005 1 LOGICAL DSINDX;
00007000 00005 1 ARRAY HEAD(0:14):="GENERATE CALENDAR DATA SEGMENT";
00008000 00017 1 ARRAY BUFR(0:1):=2(" ");
00009000 00001 1 BYTE ARRAY BBUF(*)=BUFR;
00010000 00001 1 ARRAY REGST(0:5):="ENTER YEAR: ";
00011000 00006 1 INTEGER ARRAY MAXDAY(0:11):=31,28,31,30,31,30,
00012000 00006 1 31,31,30,31,30,31;
00013000 00014 1 INTEGER ARRAY CALENDAR(0:371):=372(-1);
00014000 00001 1 DEFINE CCL = IF < THEN QUIT#,
00015000 00001 1 CCNE= IF <> THEN QUIT#;
00016000 00001 1
00017000 00001 1 INTRINSIC FOPEN,FREAD,FWRITE,GETDSEG,DMOVOUT,BINARY,QUIT;
00018000 00001 1
00019000 00001 1 <<END OF DECLARATIONS>>
00020000 00001 1
00021000 00001 1 IN:=FOPEN(INPUT,%45); CCL(1); <<SSTDIN>>
00022000 00012 1 OUT:=FOPEN(OUTPUT,%414,1); CCL(2); <<SSTDLIST>>
00023000 00025 1
00024000 00025 1 FWRITE(OUT,HEAD,15,0); CCNE(3); <<PROGRAM ID>>
00025000 00035 1
00026000 00035 1 GETDSEG(DSINDX,DSLGH,"JD"); CCL(4); <<SHARED EXTRA DS>>
00027000 00044 1
00028000 00044 1 FWRITE(OUT,REGST,6,%320); CCNE(5); <<REQUEST CALENDAR YEAR>>
00029000 00054 1 LGTH:=FREAD(IN,BUFR,-4); CCNE(6); <<INPUT YEAR>>
00030000 00065 1 YEAR:=BINARY(HBUF,LGTH); CCNE(7); <<CONVERT YEAR>>
00031000 00075 1
00032000 00075 1 IF YEAR MOD 4 = 0 THEN MAXDAY(1):=29; <<FIX FEB FOR LEAP YEAR>>
00033000 00105 1
00034000 00105 1 FOR MONTH:=0 UNTIL 11 DO <<INDEX 12 MONTHS>>
00035000 00112 1 FOR DAY:=0 UNTIL MAXDAY(MONTH)-1 DO <<INDEX DAYS IN EA MONTH>>
00036000 00127 1 BEGIN
00037000 00132 2 CALENDAR(MONTH*31+DAY):=DATE; <<SET IN JULIAN DATE>>
00038000 00140 2 DATE:=DATE+1; <<INCR JULIAN DATE>>
00039000 00141 2 END;
00040000 00143 1
00041000 00143 1 DMOVOUT(DSINDX,0,372,CALENDAR); <<JULIAN CALENDAR TO DS>>
00042000 00150 1 CCNE(8); <<CHECK FOR ERROR>>
00043000 00153 1 END.
PRIMARY DB STORAGE=%021; SECONDARY DB STORAGE=%00636
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:10

```

Figure 8-1. Using the GETDSEG and DMOVOUT Intrinsics (Program DSINIT)

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1 BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1 INTEGER IN,OUT,LGTH,PIN1,PIN2;
00006000 00005 1 BYTE ARRAY PFILE(0:6):="DSACCS ";
00007000 00005 1 ARRAY MESSAGE(0:29):="WHEN ALL JULIAN DATE OPERATIONS ARE ",
00008000 00022 1 "COMPLETE TYPE: DONE.",%6412,"? ";
00009000 00036 1 ARRAY BUFR(0:1);
00010000 00036 1 BYTE ARRAY BBUF(*)=BUFR;
00011000 00036 1 DEFINE CCL= IF < THEN QUIT#,
00012000 00036 1 CCNE= IF <> THEN QUIT#;
00013000 00036 1
00014000 00036 1 INTRINSIC FOPEN,FWRITE,FREAD,CREATE,ACTIVATE,QUIT;
00015000 00036 1
00016000 00036 1 <<END OF DECLARATIONS>>
00017000 00036 1
00018000 00036 1 IN:=FOPEN(INPUT,%45); CCL(1); <<$STDIN>>
00019000 00012 1 OUT:=FOPEN(OUTPUT,%414,1); CCL(2); <<$STDLIST>>
00020000 00025 1
00021000 00025 1 CREATE(PFILE,,PIN1," 1"); CCNE(3); <<SON 1 USES TERMIO1 FILE>>
00022000 00037 1 CREATE(PFILE,,PIN2," 2"); CCNE(4); <<SON 2 USES TERMIO2 FILE>>
00023000 00051 1
00024000 00051 1 ACTIVATE(PIN1,0); CCNE(5); <<SON 1>>
00025000 00060 1 ACTIVATE(PIN2,0); CCNE(6); <<SON 2>>
00026000 00067 1
00027000 00067 1 WAIT:
00028000 00067 1 FWRITE(OUT,MESSAGE,30,%320); CCNE(7); <<TERMINATION INSTRUCTIONS>>
00029000 00077 1 LGTH:=FREAD(IN,BUFR,-4); CCNE(8); <<SUSPEND FOR READ>>
00030000 00110 1
00031000 00110 1 IF BBUF<>"DONE" THEN GO WAIT; <<IF DONE - END KILLS SONS>>
00032000 00131 1 END.
PRIMARY DB STORAGE=%013; SECONDARY DB STORAGE=%00053
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:02; ELAPSED TIME=0:00:09

```

Figure 8-2. Creating and Activating Two Son Processes (Program DSBOSS)

```

00001000 00000 0  SCONTROL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  BYTE ARRAY NAME(0:7):="TERMIO# ";
00004000 00005 1  BYTE ARFAY DEV(0:4):="TERM ";
00005000 00004 1  INTEGER FNO, LGTH, MONTH, DAY, DSLGTH, JDATE, CURRENT:=-1;
00006000 00004 1  LOGICAL DSINDX, PARM=0-4;
00007000 00004 1  ARRAY HEAD(0:9):="JULIAN DATE CALENDAR";
00008000 00012 1  ARRAY BUFR(0:1);
00009000 00012 1  BYTE ARRAY BBUFR(*)=BUFR;
00010000 00012 1  ARRAY MESSAGE(0:16):="MONTH = ", "DAY = ", "JULIAN DATE = ";
00011000 00021 1  BYTE ARRAY BMSG(*)=MESSAGE;
00012000 00021 1  ARRAY DATES(0:30);
00013000 00021 1  DEFINE CCNE = IF <> THEN QUIT#;
00014000 00021 1
00015000 00021 1  INTRINSIC FOPEN, FREAD, FWRITE, GETDSEG, DMOVIN, BINARY, ASCII, QUIT;
00016000 00021 1
00017000 00021 1  <<END OF DECLARATIONS>>
00018000 00021 1
00019000 00021 1  NAME(6):=PARM; <<SET FORMALDES #=1 OR 2>>
00020000 00003 1  FNO:=FOPEN(NAME,%405,4,36,DEV); <<TERMINAL FILE TERMIO# >>
00021000 00015 1  IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00022000 00020 1
00023000 00020 1  FWRITE(FNO,HEAD,10,0); CCNE(2); <<PROGRAM ID>>
00024000 00030 1  GETDSEG(DSINDX,DSLGH,"JD"); <<SHARED CALENDAR DS>>
00025000 00034 1  IF <= THEN QUIT(3); <<ERROR OR NONEXISTENT>>
00026000 00037 1  GETMO:
00027000 00037 1  FWRITE(FNO,MESSAGE,4,%320); CCNE(4); <<REQUEST MONTH>>
00028000 00047 1  MOVE BUFR:=" "; <<BLANK READ BUFFER>>
00029000 00061 1  LGTH:=FREAD(FNO,BUFR,-2); CCNE(5); <<INPUT MONTH>>
00030000 00072 1  IF LGTH=0 THEN GO EXIT; <<NO MONTH - DONE>>
00031000 00075 1  MONTH:=BINARY(BBUFR,LGTH); <<CONVERT MONTH>>
00032000 00102 1  IF <> THEN GO GETMO; <<IF BAD TRY AGAIN>>
00033000 00103 1  IF NOT(1<=MONTH<=12) THEN GO GETMO; <<ILLEGAL MONTH CHECK>>
00034000 00112 1  GETDA:
00035000 00112 1  FWRITE(FNO,MESSAGE(4),3,%320); CCNE(6); <<REQUEST DAY>>
00036000 00123 1  MOVE BUFR:=" "; <<BLANK READ BUFFER>>
00037000 00132 1  LGTH:=FREAD(FNO,BUFR,-2); CCNE(7); <<INPUT DAY>>
00038000 00143 1  DAY:=BINARY(BBUFR,LGTH); <<CONVERT DAY>>
00039000 00150 1  IF <> THEN GO GETDA; <<IF BAD TRY AGAIN>>
00040000 00151 1  IF NOT(1<=DAY<=31) THEN GO GETDA; <<ILLEGAL DAY CHECK>>
00041000 00156 1  IF MONTH<>CURRENT THEN <<MONTH NOT IN BUFFER>>
00042000 00161 1  BEGIN
00043000 00161 2  DMOVIN(DSINDX,(MONTH-1)*31,31, <<GET MONTH FROM CALENDAR>>
00044000 00166 2  DATES); CCNE(8); <<CHECK FOR ERROR>>
00045000 00173 2  CURRENT:=MONTH; <<UPDATE MONTH IN BUFFER>>
00046000 00175 2  END;
00047000 00175 1  JDATE:=DATES(DAY-1); <<GET JULIAN DATE>>
00048000 00201 1  IF JDATE<0 THEN GO GETDA; <<UNINITIALIZED DATE>>
00049000 00204 1  MOVE MESSAGE(14):=" "; <<BLANK OUTPUT BUFFER>>
00050000 00216 1  LGTH:=ASCII(JDATE,10,BMSG(28)); <<CONVERT JULIAN DATE>>
00051000 00225 1  FWRITE(FNO,MESSAGE(7),10,0); CCNE(9); <<OUTPUT DATE ON TERMIO# >>
00052000 00236 1  GO GETMO; <<CONTINUE>>
00053000 00237 1  EXIT:END.
PRIMARY DB STORAGE=%020; SECONDARY DB STORAGE=%00103
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:13

```

Figure 8-3. Using the GETDSEG and DMOVIN Intrinsics (Program DSACCS)

The program in figure 8-3, called DSACCS, opens a terminal for input/output, acquires the extra data segment created by DSINIT, requests a month and day from the user, then transfers the Julian dates contained in that particular month into its own data stack. Because DSBOS (figure 8-2) has created two processes for the program shown in figure 8-3, and has activated both processes, the functions performed by DSACCS are duplicated (i.e., two terminals are opened for input/output, two users can enter the month and day, etc.).

#### NOTE

The three programs in figures 8-1, 8-2, and 8-3 must specify the Data Segment and Process Handling capability when they are prepared, as follows:

```
DSINIT (figure 8-1)
:PREP $OLDPASS,DSINIT;CAP=DS
DSBOS (figure 8-2)
:PREP $OLDPASS,DSBOS;CAP=PH
DSACCS (figure 8-3)
:PREP $OLDPASS,DSACCS;CAP=DS
```

In all cases above, it is assumed that \$OLDPASS contains the compiled USL file for each of the three programs.

In figure 8-1, the statement

```
INTEGER ARRAY MAXDAY(0:11):=31,28,31,30,31,30,
                           31,31,30,31,30,31;
```

initializes a 12-word integer array to represent the number of days in each month of the year.

The statement

```
INTEGER ARRAY CALENDAR(0:371):=372(-1);
```

declares a 372-word integer array and initializes all 372 words to -1.

The two FOPEN intrinsic calls open \$STDIN and \$STDLIST so that FREAD and FWRITE intrinsic calls can be issued against these files.

An extra data segment is created with the statement

```
GETDSEG(DSINDEX,DSLNGTH,"JD");
```

The parameters specified are

*index*                    The logical word DSINDEX, to which the logical index number of the data segment will be returned. This index is used to refer to this data segment in later intrinsic calls from this process.

*length* DSLGTH, which has been initialized to 372 words (see statement number 5 in figure 8-1).

*id* "JD", which specifies that this data segment is sharable by other processes in the same job/session. Note that any process which is to create or share an extra data segment must have the Data Segment Capability. If the data segment being created were to be private to the creating process, zero would be specified for *id*.

Statements 28, 29, and 30 in figure 8-1 request the user to enter the calendar year, and convert this ASCII string to a binary value.

The statement

```
IF YEAR MOD 4 = 0 THEN MAXDAY(1):=29;
```

checks if the year is equally divisible by 4 and, if it is, adds the 29th day to February for the leap year.

The six statements beginning with

```
FOR MONTH:=0 UNTIL DO
```

establish two FOR loops. The inner loop steps from 0 to the maximum number of days minus 1 for each entry in the array MAXDAY. For example, when MONTH = 0, MAXDAY(MONTH) = 31, thus the FOR loop performs 31 iterations (0 to MAXDAY(MONTH) -1).

The statement

```
DATE:=DATE+1;
```

increments the date each time the FOR loop is repeated. When the inner loop is satisfied, the MONTH FOR loop steps one iteration and the inner loop repeats. The array CALENDAR thus is filled with Julian dates as shown in figure 8-4. The array is linear, of course, and is shown as a day/month matrix for illustrative purposes only. All elements of the array were initialized to -1, and positions in the array that retain the value -1 signify invalid dates.

The data contained in CALENDAR is transferred from the stack to the extra data segment with the statement

```
DMOVOUT(DSINDEX,0,372,CALENDAR);
```

The parameters specified are

*index* DSINDEX, which contains the logical index returned by MPE when the GETDSEG intrinsic was executed.

*disp* 0, specifying the first word in the data segment. This is the starting location for the first word transferred from CALENDAR to the extra data segment.



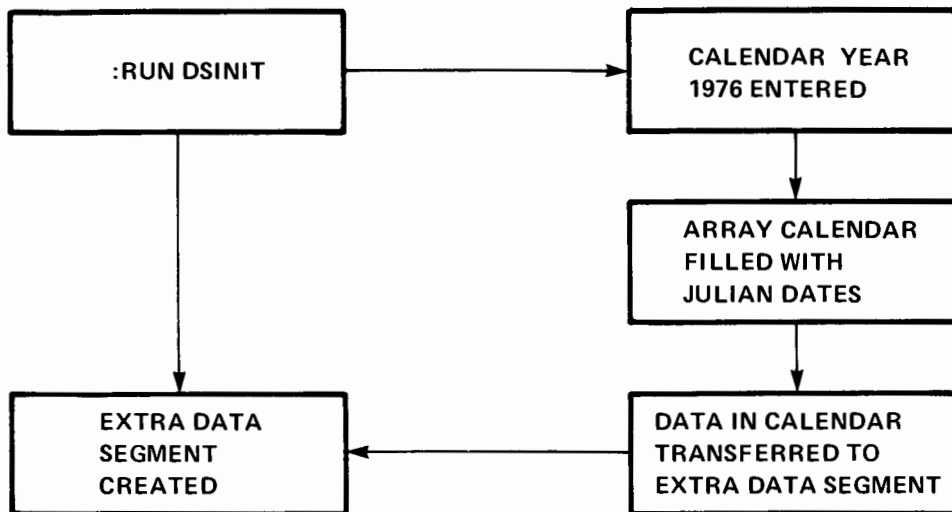
		DAYS																															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
MONTHS	JAN	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	FEB	1	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	-1	-1
	MAR	2	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91
	APR	3	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	-1
	MAY	4	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152
	JUN	5	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	-1
	JUL	6	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213
	AUG	7	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244
	SEP	8	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	-1
	OCT	9	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305
	NOV	10	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	-1
	DEC	11	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366

Figure 8-4. Array CALENDAR

*number* 372, specifying the size, in words, of the data block to be transferred.

*location* CALENDAR, the data block to be transferred begins at the address in the stack specified in CALENDAR.

At this point, the following events have occurred:



When the :RUN DSBOS command is entered, the program in figure 8-2 (DSBOS) executes. Statements 18 and 19 open \$STDIN and \$STDLIST to accept FREAD and FWRITE intrinsic calls. The statement

```
CREATE(PFILE,PIN1,"1");
```

creates a process. The parameters specified are

*programe* PFILE, a byte array containing the string "DSACCS", which is the name of the file containing the program to be run. Note that DSACCS is the name of the program in figure 8-3, thus the process is created for this program.

*entryname* Omitted. The primary entry point is specified by default.

*pin* PIN1, a word to which the Process Identification Number of the process will be returned.

*param* "1", a parameter used to transfer control information to the new process. The new process can access this control information ("1") in location Q - 4 of its data stack.

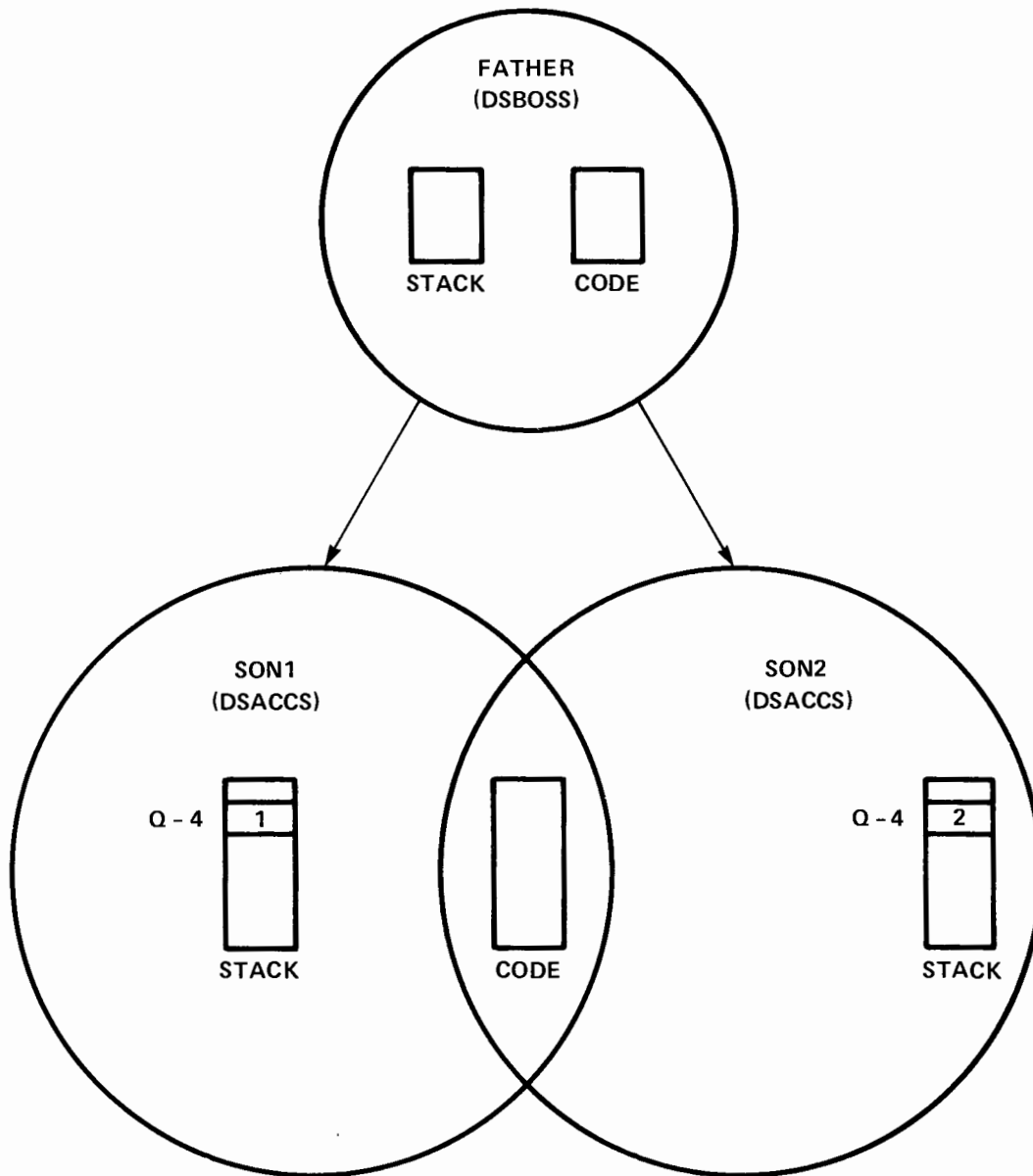
All other parameters are omitted from the CREATE intrinsic call.

The statement

```
CREATE(PFILE,PIN2, "2");
```

also creates a process for the program DSACCS. This time the Process Identification Number is returned to PIN2, and the control parameter "2" is located at stack location Q - 4 for this process.

The two `ACTIVATE` intrinsic calls activate the two processes. Note that the `susp` parameter 0 specifies that the father process will not be suspended when the sons are activated. Program `DSBOSS`, therefore, has created and activated two processes as follows:



The four statements beginning with

`WAIT:`

suspend `DSBOSS` for I/O until "DONE" is entered on `$STDIN`. `DSBOSS` did not suspend when the sons were activated. The reason for this is that when two or more sons are created, and the father is suspended when the last son is activated, it is possible that the sequence of events will be

such that the sons are unable to reactivate the father. The following sequence illustrates how this could happen:

1. Create two sons. Activate SON1.



*Father and SON1 both active.*

2. Activate SON2. Suspend father. Father expects to be reactivated when either son terminates.

*Father suspended; SON1 and SON2 active.*

3. SON1 terminates, reactivating father. Father reactivates, determines that SON2 is still active, and re-suspends itself. *However*, while father was active, SON2 terminated. Attempt to reactivate father failed because father already was active. Thus, when father re-suspends itself, it suspends indefinitely because both sons have terminated.

The two processes, SON1 and SON2, both of which are DSACCS, are shown in figure 8-3.

The statement

```
FNO:=FOPEN(NAME,%405,4,36,DEV);
```

opens a terminal for input/output. The parameters specified are

*formal designator*

NAME, which contains the string TERMI01 when this call is issued by SON1 and TERMI02 when the call is issued by SON2. Note that in statement number 3, NAME is set equal to TERMIO#. The statement

```
NAME(6):=PARM;
```

however, replaces # with 1 or 2, depending on the parameter contained in stack location Q - 4 (this parameter was passed to the process by the CREATE intrinsic call in program DSBOSS). Thus, by using different formal designators, SON1 opens one terminal and SON2 opens another.

#### NOTE

Unlike disc files, where each formal designator must be unique in its domain (temporary or permanent), two or more devices can be opened with the same formal designator. For example, two :DATA commands

```
:DATA FIELD.SUPPORT;TERMIO  
:DATA FIELD.SUPPORT;TERMIO
```

would cause MPE to search the device directory for two available terminals and, if two are available, both would be allocated. Using different formal designators, however, allows a user to direct output to a particular terminal with a :FILE command.

<i>foptions</i>	%405, which specifies the following: <ol style="list-style-type: none"> <li>a. Old file.</li> <li>b. ASCII.</li> <li>c. Actual file designator is the same as the formal file designator.</li> <li>d. Fixed-length records.</li> <li>e. Carriage-control character expected.</li> </ol>
<i>aoptions</i>	4, which specifies input/output access.
<i>reclsize</i>	36, specifying 36 words.
<i>device</i>	DEV, a byte array specifying the device class ("TERM").

All other parameters are omitted from the FOPEN intrinsic call.

The shared data segment is acquired with the statement

```
GETDSEG(DSINDEX,DSLGLTH,"JD");
```

Note that the process quits unless the CCG condition code, signifying that an extra data segment with this identity exists already, is returned (see statement number 25).

A month is requested from the user and the input is converted to binary. The user then is requested to enter a day and this information is read and converted to binary.

The statement

```
IF MONTH <> CURRENT THEN
```

checks whether the month information is different than the information currently in the stack. If it is, the statement

```
DMOVIN(DSINDEX,(MONTH-1)*31,31,DATES);
```

transfers the Julian dates for the month entered by the user into the 31-word array DATES. For example, if the user entered 3, the values 61 through 91 (see figure 8-4), corresponding to the Julian calendar dates for the month of March are transferred from the extra data segment to the array DATES in the stack. Data representing the entire month, instead of data representing the specific day entered by the user, is transferred by DMOVIN because DMOVIN, which requires considerable time to execute, should be used sparingly to maintain programming efficiency. Transferring the data for the entire month saves time if the user's next request is for a Julian date in the same month. Note that months are numbered from 0 to 11.

The buffer CURRENT is updated to the current month with the statement

```
CURRENT:=MONTH;
```

The Julian date is computed with the statement

```
JDATE:=DATES(DAY-1);
```

and this information is output to the user.

The three examples below illustrate using the three programs shown in figures 8-1, 8-2, and 8-3.

#### EXAMPLE 1

```
:RUN DSINIT  
  
GENERATE CALENDAR DATA SEGMENT  
ENTER YEAR: 1976  
  
END OF PROGRAM  
:RUN DSBOS  
  
WHEN ALL JULIAN DATE OPERATIONS ARE COMPLETE TYPE: DONE.  
? DONE  
  
END OF PROGRAM
```

#### EXAMPLE 2

```
:DATA FIELD.SUPPORT;TERMIO1  
JULIAN DATE CALENDAR  
MONTH = 11  
DAY = 31  
DAY = 30  
JULIAN DATE = 335  
MONTH = 2  
DAY = 29  
JULIAN DATE = 60  
MONTH = 6  
DAY = 1#  
DAY = 13  
JULIAN DATE = 165  
MONTH = 13  
MONTH = 0  
MONTH = 3  
DAY = 29  
JULIAN DATE = 89  
MONTH =
```

### EXAMPLE 3

```
:DATA FIELD.SUPPORT;TERM102  
JULIAN DATE CALENDAR  
MONTH = -9  
MONTH = 9  
DAY = 15  
JULIAN DATE = 259  
MONTH = 7  
DAY = 20  
JULIAN DATE = 202  
MONTH = 8  
DAY = 14  
JULIAN DATE = 227  
MONTH = 5  
DAY = 3
```

In example 1, the command :RUN DSINIT causes DSINIT to execute. It prints the purpose of the program and a request to the user to enter the year for which Julian dates are required. When 1976 is entered, DSINIT creates an extra data segment, fills an array with Julian dates for the year 1976, transfers this data to the extra data segment, and terminates.

The :RUN DSBOSS command causes DSBOSS to execute. DSBOSS creates and activates two son processes (DSACCS), then suspends itself after the message

```
WHEN ALL JULIAN DATE OPERATIONS ARE COMPLETE TYPE: DONE.  
?
```

Example 2 illustrates the SON1 process execution.

1. A user enters a :DATA statement on a terminal. (Remember that SON1 and SON2 have each opened a terminal for input/output.)
2. MPE searches the device class directory for a terminal with the formal designator TERM101, and allocates the terminal.

In response to the month and day requests, the user enters

```
MONTH = 11
```

```
DAY = 31
```

DSACCS determines that 31 is not a valid day for month 11 with the statement

```
IF JDATE < 0 THEN GETDA;
```

(see figure 8-3). Recall that invalid dates in the array CALENDAR (see figure 8-4) are signified by -1. DSACCS prompts for a new day and the user enters 30. DSACCS computes the Julian date for November 30th and displays:

```
JULAIN DATE = 335
```

Example 3 shows a second user accessing terminal 2.

When a user types DONE on \$STDIN, see example 1, the father process terminates, terminating both sons.

## **DELETING AN EXTRA DATA SEGMENT**

A process can release an extra data segment assigned to it with the FREEDSEG intrinsic. If this is a private data segment, or if it is a sharable segment not currently assigned to any other process in the job/session, the segment is deleted from the entire job/session. Otherwise, it is deleted from the calling process but continues to exist in the job/session.

For example, to delete a data segment with the logical index contained in INDX, the following intrinsic call could be used. Because the *id* parameter is specified as 0, the data segment is deleted from both the process and the job.

```
FREEDSEG(INDX,0);
```

## **TRANSFERRING DATA FROM AN EXTRA DATA SEGMENT TO THE STACK**

A process can copy a block of words from an extra data segment into the stack with the DMOVIN intrinsic. A bounds check is performed by the intrinsic on both the extra data segment and the stack to insure that the data is taken from within the data segment boundaries and moved to an area within the stack boundaries.

The DMOVIN intrinsic call is illustrated in figure 8-3 and described on page 8-12.

## **TRANSFERRING DATA FROM THE STACK TO AN EXTRA DATA SEGMENT**

A process can copy a block of words from the stack to an extra data segment with the DMOVOUT intrinsic. A bounds check is performed by the intrinsic to insure that the data is taken from an area within the stack boundaries and moved to an area within the extra data segment.

The DMOVOUT intrinsic call is illustrated in figure 8-1 and described on page 8-7.

## **CHANGING THE SIZE OF AN EXTRA DATA SEGMENT**

You can change the current size of an extra data segment with the ALTDSEG intrinsic. As a typical application, after you have obtained disc storage for a new segment by calling the GETDSEG intrinsic, you may issue ALTDSEG to reduce the storage required by the segment when it is moved into main memory, and later expand it as needed, for more efficient use of memory. In no case can ALTDSEG be used to increase segment size beyond that assigned originally through GETDSEG.

The form of the ALTDSEG intrinsic call is

```
ALTDSEG(INDEX,INC,SIZE);
```

where

INDEX is a word containing the logical index of the extra data segment, obtained through the GETDSEG intrinsic call.



**INC** is the value, in words, by which the extra data segment is to be changed. A positive integer value specifies an increase and a negative integer value specifies a decrease.

**SIZE** is a word to which is returned the new size of the data segment after incrementing or decrementing has taken place.

# PRIVILEGED MODE CAPABILITY

SECTION

IX

If you are an MPE user with standard MPE capabilities, you can access only your own code and data areas in main memory. But if you are a user with the *Privileged Mode Capability*, you can access all areas of the system and can use all features of the hardware. You can access all system tables, and can invoke all system instructions, including those in the privileged central processor instruction set. You can, in short, use the computer on the same terms as MPE itself. (In fact, MPE does not distinguish a privileged user as not being MPE itself.)

## IMPORTANT NOTE

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode. It is possible for a privileged mode program to destroy file integrity, including the MPE operating system software itself. Hewlett-Packard will investigate and attempt to resolve problems resulting from the use of privileged mode code. This service, which is not provided under the standard Service Contract, is available on a time and materials billing basis. However, Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.

You can use the *Privileged Mode Capability* in two ways:

1. You can write permanently privileged programs that are loaded and executed entirely in privileged mode.
2. You can write temporarily privileged programs that dynamically enter and leave privileged mode during execution, as required.

## PERMANENTLY PRIVILEGED PROGRAMS

A program's segments are loaded and executed directly in privileged mode when all three of the following conditions exist:

1. Any of the program's code segments contain privileged instructions.
2. The program is prepared with the *Privileged Mode Capability*, by entering the appropriate capability-class attribute in the *caplist* parameter of the :PREP or :PREPRUN command that prepares the program. (See the *MPE Commands Reference Manual* for discussions of the :PREP and :PREPRUN commands.) Note that entering a privileged capability class attribute requires that you have been assigned the *Privileged Mode Capability*.

3. The NOPRIV optional parameter is omitted from the :PREPRUN or :RUN command that executes the program, or the CREATE intrinsic that creates a process to run it. This omission leaves the privileged mode bit in the appropriate CST entries *on*.

When you add a segment to a Segmented Library (through the -ADDSL Segmenter command), the procedures within the segment are checked to determine if any one of them is privileged. If it is, the segment is always run in privileged mode. In order to add a segment containing one or more privileged procedures to a library, you must possess the *Privileged Mode Capability*. See the *MPE Segmenter Reference Manual* for instructions concerning Segmented Libraries.

## TEMPORARILY PRIVILEGED PROGRAMS

Temporarily privileged programs are initiated, upon request, in the non-privileged mode. Then, intrinsics can be used to change the program to and from the privileged mode dynamically. For example, just before a set of privileged instructions is encountered, the program can be switched to the privileged mode to allow execution of these instructions. Then, after the last privileged instruction in the set is encountered, the program can be returned to non-privileged mode. This bracketing of privileged instructions aids in reducing system violations, since the program cannot access locations or resources outside the user environment when it is running in non-privileged mode.

Before running a temporarily-privileged program, you should understand how the central processor handles procedure calls (PCAL instruction) and exits (EXIT instructions) when encountered in either mode:

In the *privileged* mode, when a PCAL instruction is executed, privileged mode is retained even though the destination code segment may have a non-privileged CST entry. When an EXIT instruction is encountered, the resulting mode depends on the status word in the stack marker.

In the *non-privileged* mode, when a PCAL instruction is encountered, the mode assumed is obtained from the CST entry for the destination code segment. When an EXIT instruction occurs, the resulting mode is taken from bit 0 of the status in the stack marker. If the entry indicates privileged mode, a system violation occurs.

In general, the status word determines the action taken in privileged mode, but the CST determines the action in non-privileged mode.

### NOTE

See the *Machine Instruction Set Reference Manual* for further discussions of the PCAL and EXIT instructions.

A program is loaded and begins execution as a temporarily-privileged program (in the non-privileged mode) when these two conditions are met:

1. The program is prepared with the *Privileged Mode Capability*, by entering the appropriate capability-class attribute in the *caplist* parameter of the :PREP or :PREPRUN command that prepares the program. This requires that you have been assigned the *Privileged Mode Capability*.

2. The NOPRIV parameter is *included* in the :PREPRUN or :PREP command that executes the program, or the CREATE intrinsic that creates a process to run it.

When a temporarily-privileged program is initiated, the CST entries corresponding to its segments have their privileged-mode bits set *off*.

If you possess *Privileged Mode Capability*, you also can call all intrinsics available to users with the *Data Segment Management Capability* (Section VIII), provided that you acknowledge these rules:

1. When calling the data segment intrinsics from *privileged mode*, ensure that the DB register points to its normal stack position. When the GETDSEG intrinsic is used to create extra data segments under these conditions, the number of segments that can be created is limited only by the space available in the Process Control Block Extension. This number is virtually unlimited.
2. When a temporarily-privileged process calls a data segment intrinsic while in *non-privileged* mode, the data segment index returned to the calling process also can be used by the process to reference that segment in *privileged* mode. If the process calls a data segment intrinsic in *privileged* mode, however, the index returned *cannot* be used to reference the segment in *non-privileged* mode.

## ENTERING PRIVILEGED MODE

The GETPRIVMODE intrinsic is used to switch a temporarily-privileged program from the non-privileged mode to the privileged mode. This intrinsic turns the privileged mode bit in the status register *on*, but leaves the privileged mode bit in the Code Segment Table entry for the executing segment *off*. Thus, if additional segments are to be run as part of the program, they will be run in privileged mode unless an intrinsic is specifically called to return to the non-privileged mode, because the status register rather than the Code Segment Table determines a mode change when running in privileged mode.

Figure 9-1 contains a program that uses the GETPRIVMODE intrinsic to switch to privileged mode. Privileged mode is necessary temporarily because the program opens a file with both NOBUF and NOWAIT *aoptions* specified in the FOPEN intrinsic call. Privileged mode capability is necessary for this because your I/O could overwrite other data in the system unless caution is used.

The program in figure 9-1 was prepared with the CAP = PM parameter specified in the :PREP command. This enables the program to be switched from non-privileged to privileged mode with the GETPRIVMODE intrinsic. The statement in the program

```
GETPRIVMODE;
```

switches the program from non-privileged to privileged mode before the next statement opens a file with both the NOBUF and NOWAIT *aoptions* specified.

The statement

```
CCG(2);
```

causes the program to quit if a CCG condition code (signifying that the program already is running in privileged mode) is returned.

```

00001000 00000 0  $CONTROL USLINIT
00002000 00000 0  BEGIN
00003000 00000 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00004000 00005 1  BYTE ARRAY TNAM(0:6):="DATAIN ";
00005000 00005 1  BYTE ARRAY DEV(0:7):="LP TERM ";
00006000 00005 1  INTEGER OUT,FILE,LGTH,I:=-1,PROMPT:="? ",DONE:=0;
00007000 00005 1  EQUATE MAXTRM=3;
00008000 00005 1  ARRAY BUFR(0:36*MAXTRM);
00009000 00005 1  INTEGER ARRAY OPEN(0:MAXTRM);
00010000 00005 1  DEFINE CCL = IF < THEN QUIT#,
00011000 00005 1  CCG = IF > THEN QUIT#,
00012000 00005 1  CCNE= IF <> THEN QUIT#;
00013000 00005 1
00014000 00005 1  INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,GETPRIVMODE,GETUSERMODE,
00015000 00005 1  IOWAIT,QUIT;
00016000 00005 1
00017000 00005 1  <<END OF DECLARATIONS>>
00018000 00005 1
00019000 00005 1  OUT:=FOPEN(OUTPUT,4,1,,DEV); CCL(1); <<LINEPRINTER OUTPUT>>
00020000 00015 1  WHILE (I:=I+1)<MAXTRM DO <<LOOP-SET UP TERMS>>
00021000 00023 1  BEGIN
00022000 00023 2  GETPRIVMODE; CCG(2); <<FOR NOWAIT FOPEN>>
00023000 00027 2  FILE:=FOPEN(TNAM,%405,%4404,36,DEV(3)); <<DATA INPUT TERMINAL>>
00024000 00042 2  CCL(3); <<CHECK FOR ERROR>>
00025000 00045 2  GETUSERMODE; CCG(4); <<FOR NOWAIT I/O>>
00026000 00051 2  OPEN(I):=FILE; <<SAVE FILE NUMBERS>>
00027000 00054 2  FWRITE(FILE,PROMPT,1,%320); CCNE(5); <<OUTPUT ? PROMPT>>
00028000 00064 2  IOWAIT(FILE); CCNE(6); <<COMPLETE REQUEST>>
00029000 00075 2  FREAD(FILE,BUFR(I*36),-72); CCNE(7); <<INPUT DATA-NOWAIT>>
00030000 00111 2  END;
00031000 00116 1  WAIT:
00032000 00116 1  FILE:=IOWAIT(0,,LGTH); CCL(8); <<WAIT FOR 1ST DONE>>
00033000 00130 1  IF > THEN <<EOF ON TERM READ>>
00034000 00131 1  BEGIN
00035000 00131 2  FCLOSE(FILE,0,0); CCL(9); <<TERMINAL FILE>>
00036000 00137 2  IF(DONE:=DONE+1)>=MAXTRM THEN GO EXIT; <<ALL TERMS CLOSED?>>
00037000 00143 2  END
00038000 00143 1  ELSE
00039000 00145 1  BEGIN
00040000 00145 2  I:=-1; <<SET BUFFER INDEX>>
00041000 00147 2  DO I:=I+1 <<INCR BUFFER INDEX>>
00042000 00147 2  UNTIL OPEN(I)=FILE OR I=MAXTRM; <<SEARCH FOR FILE NO>>
00043000 00157 2  IF I=MAXTRM THEN QUIT(10); <<FILE NOT FOUND>>
00044000 00164 2  FWRITE(OUT,BUFR(I*36),-LGTH,0); <<COPY INPUT TO LP>>
00045000 00174 2  CCNE(11); <<CHECK FOR ERROR>>
00046000 00177 2  FWRITE(FILE,PROMPT,1,%320); CCNE(12); <<OUTPUT ? PROMPT>>
00047000 00207 2  IOWAIT(FILE); CCNE(13); <<COMPLETE REQUEST>>
00048000 00220 2  FREAD(FILE,BUFR(I*36),-72); CCNE(14); <<INPUT DATA-NOWAIT>>
00049000 00234 2  END;
00050000 00234 1  GO TO WAIT; <<CONTINUE>>
00051000 00235 1  EXIT:END,
PRIMARY DB STORAGE=%013; SECONDARY DB STORAGE=%00175
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:02; ELAPSED TIME=0:00:08

```

Figure 9-1. Using the GETPRIVMODE and GETUSERMODE Intrinsics

## ENTERING NON-PRIVILEGED MODE

The `GETUSERMODE` intrinsic is used to change a temporarily-privileged program from the privileged to the non-privileged mode. This intrinsic changes the privileged mode bit in the status register to *off*.

The statement

```
GETUSERMODE;
```



in figure 9-1 illustrates this intrinsic call

## MOVING THE DB POINTER

If you have the Data Segment Management Capability, and run a process with an extra data segment in *privileged* mode, you can prepare for movement of data between this segment and the stack with the `SWITCHDB` intrinsic. This intrinsic changes the DB register so that it points to the base of the extra data segment rather than the base of the stack. The `SWITCHDB` intrinsic returns the logical index of the data segment indicated by the previous DB register setting, allowing you to restore this setting later. If the previous DB setting indicated the stack, zero is returned.

As an example, to set the DB register so that it points to the base of an extra data segment whose logical index is indicated in the word `INDEX2`, the following intrinsic call could be used:

```
SET:=SWITCHDB(INDEX2);
```

where `INDEX2` is a logical value denoting the logical index of the data segment to which the DB register is switched, as obtained through the `GETDSEG` intrinsic call. MPE checks the value specified for this parameter to insure that the process has previously acquired access to this segment. For an extra data segment, this parameter must be a positive, non-zero integer; and to switch back to the stack, this parameter must be zero.

The calling process is aborted if you try to call the `SWITCHDB` intrinsic from a program which does not have the Privileged Mode Capability.

## SCHEDULING PROCESSES

Every process in the system has a priority assigned to it. When a process is ready to run, it is placed in the `READY` list. When the dispatcher runs, it selects for execution the process with the highest priority that is in memory.

The master queue (see figure 9-2) is divided into logical areas, each corresponding to a particular type of dispatching and priority class for the processes within it. A logical area can be a linear subqueue, a circular subqueue, or a portion of the main master queue. In a linear subqueue, the process with highest priority accesses the central processor first and maintains this access until the process either is completed, terminated, or suspended to await the availability of a required resource. In a circular subqueue, all processes have equal priority and each accesses the central processor for an interval (time quantum) of maximum duration (or until completed, terminated, or suspended). At the end of this duration, control is transferred to the next process in the queue, continuing in a round-robin fashion. This time-slicing is controlled by the system timer. Processes that are not scheduled in a subqueue are scheduled in the master queue.

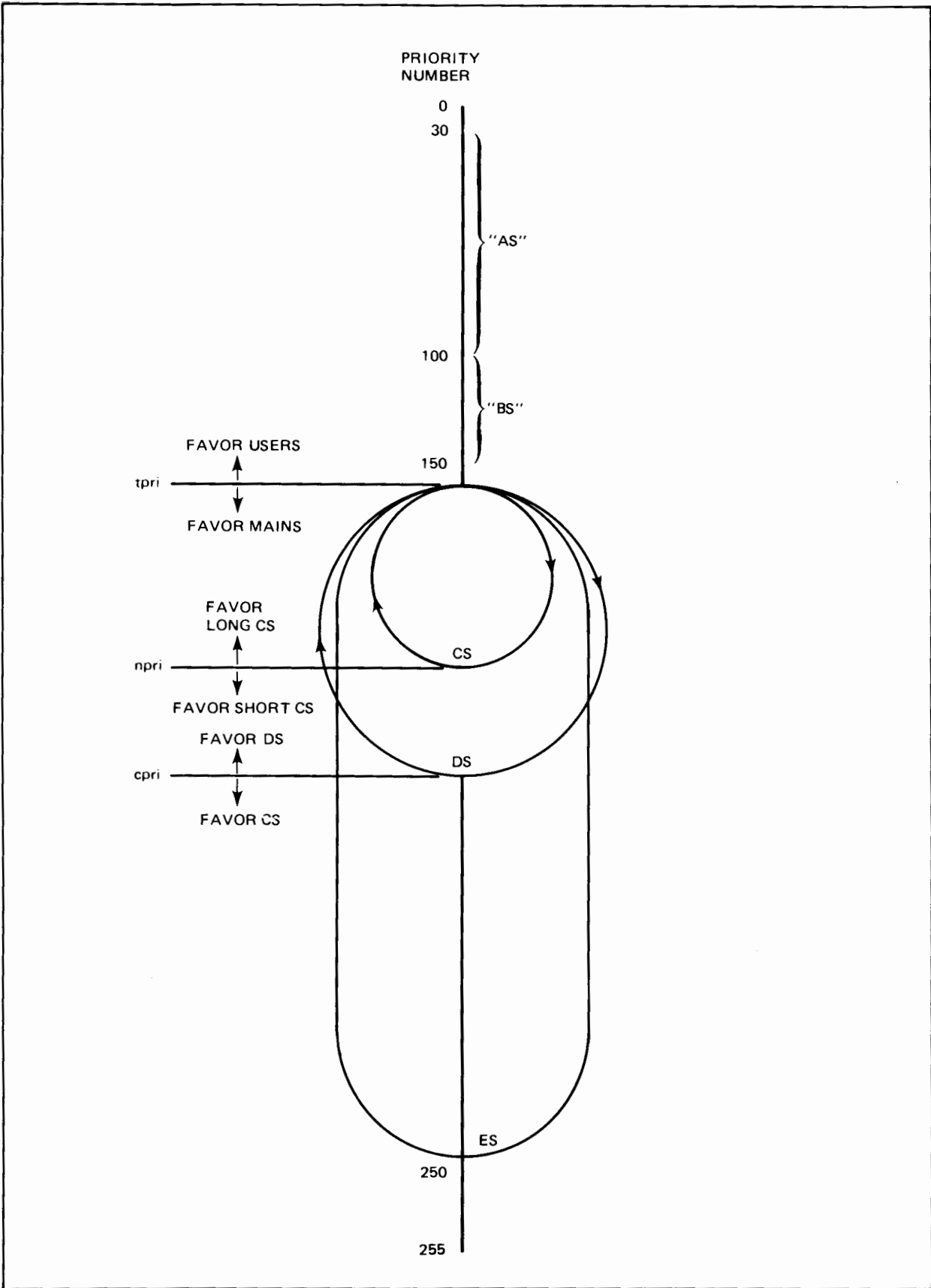


Figure 9-2. MPE Master Queue Structure

Each linear subqueue in the *master queue* is defined by a single priority number, and each circular subqueue is defined by a range of priority numbers. While the standard user is aware of the priority class associated with a subqueue, only a user with System Supervisor or Privileged Mode capability can deal with priority numbers. The standard subqueues (priority classes) are as follows:

AS Is a linear subqueue containing processes of very high priority. Its priority range is 30-99 and it is presently used only by MPE.

Scheduling Type: Linear  
Priority Range: 30-99

BS Is a linear subqueue containing processes of very high priority. It is accessible to users having MAXPRI = BS. Normally, priority for a BS process is 100. However, by specifying a rank > 0 in the CREATE or GET PRIORITY intrinsics, the process may be set in the master queue at min (100 + rank, 149).

Scheduling Type: Linear  
Priority Range: 100-149

CS Is a circular subqueue generally devoted to interactive sessions. A CS process which uses its quantum of CPU will be lowered in priority, but not below the C Subqueue Priority Limit, called cpri (which may be set in SYSDUMP or the QUANTUM Command).

Scheduling Type: Circular  
Priority Range: 150-*cpri*

DS Is a circular subqueue generally devoted to batch jobs. A DS process which uses its quantum of CPU will be lowered in priority more rapidly than a CS process, but not below the D Subqueue Priority Limit, or dpri (which may be set in SYSDUMP or QUANTUM Command).

Scheduling Type: Circular  
Priority Limits: 150-*dpri*

ES Is a circular subqueue generally used for so-called "idle" processes. When an ES process consumes a quantum of CPU, its priority is set to 250. Such a process will have a minimal impact on the performance of processes in other subqueues.

Scheduling Type: Circular  
Priority Limits: 150-250

In all cases, it should be remembered that low numeric values mean high priority in the system.

The System Manager has the ability to modify on line the value of the time quantum and priorities for the CS, DS, and ES priority classes.

Scheduling within the CS class is controlled by two factors: first, the reason why the process is ready to run and second, the amount of time that a process has been on the ready list. Typically,



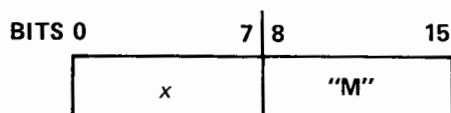
good terminal response is desirable, therefore, a low numeric priority is assigned to processes completing a terminal read operation. Processes performing other types of I/O do not need such fast response, however, and these processes can be assigned a higher numeric priority. The third class of processes have a quantum timeout. Typically, these are considered to have a lower priority than either of the foregoing types, so they are assigned a still higher priority number. As a process waits on the ready list, it steadily gets a higher priority. Thus, a CPU-bound process that has waited will be equal in priority to a process that has just completed a terminal read. This policy assures that all processes continue to run, even on a heavily loaded system. Both the time quantum and the priorities may be adjusted by the System Manager to give maximum performance as dictated by the installation's needs.

The priority class of a process can be specified by the normal user with standard or optional MPE capabilities. In the two-character string that comprises a priority-class reference, the first character refers to the location of a subqueue within the master queue (in alphabetical order) and the second character specifies whether the logical area is the subqueue itself (S) or the portion of the master queue (M) that immediately follows the subqueue. When a priority class is requested for a process, it is assigned the lowest priority within that class (relative to other processes already assigned the same class). In a circular subqueue, the actual priority of the process is modified as other processes use central processor time.

Only a user with Privileged Mode Capability can assign a priority number to a process. Priority numbers range from 1 to 255 inclusively, with 1 denoting the highest priority. Any two processes having the same priority number are in the same subqueue. The privileged mode user also can specify the relative ranks of processes having identical numbers within a linear subqueue, and can assign them specific priorities in the master queue.

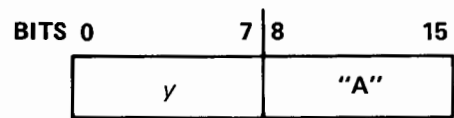
Priorities are assigned to processes through the *priorityclass* parameter of the CREATE and/or GETPRIORITY intrinsic. Because users with the Privileged Mode Capability can schedule processes within the master queue, the *priorityclass* parameter can take on the following values:

1. A string of two ASCII characters describing the standard priority-class (subqueue) in which the process is to be scheduled; the standard subqueues are "AS", "BS", "CS", "DS", or "ES".
2. An ASCII character (*x*) specifying a valid subqueue, followed immediately by the single-character string "M", indicating the Master Queue. The word format is:



This schedules the process in that region of the master queue directly adjacent to and below the subqueue *x*. The process is scheduled in the first available priority in that region.

3. An ASCII character ( $y$ ) specifying an absolute priority number, followed by the single-character string "A" indicating that  $y$  is an absolute priority number. The word format is:



This schedules the process at the location specified by  $y$  in the Master Queue. If another process or subqueue already occupies the location designated by  $y$ , the process is placed in the first location available moving toward the ES subqueue, and the process priority number is changed to reflect that location.



# MPE DIAGNOSTIC MESSAGES

SECTION

X

Programs running under MPE at any batch input device or terminal may return the following types of error messages:

- *Command Interpreter Error Messages*, reporting fatal errors that occur during the interpretation or execution of an MPE command. See the *MPE Commands Reference Manual* for a listing of these messages.
- *Command Interpreter Warning Messages*, reporting unusual conditions that occur during command interpretation or execution but that may not necessarily be detrimental to the processing of the job or session. See the *MPE Commands Reference Manual* for a listing of these messages.
- *Run-Time Messages*, denoting conditions that abort the running program, provided that an appropriate error trap has not been enabled.
- *User Messages*, which are messages sent to you by other users currently running jobs or sessions.
- *Operator Messages*, which are messages sent to you by the console operator.
- *System Messages*, which denote miscellaneous conditions that terminate or otherwise affect the job/session, such as an abort requested by the system supervisor or console operator.

Other messages may be received only at the system console. These are:

*Console Operator Messages*, including:

- Status Messages that indicate the current status of jobs/sessions or input/output devices.
- Input/Output Messages that request service for, and report errors on, input/output devices.
- User Messages, sent by users to the console operator.

*System Failure Messages*, including:

- System Failure Messages.
- Cold Load Error Messages.

## RUN-TIME MESSAGES

Your program can be aborted as a result of any of the following general types of run-time errors:

- Special violations — those detected through the internal interrupt structure (such as arithmetic trap errors, parity errors, bounds violations, etc.) and other violations detected by MPE (such as stack overflows or invalid stack markers). These are PROGRAM ERRORS and are described in a following table.
- Explicit calls to the QUIT intrinsic.
- Explicit calls to the QUITROG intrinsic.
- Violations of other callable intrinsics, such as passing of illegal parameters or the invoking of an intrinsic without having the required capability class or a valid register environment. (These are listed in the RUN-TIME error table. The intrinsics are listed in the INTRINSIC table, and errors encountered by them are listed in tables by specific intrinsic: FILESYSTEM, LOADER, CREATE, ACTIVATE, SUSPEND, MYCOMMAND and LOCKGLOBIN.

If an appropriate error trap has been armed, control transfers to the trap procedure which may attempt recovery or take some other action. But if no trap has been armed for the type of error encountered, MPE terminates the user's process and transmits a *run-time (abort) message* to the user's output device. In a multi-process structure, QUIT aborts only the violating process but all other errors abort the entire program.

If the aborted program was running in a batch job, the job is removed from the system (if no :CONTINUE command overrides termination).

If it was running in a session, control of the session is returned to you at the terminal.

### NOTE

*An abort-error will occur if a user process invokes certain callable intrinsics when the DB register is not pointing to its normal position (e.g. DB is pointing at an extra data segment). If this happens and a user trap procedure is invoked, the DB register is reset to the normal position before the trap procedure is entered.*

The format for run-time errors is:

ABORT:pname.segment.location:sname.segment.location

p-field

s-field

<msgtype>#<msgno>:<message>[.PARAM {<sub>#</sub><sup>=</sup>}<number>]

m-field (from 1 to 7 lines)

where:

**p-field** is the last location of the last instruction executed in the user program prior to the abort.

**s-field** is output only if the abort occurred when executing code belonging to a library segment referenced by the user program. The field provides the instruction location within the library segment that initiated the abort.

Within the p-field and s-field, the parameters are:

***pname*** The name of the program file containing the user's program and optionally, the group and account name.

In the special case of a process having been PROCREATED from a segment in a segmented library (SL) (for example, the Command Interpreter), an asterisk (\*) is output followed by the SL name in symbolic form (sname, below).

***sname*** The symbolic name of the SL in which the segment exists

SYSL — System SL

PUSL — Public SL

GRSL — Group SL

***segment*** The logical number of the code segment relating to either the program or SL, whichever is appropriate.

***location*** The location in the code segment. This is expressed in terms of the displacement (P-PB), where P is the absolute address of the instruction and PB the absolute address of the base of the code segment.

#### NOTE

Octal numbers are indicated by a percent sign (%) preceding the number.

If the stack is completely destroyed and no valid stack markers can be found that define a user environment, then the above-defined subfields will be output containing a question mark (?).

m-field contains the error message text.

The parameters within the m-field are

< msgtype > is one of:

PROGRAM ERROR  
ERROR: INTRINSIC  
RUN-TIME ERROR  
FILESYSTEM ERROR  
LOADER ERROR  
CREATE ERROR  
ACTIVATE ERROR  
SUSPEND ERROR  
MYCOMMAND ERROR  
LOCKGLORIN ERROR

and corresponds to the names of the following tables.

< msgno > is a message number, which is an index into the < msgtype > table.

< message > is the text of the message which can be found along with the message number in the message type table.

< number > is the number of the invalid parameter passed to an intrinsic (the message will read: PARAM # < number >) or is the parameter passed to the QUIT or QUITPROG intrinsic (the message will read: PARAM =).

Some examples of run-time messages are:

Examples:

ABORT :BIN.ED.MPE.%0.%12  
ERROR: INTRINSIC #62: BINARY  
RUN-TIME ERROR #5: PARAMETER ADDRESS VIOLATION. PARAM #1  
BINARY was called with an invalid byte address.

ABORT :0V.ED.MPE.%0.%177777  
PROGRAM ERROR #20: STACK OVERFLOW

The program was in an infinite loop doing a DUP instruction.

ABORT :PRIV.ED.MPE.%0.%3  
PROGRAM ERROR #6: PRIVILEGED INSTRUCTION

A return was made from a non-privileged segment to a privileged segment.

ABORT :QUIT.ED.MPE.%0.%1  
 PROGRAM ERROR #18; PROCESS QUIT.PARAM = 15

The program called QUIT Intrinsic with a parameter of 15.

ABORT :UF.ED.MPE.%0.%1  
 PROGRAM ERROR #29: STACK UNDERFLOW

The program was in an infinite loop doing a DEL instruction.

ABORT :EDITOR.PUB.SYS.%2.%7  
 ERROR: INTRINSIC #100: CREATE  
 CREATE ERROR #30: LOAD ERROR  
 LOADER ERROR #65: UNABLE TO OBTAIN CST ENTRIES



Nearly all CST entries were ALLOCATED and the program tried to create a process which required more CST's than were available.

ABORT :EDITOR.PUB.SYS.%2.%13  
 ERROR: INTRINSIC #104: ACTIVATE  
 ACTIVATE ERROR #21: ACTIVATION OF MAIN PROCESS NOT ALLOWED

The program tried to activate a non-existent process.

The following is a list of < msgtype > tables, the message number and text for each message found for each type of message:

Table 10-1. Program Error Table

MSGNO	MESSAGE	COMMENT
1	INTEGER OVERFLOW	} Logic error in the program.
2	FLOATING POINT OVERFLOW	
3	FLOATING POINT UNDERFLOW	
4	INTEGER DIVIDE BY ZERO	
5	FLOATING POINT DIVIDE BY ZERO	
6	PRIVILEGED INSTRUCTION	
7	ILLEGAL INSTRUCTION	
8	EXTENDED PRECISION OVERFLOW	
9	EXTENDED PRECISION UNDERFLOW	
10	EXTENDED PRECISION DIVIDE BY ZERO	
11	DECIMAL OVERFLOW	
12	INVALID ASCII DIGIT	
13	INVALID DECIMAL DIGIT	
14	INVALID WORD COUNT	
15	INVALID DECIMAL OPERAND LENGTH	
16	DECIMAL DIVIDE BY ZERO	
17	STT UNCALLABLE	



Table 10-1. Program Error Table (Continued)

MSGNO	MESSAGE	COMMENT
18	PROCESS QUIT.PARAM=<number>	} <number> is the value passed to the QUITPROG or QUIT intrinsic by the terminating process. (This value is output only if it is <i>not</i> zero.)
19	PROGRAM QUIT.PARAM=<number>	
20	STACK OVERFLOW	} Logic error in the program. Probably looping and adding to stack. May require larger MAXDATA when preparing program.
21	PROGRAM KILLED	} Program aborted from an external source.
22	INVALID STACK MARKER	} Possible hardware problem.
23	ADDRESS VIOLATION	
24	BOUNDS VIOLATION	
25	NON-RESPONDING MODULE	
26	DATA PARITY	
27	MEMORY PARITY	
28	SYSTEM PARITY	
29	STACK UNDERFLOW	} Logic error in program. Probably looping and popping stack.
30	CST VIOLATION	} Invalid CST or STT discovered by hardware. Explicit PCAL from TOS may have referenced non-existent CST or STT. May be bad program file.
31	STT VIOLATION	

Table 10-2. Intrinsic Table

MSGNO (INTRINSIC NO.)	MESSAGE (NAME)	MSGNO (INTRINSIC NO.)	MESSAGE (NAME)
1	FOPEN	15	FLOCK
2	FREAD	16	FUNLOCK
3	FWRITE	17	FRENAME
4	FUPDATE	18	FRELATE
5	FSPACE	19	FREADLABEL
6	FPOINT	20	FWRITELABEL
7	FREADDIR	21	PRINTFILEINFO
8	FCLOSE	22	IOWAIT
10	FCHECK	30	GETLOCRIN
11	FGETINFO	31	FRELOCRIN
12	FREADSEEK	32	LOCKLOCRIN
13	FCONTROL	33	UNLOCKLOCRIN
14	FSETMODE	34	LOCKGLORIN
		35	UNLOCKGLORIN
		40	TIMER

Table 10-2. Intrinsic Table (Continued)

MSGNO (INTRINSIC NO.)	MESSAGE (NAME)	MSGNO (INTRINSIC NO.)	MESSAGE (NAME)
42	PROCTIME	81	UNLOADPROC
45	PAUSE	82	INITUSLF
50	XARITRAP	83	ADJUSTSLF
51	ARITRAP	84	EXPANDUSLF
52	XLIBTRAP	99	DEBUG
53	XSYSTRAP	100	CREATE
54	XCONTRAP	102	KILL
55	RESETCONTROL	103	SUSPEND
56	CAUSEBREAK	104	ACTIVATE
60	TERMINATE	105	GETORIGIN
61	CTRANSLATE	106	MAIL
62	BINARY	107	SENDMAIL
63	ASCII	108	RECEIVEMAIL
64	READ,READX	109	FATHER
65	PRINT	110	GETPROCINFO
66	PRINTOP	112	GETPROCID
67	PRINTOREPLY	120	GETPRIORITY
68	COMMAND	130	GETDSEG
69	WHO	131	FREEDSEG
70	SEARCH	132	DMOVIN
71	MYCOMMAND	133	DMOVEOUT
72	SETJCW	134	ALTDSEG
73	GETJCW	135	DLSIZE
74	DBINARY	136	ZSIZE
75	DASCII	139	SWITCHDB
76	QUIT	191	PTAPE
77	STACKDUMP	200	GETPRIVMODE
78	SETDUMP	201	GETUSERMODE
79	RESETDUMP		
80	LOADPROC		

Table 10-3. Run-Time Error Table

	MSGNO	MESSAGE
Run-time errors are discovered by MPE performing parameter checking before attempting certain operations. These errors are caused by a logic error in the program.	1	ILLEGAL DB REGISTER
	2	ILLEGAL CAPABILITY
	3	OMITTED PARAMETER
	4	INCORRECT S REGISTER
	5	PARAMETER ADDRESS VIOLATION
	6	PARAMETER END ADDRESS VIOLATION
	7	ILLEGAL PARAMETER
	8	PARAMETER VALUE INVALID
	9	INCORRECT Q REGISTER

Table 10-4. File System Error Table

MSGNO	MESSAGE
0	END OF FILE
1	ILLEGAL DB REGISTER
2	ILLEGAL CAPABILITY
8	ILLEGAL PARAMETER VALUE
20	INVALID OPERATION
21	DATA PARITY ERROR
22	SOFTWARE TIME-OUT
23	END OF TAPE
24	UNIT NOT READY
25	NO WRITE-RING ON TAPE
26	TRANSMISSION ERROR
27	I/O TIME-OUT
28	TIMING ERROR OR DATA OVERRUN
29	SIO FAILURE
30	UNIT FAILURE
31	END OF LINE
32	SOFTWARE ABORT
33	DATA LOST
34	UNIT NOT ON-LINE
35	DATA-SET NOT READY
36	INVALID DISC ADDRESS
37	INVALID MEMORY ADDRESS
38	TAPE PARITY ERROR
39	RECOVERED TAPE ERROR
40	OPERATION INCONSISTENT WITH ACCESS TYPE
41	OPERATION INCONSISTENT WITH RECORD TYPE
42	OPERATION INCONSISTENT WITH DEVICE TYPE
43	WRITE EXCEEDS RECORD SIZE
44	UPDATE AT RECORD ZERO
45	PRIVILEGED FILE VIOLATION
46	OUT OF DISC SPACE
47	I/O ERROR ON FILE LABEL
48	INVALID OPERATION DUE TO MULTIPLE FILE ACCESS
49	UNIMPLEMENTED FUNCTION
50	NONEXISTENT ACCOUNT
51	NONEXISTENT GROUP
52	NONEXISTENT PERMANENT FILE
53	NONEXISTENT TEMPORARY FILE
54	INVALID FILE REFERENCE
55	DEVICE UNAVAILABLE
56	INVALID DEVICE SPECIFICATION
57	OUT OF VIRTUAL MEMORY
58	NO PASSED FILE
59	STANDARD LABEL VIOLATION
60	GLOBAL RIN UNAVAILABLE
61	OUT OF GROUP DISC SPACE
62	OUT OF ACCOUNT DISC SPACE
63	USER LACKS NON-SHARABLE DEVICE CAPABILITY
64	USER LACKS MULTI-RIN CAPABILITY
66	PLOTTER LIMIT SWITCH REACHED

Table 10-4. File System Error Table (Continued)

MSGNO	MESSAGE
67	PAPER TAPE ERROR
68	SYSTEM INTERNAL ERROR
69	ATTACHED ERROR
71	TOO MANY FILES OPEN
72	INVALID FILE NUMBER
73	BOUNDS VIOLATION
76	INPUT BUFFER ABSENT
77	NO-WAIT I/O PENDING
78	NO NO-WAIT I/O PENDING FOR ANY FILE
79	NO NO-WAIT I/O PENDING FOR SPECIFIED FILE
80	SPOOFLE SIZE EXCEEDS CONFIGURATION
81	NO "SPOOL" CLASS IN SYSTEM
82	INSUFFICIENT SPACE FOR SPOOFLE
83	I/O ERROR ON SPOOFLE
84	DEVICE UNAVAILABLE FOR SPOOFLE
85	OPERATION INCONSISTENT WITH SPOOLING
86	NONEXISTENT SPOOFLE
87	BAD SPOOFLE BLOCK
89	POWER FAILURE
90	EXCLUSIVE VIOLATION: FILE BEING ACCESSED
91	EXCLUSIVE VIOLATION: FILE ACCESSED EXCLUSIVELY
92	LOCKWORD VIOLATION
93	SECURITY VIOLATION
94	USER IS NOT CREATOR
95	BROKEN TERMINAL READ
96	DISC I/O ERROR
97	NO CONTROL-Y PIN
98	READ TIME OVERFLOW
99	BEGINNING OF TAPE ON BSR
100	DUPLICATE PERMANENT FILE NAME
101	DUPLICATE TEMPORARY FILE NAME
102	I/O ERROR ON DIRECTORY
103	PERMANENT FILE DIRECTORY OVERFLOW
104	TEMPORARY FILE DIRECTORY OVERFLOW
105	BAD VARIABLE BLOCK STRUCTURE
106	EXTENT SIZE EXCEEDS MAXIMUM
107	INSUFFICIENT SPACE FOR USER LABELS
108	DEFECTIVE FILE LABEL ON DISC
109	ILLEGAL CARRIAGE CONTROL
110	ATTEMPT TO SAVE PERMANENT FILE AS TEMPORARY
111	USER LACKS SAVE FILE CAPABILITY

Table 10-5. Loader Error Table

MSGNO	MESSAGE	COMMENT
20	ILLEGAL LIBRARY SEARCH	
21	UNKNOWN ENTRY POINT	
22	TRACE SUBSYSTEM NOT PRESENT	
23	STACK SIZE TOO SMALL	
24	MAXDATA TOO LARGE	MAXDATA must be no greater than 31,232
25	DATA SEGMENT TOO LARGE	
26	PROGRAM LOADED IN OPPOSITE MODE	A privileged program is currently loaded in the opposite PRIV/ NON-PRIV mode.
27	SL BINDING ERROR	
28	INVALID SYSTEM SL FILE	
29	INVALID PUBLIC SL FILE	
30	INVALID GROUP SL FILE	
31	INVALID PROGRAM FILE	
32	INVALID LIST FILE	
33	CODE SEGMENT TOO LARGE	System may be reconfigured by system supervisor/manager for larger code segment.
34	PROGRAM USES MORE THAN ONE EXTENT	Programs must be located in contiguous disc space. Build new program file with larger extent size.
35	DATA SEGMENT TOO LARGE	Data segment greater than 32,767 words, the hardware limitation.
36	DATA SEGMENT TOO LARGE	System may be reconfigured by system supervisor/manager for larger data segment.
37	TOO MANY CODE SEGMENTS	A program file can contain a maximum of 152 segments.
38	TOO MANY CODE SEGMENTS	System may be reconfigured by system supervisor/manager for more code segments.
39	ILLEGAL CAPABILITY	
40	TOO MANY PROCEDURES LOADED	
41	UNKNOWN PROCEDURE NAME	
42	INVALID PROCEDURE NUMBER	
43	ILLEGAL PROCEDURE UNLOAD	
50	UNABLE TO OPEN SYSTEM SL FILE	
51	UNABLE TO OPEN PUBLIC SL FILE	
52	UNABLE TO OPEN GROUP SL FILE	
53	UNABLE TO OPEN PROGRAM FILE	
54	UNABLE TO OPEN LIST FILE	
55	UNABLE TO CLOSE SYSTEM SL FILE	
56	UNABLE TO CLOSE PUBLIC SL FILE	
57	UNABLE TO CLOSE GROUP SL FILE	
58	UNABLE TO CLOSE PROGRAM FILE	
59	UNABLE TO CLOSE LIST FILE	

Table 10-5. Loader Error Table (Continued)

MSGNO	MESSAGE	COMMENT
60 61 62 63 64 65	EOF OR I/O ERROR ON SYSTEM SL FILE EOF OR I/O ERROR ON PUBLIC SL FILE EOF OR I/O ERROR ON GROUP SL FILE EOF OR I/O ERROR ON PROGRAM FILE EOF OR I/O ERROR ON LIST FILE UNABLE TO OBTAIN CST ENTRIES	System is loaded to capacity. Either a running program must terminate, or an ALLOCATED program or procedure not in use must be DEALLOCATED.
66 67 68	UNABLE TO OBTAIN PROCESS DST ENTRY UNABLE TO OBTAIN MAIL DATA SEGMENT UNABLE TO CREATE LOAD PROCESS	System is loaded and there are insufficient resources to create the load process.
70 71 72 73 74	SEGMENT TABLE OVERFLOW UNABLE TO OBTAIN SUFFICIENT DL STORAGE ATTIO ERROR UNABLE TO OBTAIN VIRTUAL MEMORY DIRECTORY I/O ERROR	



Table 10-5. Loader Error Table (Continued)

MSGNO	MESSAGE	COMMENT
75	PRINT I/O ERROR	
76	ILLEGAL DLSIZE	
80	PROGRAM ALREADY ALLOCATED	
81	ILLEGAL PROGRAM ALLOCATION	
82	PROGRAM NOT ALLOCATED	
83	ILLEGAL PROGRAM DEALLOCATION	
84	PROCEDURE ALREADY ALLOCATED	
85	ILLEGAL PROCEDURE ALLOCATION	
85	PROCEDURE NOT ALLOCATED	
87	ILLEGAL PROCEDURE DEALLOCATION	

Table 10-6. Create Error Table

MSGNO	MESSAGE	COMMENT
20	UNKNOWN SUBQUEUE NAME	
21	SUBQUEUE 'A' REQUESTED WITHOUT FROZEN STACK	
23	INSUFFICIENT CAPABILITY FOR NON-STANDARD SUBQUEUE	
24	UNKNOWN PORTION OF MASTER QUEUE	
25	INSUFFICIENT CAPABILITY FOR MASTER QUEUE	
26	ABSOLUTE PRIORITY REQUESTED WITHOUT CAPABILITY	
27	ILLEGAL PRIORITY CLASS SPECIFIED	
28	PRIORITY OMITTED WHILE FATHER PROCESS IN MASTER QUEUE	
29	PRIORITY RANK RESERVED TO SUPERVISOR CAPABILITY	
30	LOAD ERROR	Error occurred in loader. System is loaded and there are insufficient PCB's to load process.
31	LACK OF SYSTEM RESOURCE	
32	MAXIMUM ACCOUNT PRIORITY EXCEEDED	

Table 10-7. Activate Error Table

MSGNO	MESSAGE	COMMENT
20	ACTIVATION OF SYSTEM PROCESS NOT ALLOWED	Process may not exist.
21	ACTIVATION OF MAIN PROCESS NOT ALLOWED	



Table 10-8. Suspend Error Table

MSGNO	ERROR
20	INSUFFICIENT CAPABILITY

Table 10-9. Mycommand Error Table

MSGNO	ERROR
20	PARSED PARAM OF COMIMAGE >255 CHARACTERS

Table 10-10. Lockglorin Error Table

MSGNO	ERROR
20	INCORRECT PASSWORD FOR RIN
21	ONLY ONE RIN CAN BE LOCKED
22	RIN IS NOT ALLOCATED
23	RIN IS TOO LARGE FOR THE RIN TABLE
24	RIN IS NOT GLOBAL RIN

## USER MESSAGES

When your batch job or session receives a message from another user's job or session, that message appears in the following format:

*FROM/[jsname,] username.acctname/message*

*jsname*  
*username*  
*acctname* } The names of the transmitting job/session and user, and the name of the account under which they are running.

*message* The message.

As an example, if a user identified as BOB running a session named S, under an account named A, sends a message to you that he is changing the name of a file used frequently by both programs: you would see the following message:

FROM/S,BOB.A/FILE NAMED BAKER IS NOW RENAMED JONES.

## OPERATOR MESSAGES

When your batch job or session receives a message from the console operator, that message appears in one of two formats, depending on its degree of urgency. Urgent messages which pre-empt any form of input/output being conducted on the standard list device, appear in this format:

*WARN/message*

*message* is the message text.

Less serious messages used for normal communication between the operator and you do not pre-empt input/output in progress, and appear on the standard list device in this format:

*FROM/OPERATOR/message*

*message* is the message text.

## SYSTEM MESSAGES

Miscellaneous conditions that terminate or otherwise affect your job/session are reported through system messages, shown in table 10-11. These messages may appear, asynchronously, during the course of a running job/session on the standard list device.

Table 10-11. System Messages

### CAN NOT INITIATE NEW SESSIONS NOW

New sessions cannot be initiated due to one of the following problems:

1. Insufficient system resources to start job.
2. Session limit would be exceeded (see = LIMIT and = LOGOFF).
3. Requestor's input priority (INPRI =) is not greater than current <jobfence> (see = JOBFENCE).

*NOTE: System managers and system supervisors can bypass rejections due to 2 and 3, above, by supplying HIPRI on :HELLO command.*

\* { SESSION }  
  JOB                    ABORTED BY SYSTEM MANAGEMENT \*

The job/session has been aborted by the computer operator or system supervisor user through the appropriate command. An immediate log-off then takes place.

\* { SESSION }  
  JOB                    HAS EXCEEDED TIME LIMIT \*

The job/session has exceeded the time limit which was specified in the TIME=parameter of the JOB/HELLO command. An immediate log-off then takes place.

Table 10-11. System Messages (Continued)

<p>WARNING:                    PRIORITY = XXX</p> <p>The priority passed to the CREATE intrinsic resulted in a conflict with another process, and the priority then assigned was XXX instead of the requested value.</p> <p>LMAP NOT AVAILABLE</p> <p>An LMAP of the process being created, or program file being :RUN, is not available because the code segments are already loaded.</p> <p><b>**POWER FAIL**</b></p> <p>Power failure has occurred and automatic restart is in progress. It is possible that a character has been lost due to a transmission error when the power failure occurred.</p>
--

## FILE INFORMATION DISPLAY

In addition to Command Interpreter and run-time (abort) error messages, certain file input/output errors result in the output of a file information display. For files not yet opened, or for which the FOPEN intrinsic failed, this display appears as in the example below.

```

+-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
① → ! FILE NUMBER 5          IS UNDEFINED.          !
② → ! ERROR NUMBER: 2      RESIDUE: 101      (WORDS) !
③ → ! BLOCK NUMBER: 131072      NUMREC: 58      !
+-----+

```

In this display, the lines indicated show the following information.

Line	Content
1	A message stating why the file could not be opened.
2	The appropriate <i>error number</i> , relating to table 10-3 in this case. The <i>residue</i> , which is the number of words <i>not</i> transferred in an input/output request; since no such request applies in this case, this is <i>zero</i> .
3	The <i>block number</i> , indicating the physical record where the error was encountered. The <i>numrec</i> (number of logical records in the block).

For files that were open when a CCG (end-of-file error) or CCL (irrecoverable file error) condition code was returned, the file information display appears as shown in this example:



```

+-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
 ① → ! FILE NAME IS IN.PUB.TECHPUBS !
 ② → ! FOPTIONS: NEW,A,*FORMAL*,F,N,FEQ !
 ③ → ! AOPTIONS: INPUT,SREC,NOLOCK,DEF,BUFFER !
 ④ → ! DEVICE TYPE: 0 DEVICE SUBTYPE: 3 !
 ⑤ → ! LDEV: 17 DRT: 12 UNIT: 1 !
 ⑥ → ! RECORD SIZE: 256 BLOCK SIZE: 256 (BYTES) !
 ⑦ → ! EXTENT SIZE: 128 MAX EXTENTS: 8 !
 ⑧ → ! RECPTR: 0 RECLIMIT: 1023 !
 ⑨ → ! LOGCOUNT: 0 PHYSCOUNT: 0 !
 ⑩ → ! EOF AT: 0 LABEL ADDR: %02100060123 !
 ⑪ → ! FILE CODE: 0 ID IS MAC ULABELS: 0 !
 ⑫ → ! PHYSICAL STATUS: 1111000000000001 !
 ⑬ → ! ERROR NUMBER: 40 RESIDUE: 0 !
 ⑭ → ! BLOCK NUMBER: 0 NUMREC: 1 !
+-----+

```

The lines indicated show the following information:

Line	Content
1	The <i>file name</i> : in this case, the name is IN.PUB.TECHPUBS.
2	The <i>foptions</i> in effect, including: <ul style="list-style-type: none"> <li>Domain:      NEW = New file (as in this case).</li> <li>              SYS = System file domain.</li> <li>              JOB = Job temporary file domain.</li> <li>              ALL = System <i>and</i> job temporary file domain.</li> <li>Type:        A = ASCII File (as in this case).</li> <li>              B = Binary File.</li> <li>Default File Designator:   *FORMAL* = Actual file designator is same as formal file designator.</li> </ul>

Line	Contents
2 (Cont.)	<p>Record            F = Fixed length.  Format:            V = Variable length.                        U = Undefined length (as in this case).                        ? = Unknown format.</p> <p>Carriage          N = None (as in this case).  Control:           C = Carriage control character expected.</p> <p>File Equation    FEQ = :FILE allowed (as in this case).  Option:            DEQ = :FILE not allowed.</p>
3	<p>The <i>options</i> in effect, including:</p> <p>Access Type:    INPUT        = Read access (as in this case).                        OUTPUT       = Write access.                        OUTKEEP     = Write-only access, without deleting.                        APPEND       = Append access.                        IN/OUT       = Input and output access.                        UPDATE       = Update access.</p> <p>Multi-record    SREC = Single record access (as in this case).  Option:           MREC = Multi-record access.</p> <p>Dynamic         NOLOCK      = No locking permitted (as in this case).  Locking          LOCK         = Locking permitted.  Option:</p> <p>Exclusive       DEF         = Default specification (as in this case).  Access            EXC         = Exclusive access allowed.  Option:           SEA         = Semi-exclusive access allowed.                        SHR         = Sharable file.</p> <p>Buffering:      BUFFER = Automatic buffering.                        NOBUFF = Inhibit buffering (as in this case).</p>
4,5	<p>The <i>Device Type</i>, <i>Device Subtype</i>, <i>LDEV</i> (<i>Logical Device Number</i>), <i>DRT</i> (<i>Device Reference Table Entry Number</i>) and <i>Unit</i> of the device on which the file resides. (These are 0, 3, 17, 12, and 1 respectively, in this case.)</p>
6	<p>The <i>record size</i> and <i>block size</i> of the offending record, in <i>bytes</i>. (In this case, these are both specified as 256 bytes.)</p>
7	<p>The <i>extent size</i> (of the current extent) and the <i>max extents</i> (maximum number of extents) allowed this file.</p>
8	<p>The <i>recptr</i> (current record pointer) and <i>reclimit</i> (limit on number of records in the file).</p>

Line	Contents
9	The <i>logcount</i> (present count of logical records) and <i>physcount</i> (present count of physical records) in the file.
10	The <i>EOF at</i> (location of the current end-of-file) and the <i>label addr</i> (location of the header label of the file).
11	The <i>file code, id</i> (name of creating user), and <i>ulabels</i> (number of user-created labels) for the file.
12	The <i>physical status</i> of the file.
13	The <i>error number</i> and <i>residue</i> , as described under the abbreviated file information display format, above.
14	The <i>block number</i> and <i>numrec</i> , as described under the abbreviated file information display format, above.



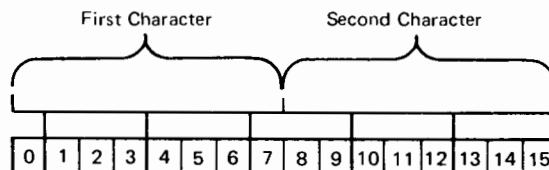
# ASCII CHARACTER SET

APPENDIX

A

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
A	040400	000101
B	041000	000102
C	041400	000103
D	042000	000104
E	042400	000105
F	043000	000106
G	043400	000107
H	044000	000110
I	044400	000111
J	045000	000112
K	045400	000113
L	046000	000114
M	046400	000115
N	047000	000116
O	047400	000117
P	050000	000120
Q	050400	000121
R	051000	000122
S	051400	000123
T	052000	000124
U	052400	000125
V	053000	000126
W	053400	000127
X	054000	000130
Y	054400	000131
Z	055000	000132
a	060400	000141
b	061000	000142
c	061400	000143
d	062000	000144
e	062400	000145
f	063000	000146
g	063400	000147
h	064000	000150
i	064400	000151
j	065000	000152
k	065400	000153
l	066000	000154
m	066400	000155
n	067000	000156
o	067400	000157
p	070000	000160
q	070400	000161
r	071000	000162
s	071400	000163
t	072000	000164
u	072400	000165
v	073000	000166
w	073400	000167
x	074000	000170
y	074400	000171
z	075000	000172
0	030000	000060
1	030400	000061
2	031000	000062
3	031400	000063
4	032000	000064
5	032400	000065
6	033000	000066
7	033400	000067
8	034000	000070
9	034400	000071
NUL	000000	000000
SOH	000400	000001
STX	001000	000002
ETX	001400	000003
EOT	002000	000004
ENQ	002400	000005

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
ACK	003000	000006
BEL	003400	000007
BS	004000	000010
HT	004400	000011
LF	005000	000012
VT	005400	000013
FF	006000	000014
CR	006400	000015
SO	007000	000016
SI	007400	000017
DLE	010000	000020
DC1	010400	000021
DC2	011000	000022
DC3	011400	000023
DC4	012000	000024
NAK	012400	000025
SYN	013000	000026
ETB	013400	000027
CAN	014000	000030
EM	014400	000031
SUB	015000	000032
ESC	015400	000033
FS	016000	000034
GS	016400	000035
RS	017000	000036
US	017400	000037
SPACE	020000	000040
!	020400	000041
"	021000	000042
#	021400	000043
\$	022000	000044
%	022400	000045
&	023000	000046
'	023400	000047
(	024000	000050
)	024400	000051
*	025000	000052
+	025400	000053
,	026000	000054
-	026400	000055
.	027000	000056
/	027400	000057
:	035000	000072
;	035400	000073
<	036000	000074
=	036400	000075
>	037000	000076
?	037400	000077
@	040000	000100
[	055400	000133
\	056000	000134
]	056400	000135
Δ	057000	000136
—	057400	000137
~	060000	000140
{	075400	000173
	076000	000174
}	076400	000175
~	077000	000176
DEL	077400	000177







# DISC FILE LABELS

APPENDIX

B

Whenever a disc file is created, MPE automatically supplies a file label in the first sector of the first extent occupied by that file. Such labels always appear in the format described below. (User-supplied labels, if present, are located in the sectors immediately following the MPE file label.) The contents of a label may be listed by using the `:LISTF -1` command described in the *MPE Commands Reference Manual*.

Words		Contents
0-3		Local file name.
4-7		Group name.
8-11		Account name.
12-15		User name of file creator.
16-19		File lockword.
20-21		File security matrix.
22	(Bits 0:15)	Not used.
	(Bit 15:1)	File secure bit: If 1, file secured. If 0, file released.
23		File creation date
24		Last access date.
25		Last modification date.
26		File code.
27		File control block vector.
28	(Bit 0:1)	Store Bit. (If on, :STORE or :RESTORE, in progress.)
	(Bit 1:1)	Restore Bit. (If on, :RESTORE in progress.)
	(Bit 2:1)	Load Bit. (If on, program file is loaded.)
	(Bit 3:1)	Exclusive Bit. (If on, file is opened with exclusive access.)
	(Bits 4:4)	Device sub-type.
	(Bits 8:6)	Device type.
	(Bit 14:1)	File is open for write.
	(Bit 15:1)	File is open for read.

Words		Contents
29	(Bits 0:8)	Number of user labels written.
	(Bits 8:8)	Number of user labels available.
30-31		File limit in blocks.
32-33		Not used.
34		File label check sum (used for error detection).
35		Cold-load identity.
36		Foptions specifications.
37		Logical record size (in negative bytes).
38		Block size (in words).
39	(Bits 0:8)	Sector offset to data.
	(Bits 8:3)	Not used.
	(Bits 11:5)	Number of extents-1.
40		Last extent size in sectors.
41		Extent size in sectors.
42-43		Number of logical records in file.
44-45		First extent descriptor.
46-107		Remaining extent descriptors (32 maximum).
108-123		Not used.
124-127		Device class name.

#### Note

An extent descriptor (words 44 through 107 above) is a double word. The first byte contains the volume table index of the volume in which the extent resides; the remaining three bytes of the double word extent descriptor contain the first sector number of the extent.

# END-OF-FILE INDICATION

APPENDIX

C

The end-of-file indication will be returned by the card reader and tape drivers under conditions specified by the initiators of read requests. The type of requests are as follows:

Type	Class of end-of-file
A	All records that begin with a colon (:).
B	All records that contain, starting in the first byte, :EOD, :EOJ, :JOB and :DATA (See Note.)
E	Hardware-sensed end-of-file.



*NOTE: If the word count is less than 3 or the byte count is less than 6, then Type B reads are converted to Type A reads.*

In utilizing the card/tape devices as files via the file system, the following types are assigned:

File Specified	Type
\$STDIN	Type A.
\$STDINX	Type B.
Dev=CARD/TAPE	Type B, if device job/data accepting. Type E, if device not job/data accepting.

Any subsequent requests initiated by the driver following sensing of an end-of-file condition will be rejected with an end-of-file indication.

When reading from an unlabeled tape file, the request encountering a tape mark will respond with an end-of-file indication but succeeding requests will be allowed to continue to read data past the tape mark. Under these conditions, it is the responsibility of the caller to protect against the occurrence of data beyond an end-of-file and to prevent reading off the end of the reel.



- Aborting a process, 4-20
- Aborting a program, 4-20
- Accessing files, 3-7, 3-14
- Accessing files already in use, 3-10
- Access-mode options for files, 3-77
- Access mode, user, 4-10
- Acquiring global RIN's, 6-2
- Acquiring local RIN's, 6-8
- ACTIVATE errors, 10-11
- ACTIVATE intrinsic
  - specifications, 2-4
  - usage, 8-10
- Activating an extra data segment, 8-10
- Activating processes, 7-3
- Actual file designator, 3-7
- ADJUSTUSLF intrinsic, 2-6
- Allocation of devices, 3-23
- Allocating a terminal, 5-24
- ALTDSEG intrinsic
  - specifications, 2-8
  - usage, 8-15
- aoptions*
  - bit summary, 2-51
  - description, 2-61
- Arithmetic traps, 4-30
- ARITRAP intrinsic
  - specifications, 2-10
  - usage, 4-30
- Arrays, searching, 4-3
- ASCII intrinsic
  - specifications, 2-11
  - usage, 4-10
- AS subqueue, 9-7
- Attributes, user, 4-10
- Available file table, 3-14
- Avoiding deadlocks, 7-13
  
- Back referencing files, 3-8
- BINARY intrinsic
  - specifications, 2-13
  - usage, 4-13
- Block factor, 2-65, 3-3
- Blocks, 3-2
- Block size, 3-4
- Block transfers, 5-22
- Bounds check, 1-11
- BS subqueue, 9-7
- Buffering, 3-6
  
- Calculating disc space, 3-2
- Calendar date, 4-44
- CALENDAR intrinsic
  - specifications, 2-14
  - usage, 4-44
- Calling intrinsics from other languages, 1-10
- Calling intrinsics from SPL, 1-2
- Card reader, 5-3
- Carriage control
  - byte, 2-70
  - characters, 5-1
  - codes, 5-7
  - directives, 2-89
  - summary, 2-90
- Carriage-return characters, 5-2
- CAUSEBREAK intrinsic
  - specifications, 2-15
  - usage, 4-19
- Central processor run-time, 4-44
- Changing DL to DB area size, 4-22
- Changing input echo facility, 5-11
- Changing size of an extra data segment, 8-15
- Changing terminal characteristics, 5-10
- Changing terminal speed, 5-10
- Changing Z to DB area size, 4-27
- Circular subqueues, 9-5
- Classification of devices, 3-22
- CLOCK intrinsic
  - specifications, 2-16
  - usage, 4-44
- Closing a new file as a permanent file, 3-38
- Closing a new file as a temporary file, 3-36
- Closing files, 3-35
- Closing magnetic tape files, 3-69
- Code segments, 8-1
- Collecting mail, 7-12
- COMMAND intrinsic
  - specifications, 2-17
  - usage, 4-9
- Command parameters, formatting, 4-4
- Commercial instruction traps, 4-31
- Communication
  - inter-process, 4-44, 7-10
  - process-to-process, 7-2
- Condition codes
  - definitions, 1-10
  - file system, 3-10
- Console Operator, 4-18
- Control-Y traps, 4-38
- Converting characters from EDCDIC to ASCII and ASCII to EBCDIC, 4-13
- Converting numbers
  - from ASCII to binary, 4-13
  - from ASCII to EBCDIC, 4-13
  - from binary to ASCII, 4-10
  - from EBCDIC to ASCII, 4-13
- CREATE errors, 10-11
- CREATE intrinsic
  - specifications, 2-18
  - usage, 7-3
- Creating an extra data segment, 8-2
- Creating processes, 7-3
- CS subqueue, 9-7
- CTranslate intrinsic

- specifications, 2-23
- usage, 4-13
- Current time, 4-44
- DASCII intrinsic
  - specifications, 2-25
  - usage, 4-10
- Data segments, 8-1
- Data information, 4-42
- DBINARY intrinsic
  - specifications, 2-27
  - usage, 4-13
- DB pointer, 9-5
- Deadlocks, 7-13
- Declaring access-mode options for files, 3-77
- Defining line-termination characters for terminal input, 5-20
- Deleting an extra data segment, 8-15
- Deleting processes, 7-8
- Determining father process, 7-14
- Determining interactive and duplicative file pairs, 3-78
- Determining process priority and state, 7-15
- Determining son processes, 7-15
- Determining source of process activation, 7-14
- Determining user's access mode and attributes, 4-10
- Device access, 3-4
- Device allocation, 3-23
- Device characteristics, 5-1
- Device-dependent restrictions on files, 3-21
- Devices, 3-6
- Devices, classification of, 3-22
- Direct-access file reading, 3-47, 3-49
- Direct-access file writing, 3-49
- Disabling traps, 4-29
- Disc space, 3-2
- DLSIZE intrinsic
  - specifications, 2-28
  - usage, 4-22
- DL to DB area, 4-22
- DMOVIN intrinsic
  - specifications, 2-30
  - usage, 8-15
- DMOVOUT intrinsic
  - specifications, 2-32
  - usage, 8-15
- Domains, file, 3-6
- DS subqueue, 9-7
- Duplex mode, 5-11
- Duplicative file pairs, 3-78
- Dynamic loading and unloading of library procedures, 4-2
- Enabling and disabling binary transfers, 5-21
- Enabling and disabling line deletion
  - echo suppression, 5-23
- Enabling and disabling parity checking, 5-14
- Enabling and disabling subsystem break function, 5-14
- Enabling and disabling system break function, 5-13
- Enabling and disabling tape-mode option, 5-15
- Enabling and disabling terminal input timer, 5-16
- Enabling and disabling user block transfers, 5-22
- Enabling traps, 4-29

- End-of-file indication, 5-6
- End-of-file marks on magnetic tape, 3-67
- Entering non-privileged mode, 9-5
- Entering privileged mode, 9-3
- Error-check procedure, 3-43
- Errors
  - ACTIVATE errors, 10-11
  - condition code, 1-10, 3-10
  - CREATE errors, 10-11
  - file information display, 10-14
  - file system errors, 10-8
  - intrinsic errors, 10-6
  - loader errors, 10-9
  - LOCKGLORIN errors, 10-12
  - MYCOMMAND errors, 10-12
  - obtaining file error information, 3-65
  - operator messages, 10-13
  - program errors, 10-5
  - run-time errors, 10-7
  - run-time messages, 10-2
  - SUSPEND errors, 10-12
  - system messages, 10-13
  - user messages, 10-12
  - writing an error-check procedure, 3-43
- ES subqueue, 9-7
- Executing MPE commands programmatically, 4-9
- EXPANDUSLF intrinsic, 2-34
- Extended precision floating-point traps, 4-31
- Extents, 3-2
- Extra data segment, 8-2
- FATHER intrinsic
  - specifications, 2-36
  - usage, 7-14
- Father process, 7-3
- FCARD intrinsic
  - specifications, 2-36A
  - usage, 5-28
- FCHECK intrinsic
  - specifications, 2-37
  - usage, 3-65
- FCLOSE intrinsic
  - specifications, 2-41
  - usage, 3-35
- FCLOSE with magnetic tape files, 3-69
- FCONTROL intrinsic
  - specifications, 2-44
  - usage, 3-76, 5-1
- FGETINFO intrinsic
  - specifications, 2-48
  - usage, 3-63
- File characteristics, 3-2
- File control operations, 3-76
- File designators, 3-7
- File-device relationships, 3-6
- File domains, 3-6
- File error information, 3-65
- File information display, 10-14
- File label, 3-7
- File management system, 3-1
- File marks on magnetic tape, 3-67
- File numbers, 3-10

## Files

- access information, 3-63
  - accessing, 3-7
  - back referencing, 3-8
  - block factor, 3-3
  - blocks, 3-2
  - block size, 3-4
  - buffered, 2-63
  - characteristics, 3-2
  - closing, 3-35
  - condition codes, 3-10
  - control, 3-76
  - declaring access-mode options, 3-77
  - designators, 3-7
  - device relationships, 3-6
  - disc, 3-2
  - domains, 3-6
  - duplicative pairs, 3-78
  - error-check procedure, 3-43
  - error information, 3-65
  - extents, 3-2
  - file information display, 10-14
  - file management system, 3-1
  - fixed-length records, 3-3
  - how to use, 3-14
  - interactive pairs, 3-78
  - label, 3-7
  - locking and unlocking, 3-52
  - logical records, 3-2
  - magnetic tape, 3-65
  - multiple access, 3-11
  - \$NEWPASS, 3-8
  - non-sharable devices, 3-13
  - no-wait I/O, 3-57
  - \$NULL, 3-8
  - numbers, 3-10
  - \$OLDPASS, 3-9
    - opening, 3-24
  - permanent, 3-6
  - physical records, 3-2
  - pointer, 3-77
  - reading, 3-43, 3-47
  - record formats, 3-3
  - records, 3-2
  - renaming, 3-40
  - sectors, 3-2
  - spooling, 3-20
  - \$STDIN, 3-7
  - \$STDINX, 3-8
  - \$STDLIST, 3-8
  - system defined, 3-7
  - temporary, 3-6
  - types, 3-7
  - undefined-length records, 3-3
  - unlocking, 3-52
  - updating, 3-54
  - user labels, 3-59
  - using, 3-14
  - variable-length records, 3-3
  - writing, 3-46, 3-49
- Files on non-sharable devices, 3-13
- File system condition codes, 3-10
- File system errors, 10-8
- File types, 3-7
- Fixed-length records, 3-3
- FLOCK intrinsic
  - specifications, 2-56
  - usage, 3-52
- FOPEN intrinsic
  - specifications, 2-58
  - usage, 3-24
- foptions*
  - bit summary, 2-49
  - description, 2-58
- Formal file designator, 3-7
- Formatting command parameters, 4-4
- FPOINT intrinsic
  - specifications, 2-67
  - usage, 3-77
- FREAD and FWRITE for \$STDIN and \$STDLIST, 3-32
- FREAD for magnetic tape files, 3-67
- FREAD intrinsic
  - specifications, 2-70
  - usage, 3-43
- FREADDIR intrinsic
  - specifications, 2-72
  - usage, 3-47
- FREADLABEL intrinsic
  - specifications, 2-74
  - usage, 3-63
- FREADSEEK intrinsic
  - specifications, 2-75
  - usage, 3-49
- FREEDSEG intrinsic
  - specifications, 2-76
  - usage, 8-15
- Freeing local RIN's, 6-9
- FREELOCRIN intrinsic
  - specifications, 2-77
  - usage, 6-9
- FRELATE intrinsic
  - specifications, 2-78
  - usage, 3-78
- FRENAME intrinsic
  - specifications, 2-80
  - usage, 3-77
- FSETMODE intrinsic
  - specifications, 2-82
  - usage, 3-77
- FSPACE intrinsic
  - specifications, 2-84
  - usage, 3-75
- Functional return, 2-2
- FUNLOCK intrinsic
  - specifications, 2-85
  - usage, 3-54
- FUPDATE intrinsic
  - specifications, 2-86
  - usage, 3-54
- FWRITE and FREAD for \$STDLIST and \$STDIN, 3-32
- FWRITE for magnetic tape files, 3-65
- FWRITE intrinsic
  - specifications, 2-87
  - usage, 3-46



FWRITEDIR intrinsic  
   specifications, 2-91  
   usage, 3-49  
 FWRITELABEL intrinsic  
   specifications, 2-93  
   usage, 3-59

GETDSEG intrinsic  
   specifications, 2-94  
   usage, 8-6  
 GETJCW intrinsic  
   specifications, 2-96  
   usage, 4-45  
 GETLOCRIN intrinsic  
   specifications, 2-97  
   usage, 6-8  
 GETORIGIN intrinsic  
   specifications, 2-98  
   usage, 7-14  
 GETPRIORITY intrinsic  
   specifications, 2-99  
   usage, 7-13  
 GETPRIVMODE intrinsic  
   specifications, 2-100  
   usage, 9-3  
 GETPROCID intrinsic  
   specifications, 2-101  
   usage, 7-15  
 GETPROCINFO intrinsic  
   specifications, 2-102  
   usage, 7-15  
 :GETRIN command, 6-2  
 GETUSERMODE intrinsic  
   specifications, 2-104  
   usage, 9-5  
 Global RIN's, 6-2

Hand-shaking arrangement, 6-1  
 How to use files, 3-14

INITUSLF intrinsic, 2-105  
 Input echo facility, 5-11  
 Input/output devices, 4-16  
 Input/output files, 3-9  
 Interactive file pairs, 3-78  
 Inter-job level RIN's, 6-2  
 Internal operations for file accessing, 3-14  
 Inter-process communication, 4-44, 7-10  
 Inter-process level RIN's, 6-6  
 Inter-record gap, 3-71  
 Intrinsic errors, 10-6  
 Intrinsic
 

- ACTIVATE, 2-4, 8-10
- ADJUSTUSLF, 2-6
- ALTDSEG, 2-8, 8-15
- ARITRAP, 2-10, 4-30
- ASCII, 2-11, 4-10
- BINARY, 2-13, 4-13
- CALENDAR, 2-14, 4-44

 calling from other languages, 1-10  
 calling from SPL, 1-2

CAUSEBREAK, 2-15, 4-19  
 CLOCK, 2-16, 4-44  
 COMMAND, 2-17, 4-9  
 CREATE, 2-18, 7-3  
 CTRANSLATE, 2-23, 4-13  
 DASCII, 2-25, 4-10  
 DBINARY, 2-27, 4-13  
 declaration, 1-2  
 definition, 1-1  
 DLSIZE, 2-28, 4-22  
 DMOVIN, 2-30, 8-15  
 DMOVOUT, 2-32, 8-15  
 error definitions, 1-10  
 error messages, 10-1  
 EXPANDUSLF, 2-34  
 FATHER, 2-36, 7-14  
 FCARD, 2-36A, 5-28  
 FCHECK, 2-37, 3-65  
 FCLOSE, 2-41, 3-35  
 FCONTROL, 2-44, 3-76, 5-1  
 FGETINFO, 2-48, 3-63  
 FLOCK, 2-56, 3-52  
 FOPEN, 2-58, 3-24  
 FPOINT, 2-69, 3-77  
 FREAD, 2-70, 3-43  
 FREADDIR, 2-72, 3-47  
 FREADLABEL, 2-74, 3-63  
 FREADSEEK, 2-75, 3-49  
 FREEDSEG, 2-76, 8-15  
 FREELOCRIN, 2-77, 6-9  
 FRELATE, 2-78, 3-78  
 FRENAME, 2-80, 3-77  
 FSETMODE, 2-82, 3-77  
 FSPACE, 2-84, 3-75  
 FUNLOCK, 2-85, 3-54  
 FUPDATE, 2-86, 3-54  
 FWRITE, 2-87, 3-46  
 FWRITEDIR, 2-91, 3-49  
 FWRITELABEL, 2-93, 3-59  
 GETDSEG, 2-94, 8-6  
 GETJCW, 2-96, 4-45  
 GETLOCRIN, 2-97, 6-8  
 GETORIGIN, 2-98, 7-14  
 GETPRIORITY, 2-99, 7-13  
 GETPRIVMODE, 2-100, 9-3  
 GETPROCID, 2-101, 7-15  
 GETPROCINFO, 2-102, 7-15  
 GETUSERMODE, 2-104, 9-5  
 INITUSLF, 2-105  
 IODONTWAIT, 2-106A, 3-59  
 IOWAIT, 2-107, 3-57  
 KILL, 2-109 7-8  
 LOADPROC, 2-110, 4-2  
 LOCKGLORIN, 2-111, 6-3  
 LOCKLOCRIN, 2-113, 6-8  
 MAIL, 2-115, 7-10  
 MYCOMMAND, 2-117, 4-4  
 PAUSE, 2-120, 4-19  
 PRINT, 2-121, 4-16  
 PRINTFILEINFO, 2-122, 3-43  
 PRINTOP, 2-123, 4-18  
 PRINTOPREPLY, 2-124, 4-18

PROCTIME, 2-126, 4-44  
 PTAPE, 2-127, 5-21  
 purposes, 1-1  
 QUIT, 2-128, 4-20  
 QUITPROG, 2-129, 4-20  
 READ, 2-130, 4-16  
 READX, 2-131, 4-16  
 RECEIVEMAIL, 2-132, 7-12  
 RESETCONTROL, 2-134, 4-40  
 SEARCH, 2-135, 4-3  
 SENDMAIL, 2-136, 7-11  
 SETJCW, 2-138, 4-45  
 summary, 1-3  
 SUSPEND, 2-139, 7-8  
 SWITCHDB, 2-140, 9-5  
 TERMINATE, 2-141, 4-20  
 TIMER, 2-142, 4-42  
 types, 2-2  
 UNLOADPROC, 2-143, 4-3  
 UNLOCKGLORIN, 2-144, 6-3  
 UNLOCKLOCRI, 2-145, 6-8  
 WHO, 2-146, 4-10  
 XARITRAP, 2-149, 4-32  
 XCONTRAP, 2-151, 4-41  
 XLIBTRAP, 2-152, 4-35  
 XSYSTRAP, 2-153, 4-36  
 ZSIZE, 2-154, 4-27  
 IOWAIT intrinsic  
   specifications, 2-107  
   usage, 3-57  
 Issuing FREAD and FWRITE calls for  
 \$STDIN and \$STDLIST, 3-32

Job main process, 7-1  
 Job or session file domains, 3-6  
 Job/session input/output devices, 4-16  
 Job temporary file directory, 3-17  
 Julian calendar, 8-8

Keys, terminal, 5-9  
 KILL intrinsic  
   specifications, 2-109  
   usage, 7-8

Library procedures, 4-2  
 Library traps, 4-34  
 Linear subqueue, 9-5  
 Line deletion echo suppression, 5-23  
 Line printer, 5-3  
 Line printer and terminal carriage-control codes, 5-6  
 Line-termination characters for terminal input, 5-20  
 Loader errors, 10-9  
 Loading library procedures, 4-2  
 LOADPROC intrinsic  
   specifications, 2-110  
   usage, 4-2  
 Local RIN's, 6-6  
 Locking and unlocking files, 3-52  
 Locking and unlocking global RIN's, 6-3

JAN 1977



Locking and unlocking local RIN's, 6-8  
 LOCKGLORIN errors, 10-12  
 LOCKGLORIN intrinsic  
   specifications, 2-111  
   usage, 6-3  
 LOCKLOCRI intrinsic  
   specifications, 2-113  
   usage, 6-8  
 Logical index number, 8-2  
 Logical record pointer, 3-77  
 Logical records, 3-2

Magnetic tape considerations for files, 3-65  
 Magnetic tape unit, 5-3  
 Mailbox, 7-10  
 MAIL intrinsic  
   specifications, 2-115  
   usage, 7-10  
 Master queue, 9-5  
 Messages  
   operator, 10-13  
   run-time, 10-2  
   system, 10-13  
   user, 10-12  
 Moving the DB pointer, 9-5  
 Multi-access, 3-6  
 Multiple access of files, 3-11  
 Multiple RIN optional capability, 6-1  
 MYCOMMAND errors, 10-12  
 MYCOMMAND intrinsic  
   specifications, 2-117  
   usage, 4-4

New files, 3-8  
 \$NEWPASS, 3-8  
 Non-sharable device access, 3-6  
 Non-sharable devices, 3-13  
 No-wait I/O, 3-57  
 \$NULL, 3-8

Obtaining file access information, 3-63  
 Obtaining file error information, 3-65  
 Obtaining process run time, 4-44  
 Obtaining system timer information, 4-42  
 Obtaining terminal output speed, 5-26  
 Obtaining terminal type information, 5-25  
 Obtaining the calendar date, 4-44  
 Obtaining the current time, 4-44  
 Old files, 3-9  
 \$OLDPASS, 3-9  
 Opening a file on a device other than disc, 3-29  
 Opening a new disc file, 3-24  
 Opening an old disc file, 3-27  
 Opening files, 3-24  
 Opening \$STDIN, 3-32  
 Opening \$STDLIST, 3-34  
 Operator messages, 10-13  
 Operator's Console, 4-18  
 Optical Mark Reader, 5-28  
 Optional capabilities

I-5

- data segment management, 8-1
- definitions, 1-12
- multiple resource, 6-1
- privileged mode, 7-1
- process handling, 7-1
- Optional parameters, 1-7
- Option variable, 1-7, 2-1
- Organization of user processes, 7-1

Paper tape punch, 5-3

Paper tape reader, 5-1

Paper tapes, 5-21

Parameters

- definition, 1-7
- optional, 1-7
- option variable, 1-7
- passing by reference, 1-7
- passing by value, 1-7
- positional, 1-7
- required, 1-8
- using numeric values, 1-8

Parity checking, 5-14

Parity, setting, 5-23

Passing parameters, 1-7

PAUSE intrinsic

- specifications, 2-120
- usage, 4-19

Permanent files, 3-6

Permanently privileged programs, 9-1

Physical records, 3-2

PIN, 7-1

Positional parameters, 1-7

Pre-defined files, 3-8

PRINT intrinsic

- specifications, 2-121
- usage, 4-16

PRINTFILEINFO intrinsic

- specifications, 2-122
- usage, 3-43

Printing reader/punch, 5-3

PRINTOP intrinsic

- specifications, 2-123
- usage, 4-18

PRINTOPREPLY intrinsic

- specifications, 2-124
- usage, 4-18

Private data area, 7-1

Privileged programs, 9-1

Privileged mode capability, 9-1

Procedures, 1-1

Procedure type, 2-2

Process activation, 7-14

Process break, 4-19

Process control block extension, 3-14

Processes

- aborting, 4-20
- activating, 7-3
- avoiding deadlocks, 7-13
- breaking, 4-19
- creating, 7-3
- deleting, 7-8

- description, 7-1
- father, 7-3
- identification number, 7-1
- inter-process communication, 4-44
- mail, 7-10
- organization, 7-2
- priority, 7-15
- process-handling capability, 7-1
- rescheduling, 7-13
- run time, 4-44
- scheduling, 9-5
- son, 7-3
- state, 7-15
- substate, 7-2
- suspending, 4-19, 7-8
- terminating, 4-20

Process-handling capability, 7-1

Process identification number, 7-1

Process priorities, 7-15

Process run time, 4-44

Process states, 7-15

Process substates, 7-2

Process-to-process communication, 7-2

PROCTIME intrinsic

- specifications, 2-126
- usage, 4-44

Program errors, 10-5

Program label, 7-1

Programmatic execution of MPE commands, 4-9

PTAPE intrinsic

- specifications, 2-127
- usage, 5-21

Queues, 9-5

QUIT intrinsic

- specifications, 2-128
- usage, 4-20

QUITPROG intrinsic

- specifications, 2-129
- usage, 4-20

Reading a file in direct-access mode, 3-47

Reading a file in sequential order, 3-43

Reading input from \$STDIN and \$STDINX, 4-16

Reading magnetic tape files, 3-67

Reading paper tapes without X-OFF control, 5-27

Reading the terminal input timer, 5-19

Reading user file labels, 3-59

READ intrinsic

- specifications, 2-130
- usage, 4-16

READX intrinsic

- specifications, 2-131
- usage, 4-16

Receiving mail, 7-12

RECEIVEMAIL intrinsic

- specifications, 2-132
- usage, 7-12

Record formats, 3-3

Records, 3-2

- Releasing global RIN's, 6-3
- Renaming a file, 3-40
- Requesting a process break, 4-17
- Requesting a reply from the Operator's Console, 4-18
- Required parameters, 1-8
- Rescheduling processes, 7-13
- RESETCONTROL intrinsic
  - specifications, 2-134
  - usage, 4-40
- Resetting the logical record pointer, 3-77
- Resource identification number, 6-1
- Resource management, 6-1
- Resources, 6-1
- Returns, intrinsic, 2-2
- RIN, 6-1
- Run-time errors, 10-7
- Run-time messages, 10-2

- Scheduling processes, 9-5
- Searching arrays, 4-3
- SEARCH intrinsic
  - specifications, 2-135
  - usage, 4-3
- Sectors, 3-2
- Segmenter driver, 7-6
- Segmenter subsystem, 7-6
- Segments
  - activating, 8-10
  - changing size of extra data segment, 8-15
  - code segments, 8-1
  - creating an extra data segment, 8-2
  - data segment management capability, 8-1
  - data segments, 8-1
  - deleting, 8-15
  - description, 8-1
  - logical index number, 8-2
  - stack segment, 8-1
  - transferring data from extra data segment to stack, 8-15
  - transferring data from stack to extra data segment, 8-15
- Sending mail, 7-11
- SENDMAIL intrinsic
  - specifications, 2-136
  - usage, 7-11
- Sequential access file reading, 3-43
- Sequential access file writing, 3-46
- Session main process, 7-1
- SETJCW intrinsic
  - specifications, 2-138
  - usage, 4-45
- Setting parity, 5-23
- Setting terminal type, 5-25
- Setting unedited terminal mode, 5-26
- Son process, 7-3
- Source of file characteristics, 3-17
- Source of process activation, 7-14
- Spacing on disc or tape files, 3-75
- Special terminal keys, 5-9
- SPL
  - calling intrinsics from, 1-2
  - description, 1-1
- Split stack, 1-11, 2-3
- Spooling, 3-20
- Stack
  - changing size, 4-22
  - sizes, 4-22
  - split stack, 1-11, 2-3
- Stack segment, 8-1
- Standard traps, 4-31
- \$STDIN, 3-7
- \$STDINX, 3-8
- \$STDLIST, 3-8
- Subqueues, 9-5
- Substates, 7-2
- Subsystem break function, 5-14
- SUSPEND errors, 10-12
- Suspending processes, 7-8
- SUSPEND intrinsic
  - specifications, 2-139
  - usage, 7-8
- Suspending the calling process, 4-19
- SWITCHDB intrinsic
  - specifications, 2-140
  - usage, 9-5
- System break function, 5-13
- System defined files, 3-7
- System file domain, 3-6
- System messages, 10-13
- System procedures, 1-1
- Systems Programming Language
  - calling intrinsics from, 1-2
  - description, 1-1
- System timer, 4-42
- System traps, 4-35

- Tape-mode option, 5-15
- Temporarily privileged programs, 9-2
- Temporary files, 3-6
- Terminal and line printer carriage-control codes, 5-6
- Terminal input timer, 5-16
- Terminals, 5-8
- Terminal speed, 5-10
- Terminating a process, 4-20
- TERMINATE intrinsic
  - specifications, 2-141
  - usage, 4-20
- Testing mailbox status, 7-10
- Time and date intrinsics, 4-42
- TIMER intrinsic
  - specifications, 2-142
  - usage, 4-42
- Transferring data from an extra data segment to the stack, 8-15
- Transferring data from the stack to an extra data segment, 8-15
- Translating characters from EBCDIC to ASCII and ASCII to EBCDIC, 4-13
- Transmitting program input/output from job/session input/output devices, 4-16
- Traps
  - arithmetic, 4-30

- commercial instruction, 4-32
  - Control-Y, 4-38
  - enabling and disabling, 4-29
  - extended precision floating-point, 4-31
  - library, 4-34
  - standard, 4-31
  - system, 4-35
- Types of files, 3-7
- Types of procedures, 2-2

- Undefined-length records, 3-3
- Unloading library procedures, 4-2
- UNLOADPROC intrinsic
  - specifications, 2-143
  - usage, 4-3
- UNLOCKGLORIN intrinsic
  - specifications, 2-144
  - usage, 6-3
- Unlocking files, 3-52
- Unlocking global RIN's, 6-3
- Unlocking local RIN's, 6-8
- UNLOCKLOCRIN intrinsic
  - specifications, 2-145
  - usage, 6-8
- Updating a file, 3-54
- Updating magnetic tape files, 3-69
- User block transfers, 5-22
- User file labels, 3-59
- User messages, 10-12
- User pre-defined files, 3-8
- User processes, 7-2
- User's access mode, 4-10
- User's attributes, 4-10
- User's stack segment, 8-1
- Using disc space efficiently, 3-4
- Using numeric values as parameters, 1-8
- Using the FCARD intrinsic
  - to operate the HP7260A Optical Mark Reader, 5-28
- Utility functions of intrinsics, 4-1

- Variable-length records, 3-3
- Virtual device directory, 3-13

- WHO intrinsic
  - specifications, 2-146
  - usage, 4-10
- Writing a file system error-check procedure, 3-43
- Writing on a magnetic tape file, 3-72
- Writing output to \$TDLIST, 4-18
- Writing output to the Operator's Console, 4-18
- Writing records into a file in direct-access mode, 3-49
- Writing records into a file in sequential order, 3-46
- Writing to magnetic tape files, 3-65
- Writing user file labels, 3-59

- XARITRAP intrinsic
  - specifications, 2-149
  - usage, 4-32
- XCONTRAP intrinsic
  - specifications, 2-151
  - usage, 4-41
- XLIBYRAP intrinsic
  - specifications, 2-152
  - usage, 4-35
- X-OFF control, 5-21
- XSYSTRAP intrinsic
  - specifications, 2-153
  - usage, 4-36

- ZSIZE intrinsic
  - specifications, 2-154
  - usage, 4-27
- Z to DB area, 4-27

Part No. 30000-90010  
Printed in U.S.A. 2/77  
Updates through #2 incorporated Feb 1977



Sales and service from 172 offices in 65 countries.  
5303 Stevens Creek Blvd., Santa Clara, California 95050