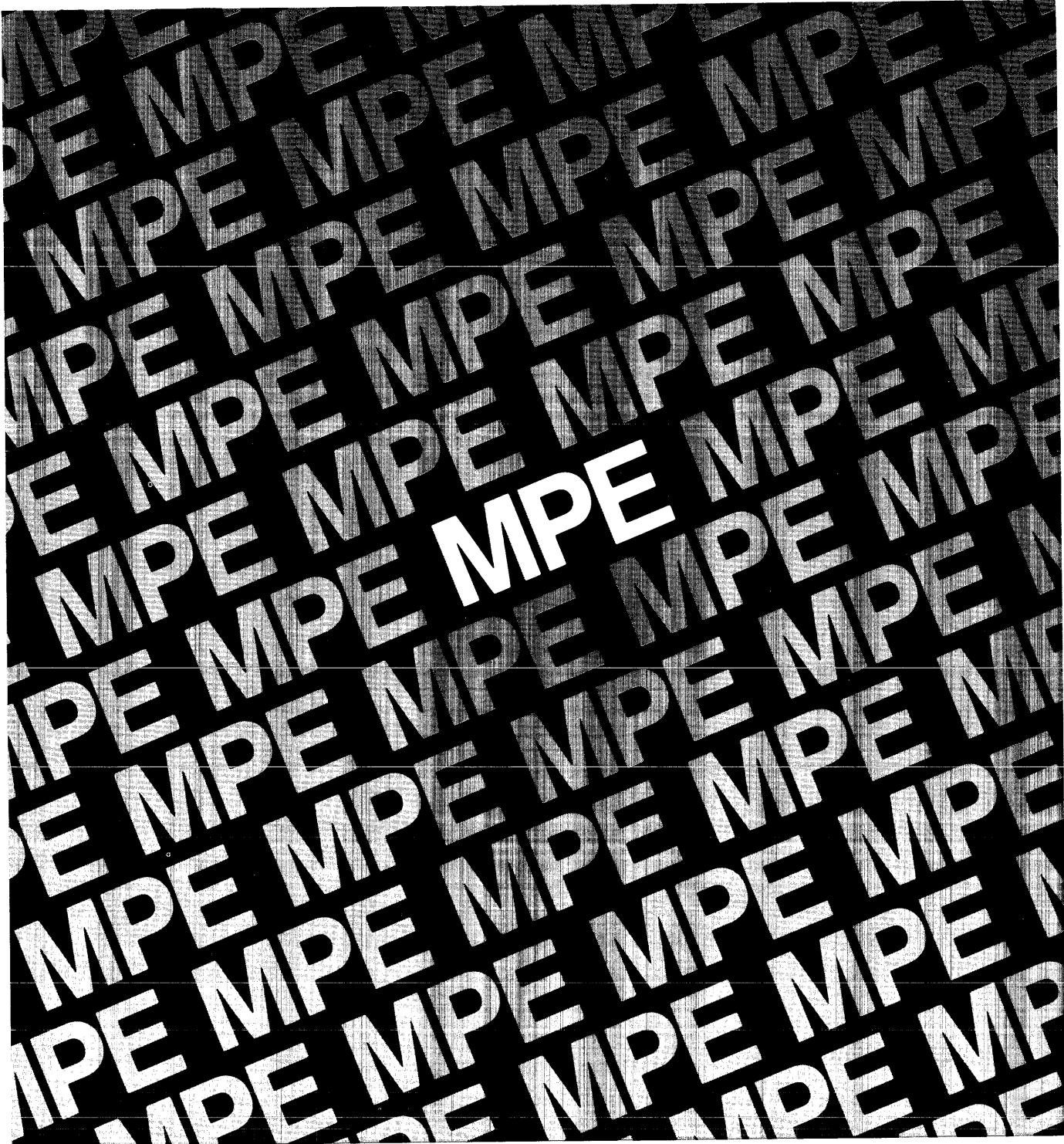


## MPE Intrinsic reference manual



# HP 3000 Computer Systems

## MPE Intrinsic Reference Manual



19447 PRUNERIDGE AVENUE, CUPERTINO, CA 95014

## **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 date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars are removed but the dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

Page	Date	Page	Date
i	DEC 1981	2-88 through 2-89	JUL 1981
iii	JUL 1981	2-93	JUL 1981
iv	DEC 1981	2-94	DEC 1981
v	JUL 1981	2-97	JUL 1981
vi	JUL 1981	2-146	JUL 1981
vii	JUL 1981	2-150a	JUL 1981
viii	DEC 1981	2-165	JUL 1981
ix	DEC 1981	2-167a through 2-167c	DEC 1981
x	JUL 1981	3-3	JUL 1981
xi	JUL 1981	4-1	JUL 1981
xii	JUL 1981	4-3 through 4-3a	JUL 1981
xiii	DEC 1981	4-35	DEC 1981
xiv	DEC 1981	6-2	JUL 1981
1-1	JUL 1981	6-3	JUL 1981
1-4	JUL 1981	10-32	DEC 1981
2-5	JUL 1981	10-40	JUL 1981
2-12	DEC 1981	10-71	JUL 1981
2-13a	JUL 1981	10-74 through 10-75a	JUL 1981
2-26 through 2-27	JUL 1981	10-82	JUL 1981
2-40a	JUL 1981	10-84	JUL 1981
2-49	DEC 1981	10-89 through 10-89b	JUL 1981
2-55	JUL 1981	10-91	JUL 1981
2-58	DEC 1981	10-96 through 10-97	DEC 1981
2-49 through 2-53	JUL 1981	D-3 through D-4	DEC 1981
2-55 through 2-56	JUL 1981	E-8	DEC 1981
2-59 through 2-60	JUL 1981	E-17	JUL 1981
2-61a through 2-61i	JUL 1981	E-19	JUL 1981
2-64	JUL 1981	E-22	JUL 1981
2-64a	DEC 1981	I-1 through I-7	JUL 1981
2-71	DEC 1981	I-8	DEC 1981
2-73	DEC 1981	I-9 through I-10	JUL 1981
2-76a	JUL 1981		

# PRINTING HISTORY

New editions are complete revisions of the manual. 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 of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

The software product part number printed alongside the date indicates the version and update level of the software product at the time the manual edition or update was issued. Many product updates and fixes do not require manual changes, and conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition . . . . .	Jun 1976 . . . . .	32002A
Update Package #1 . . . . .	Oct 1976 . . . . .	32002A
Update Package #2 . . . . .	Jan 1977 . . . . .	32002A
Update Package #2 . . . . .	Incorporated Feb 1977 . . . . .	32002A
Update Package #3 . . . . .	Apr 1977 . . . . .	32002A
Second Edition . . . . .	Apr 1978 . . . . .	32002B
Update Package #1 . . . . .	Jul 1979 . . . . .	32002B
Update Package #2 . . . . .	Jan 1980 . . . . .	32002B
Update Package #3 . . . . .	Mar 1980 . . . . .	32002B,32033B
Third Edition . . . . .	Jan 1981 . . . . .	32002C,32033C
Update Package #1 . . . . .	Jul 1981 . . . . .	32002C,32033C
Update Package #2 . . . . .	Dec 1981 . . . . .	32002C,32033C

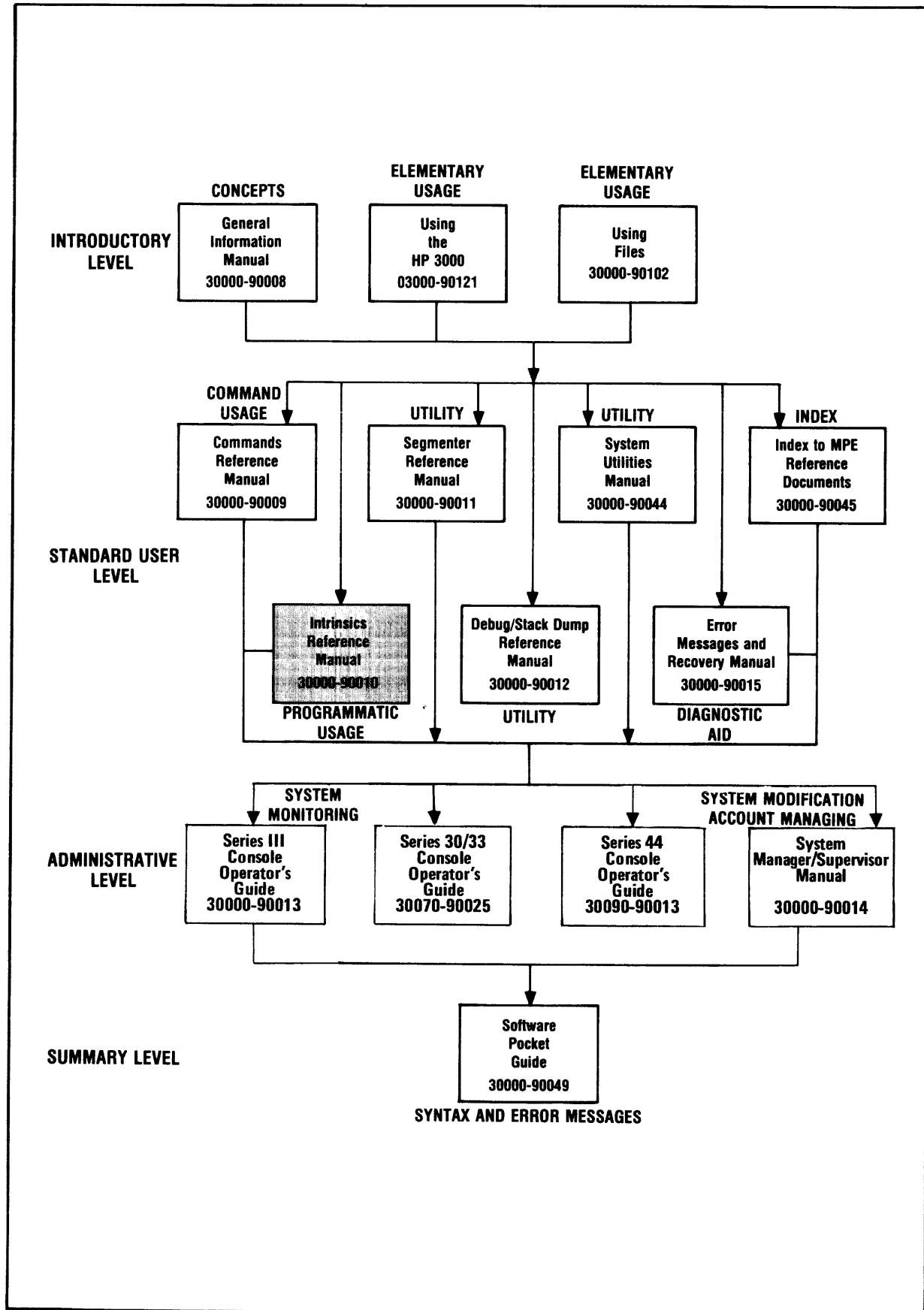
This manual is one of the set of manuals that document the Multiprogramming Executive Operating System (MPE-IV). The manual plan on the next page indicates the position of this manual (shaded block) in the overall set.

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	Interprocess Communication and Circular 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	Accessing and Altering Files

# MANUAL PLAN



# CONVENTIONS USED IN THIS MANUAL

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

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







# CONTENTS (continued)

REJECT	2-178
RESETCONTROL	2-179
RESETDUMP	2-180
SEARCH	2-181
SENDMAIL	2-182
SETDUMP	2-184
SETJCV	2-185
STACKDUMP	2-186
SUSPEND	2-188
SWITCHDB	2-189
TERMINATE	2-190
TIMER	2-191
UNLOADPROC	2-192
UNLOCKGLORIN	2-193
UNLOCKLOCRIN	2-194
WHO	2-195
WRITELOG	2-198
XARITRAP	2-199
XCONTRAP	2-201
XLIBTRAP	2-202
XSYSTRAP	2-203
ZSIZE	2-204

**Section III** Page

**INTERPROCESS COMMUNICATION AND CIRCULAR FILES**

Introduction	3-1
Operation	3-1
FOPEN	3-1
FREAD	3-2
FWRITE	3-2
FCONTROL	3-2
FCLOSE	3-2
Additional Features	3-2
Using IPC	3-3
Features of Intrinsic for Message Files	3-5
FOPEN	3-5
FCONTROL	3-8
FCHECK	3-8
FGETINFO	3-9
FFILEINFO	3-9
Circular Files	3-9
Features of Intrinsic for Circular Files	3-10
FOPEN	3-10
FWRITE	3-12
FCLOSE	3-12
Examples	3-12

**Section IV** Page

**UTILITY FUNCTIONS OF MPE INTRINSICS**

Dynamic Loading and Unloading of Library Procedures	4-2
Dynamic Loading	4-2
Dynamic Unloading	4-3
Searching Arrays	4-3
Formatting Command Parameters	4-4
Executing MPE Commands Programmatically	4-9
Determining the User's Access Mode and Attributes	4-10
Converting Numbers from Binary Code to	

ASCII Strings	4-12
Converting Numbers from an ASCII Numeric String to a Binary Coded Value	4-12
Translating Characters with the CTRANSLATE Intrinsic	4-14
Transmitting Program Input/Output from Job/Session Input/Output Devices	4-16
Reading Input from the Job/Session List Device	4-17
Writing Output to the Job/Session List Device	4-17
Writing Output to the Operator's Console	4-17
Writing Output to the Operator's Console and Requesting a Reply	4-19
Suspending the Calling Process	4-19
Requesting a Process Break	4-19
Terminating a Process	4-20
Aborting a Process	4-20
Aborting a Program	4-22
Changing Stack Sizes	4-22
Changing the DL to DB Area Size	4-23
Changing the Z to DB Area Size	4-29
Enabling and Disabling Traps	4-29
Arithmetic Traps	4-30
Standard Traps	4-31
Extended Precision Floating-Point Traps	4-31
Commercial Instruction Traps	4-32
Library Trap	4-34
System Trap	4-35
Control-Y Traps	4-38
Time and Date Intrinsic	4-42
Obtaining System Timer Information	4-42
Obtaining the Current Time	4-44
Obtaining the Calendar Date	4-44
Obtaining Process Run Time (Use of the Central Processor)	4-44
Formatting Calendar Date and Time Information	4-45
Interprocess Communication	4-46
User-Defined Job Control Words	4-47
MPE Message System	4-48
Message Catalog	4-48
MAKECAT Program	4-49
Using the GENMESSAGE Intrinsic	4-50

**Section V** Page

**DEVICE CHARACTERISTICS**

Device Characteristics	5-1
Paper Tape Reader	5-1
Binary Mode	5-1
ASCII Mode	5-1
Paper Tape Punch	5-3
Binary Mode	5-3
ASCII Mode	5-3
Card Reader	5-3
Line Printer	5-3
Magnetic Tape	5-4
Printing Reader/Punch	5-5
Line Printer and Terminal Carriage-Control Codes	5-7
End-of-File Indication	5-7
Terminals	5-9

# CONTENTS (continued)

<p>Terminal Types . . . . . 5-9</p> <p>Special Keys . . . . . 5-10</p> <p>Changing Terminal Characteristics . . . . . 5-12</p> <p style="padding-left: 20px;">Changing Terminal Speed . . . . . 5-12</p> <p style="padding-left: 20px;">Changing Input Echo Facility . . . . . 5-13</p> <p style="padding-left: 20px;">Enabling and Disabling System Break</p> <p style="padding-left: 40px;">Function . . . . . 5-15</p> <p style="padding-left: 20px;">Enabling and Disabling Subsystem Break</p> <p style="padding-left: 40px;">Function . . . . . 5-16</p> <p style="padding-left: 20px;">Enabling and Disabling Parity Checking . . . . . 5-16</p> <p style="padding-left: 20px;">Enabling and Disabling Tape-Mode Option . . . . . 5-17</p> <p style="padding-left: 20px;">Enabling and Disabling the Terminal Input</p> <p style="padding-left: 40px;">Timer . . . . . 5-18</p> <p style="padding-left: 20px;">Reading the Terminal Input Timer . . . . . 5-21</p> <p style="padding-left: 20px;">Defining Line-Termination Characters for</p> <p style="padding-left: 40px;">Terminal Input . . . . . 5-22</p> <p style="padding-left: 20px;">Enabling and Disabling Binary Transfers . . . . . 5-23</p> <p style="padding-left: 20px;">Enabling and Disabling User Block Transfers . . . . . 5-24</p> <p style="padding-left: 20px;">Enabling and Disabling Line Deletion Echo</p> <p style="padding-left: 40px;">Suppression . . . . . 5-25</p> <p style="padding-left: 20px;">Setting Parity . . . . . 5-25</p> <p style="padding-left: 20px;">Allocating a Terminal . . . . . 5-26</p> <p style="padding-left: 20px;">Setting Terminal Type . . . . . 5-27</p> <p style="padding-left: 20px;">Obtaining Terminal Type Information . . . . . 5-27</p> <p style="padding-left: 20px;">Obtaining Terminal Output Speed . . . . . 5-28</p> <p style="padding-left: 20px;">Setting Unedited Terminal Mode . . . . . 5-28</p> <p style="padding-left: 20px;">Reading Paper Tapes without X-OFF Control . . . . . 5-29</p> <p style="padding-left: 20px;">Using the FCARD Intrinsic to Operate the</p> <p style="padding-left: 40px;">HP 7260A Optical Mark Reader . . . . . 5-30</p> <p style="padding-left: 40px;">ASCII and Column Image Reading Formats . . . . . 5-31</p> <p style="text-align: right; margin-right: 20px;">Section VI . . . . . Page</p> <p><b>RESOURCE MANAGEMENT</b></p> <p>Inter-Job Level (Global) RIN's . . . . . 6-2</p> <p style="padding-left: 20px;">Acquiring Global RIN's . . . . . 6-2</p> <p style="padding-left: 20px;">Releasing Global RIN's . . . . . 6-3</p> <p style="padding-left: 20px;">Locking and Unlocking Global RIN's . . . . . 6-3</p> <p>Inter-Process (Local) Level RIN's . . . . . 6-6</p> <p style="padding-left: 20px;">Acquiring Local RIN's . . . . . 6-8</p> <p style="padding-left: 20px;">Locking and Unlocking Local RIN's . . . . . 6-8</p> <p style="padding-left: 20px;">Identifying Local RIN Owners . . . . . 6-9</p> <p style="padding-left: 20px;">Freeing Local RIN's . . . . . 6-10</p> <p style="text-align: right; margin-right: 20px;">Section VII . . . . . Page</p> <p><b>PROCESS-HANDING CAPABILITY</b></p> <p>Processes . . . . . 7-1</p> <p style="padding-left: 20px;">Organization of User Processes . . . . . 7-2</p> <p style="padding-left: 20px;">Process Substates . . . . . 7-2</p> <p style="padding-left: 20px;">Process to Process Communication . . . . . 7-2</p> <p>Creating and Activating Processes . . . . . 7-3</p> <p>Suspending Processes . . . . . 7-8</p> <p>Deleting Processes . . . . . 7-8</p> <p>Interprocess Communication . . . . . 7-10</p> <p style="padding-left: 20px;">Testing Mailbox Status . . . . . 7-10</p> <p style="padding-left: 20px;">Sending Mail . . . . . 7-11</p> <p style="padding-left: 20px;">Receiving (Collecting) Mail . . . . . 7-12</p> <p>Avoiding Deadlocks . . . . . 7-13</p> <p>Rescheduling Processes . . . . . 7-13</p> <p>Determining Source of Activation . . . . . 7-14</p>	<p>Determining Father Process . . . . . 7-14</p> <p>Determining Son Processes . . . . . 7-15</p> <p>Determining Process Priority and State . . . . . 7-15</p> <p style="text-align: right; margin-right: 20px;">Section VIII . . . . . Page</p> <p><b>DATA SEGMENT CAPABILITY</b></p> <p>Creating an Extra Data Segment . . . . . 8-2</p> <p>Deleting an Extra Data Segment . . . . . 8-15</p> <p>Transferring Data from an Extra Data Segment</p> <p style="padding-left: 20px;">to the Stack . . . . . 8-15</p> <p>Transferring Data from the Stack to an Extra Data</p> <p style="padding-left: 20px;">Segment . . . . . 8-15</p> <p>Changing the Size of an Extra Data Segment . . . . . 8-15</p> <p style="text-align: right; margin-right: 20px;">Section IX . . . . . Page</p> <p><b>PRIVILEGED MODE CAPABILITY</b></p> <p>Permanently Privileged Programs . . . . . 9-1</p> <p>Temporarily Privileged Programs . . . . . 9-2</p> <p>Entering Privileged Mode . . . . . 9-3</p> <p>Entering Non-Privileged Mode . . . . . 9-5</p> <p>Moving the DB Pointer . . . . . 9-5</p> <p>Scheduling Processes . . . . . 9-5</p> <p style="text-align: right; margin-right: 20px;">Section X . . . . . Page</p> <p><b>ACCESSING AND ALTERING FILES</b></p> <p>File Management System . . . . . 10-1</p> <p>File Characteristics . . . . . 10-2</p> <p style="padding-left: 20px;">Record Formats . . . . . 10-3</p> <p style="padding-left: 20px;">Relative I/O Block Format . . . . . 10-6</p> <p>File Device Relationships . . . . . 10-7</p> <p style="padding-left: 20px;">Non-Sharable Device Access . . . . . 10-7</p> <p style="padding-left: 20px;">File Domains . . . . . 10-7</p> <p style="padding-left: 20px;">File Label . . . . . 10-8</p> <p>File Accessing . . . . . 10-8</p> <p style="padding-left: 20px;">Relative I/O . . . . . 10-8</p> <p style="padding-left: 20px;">System-Defined Files . . . . . 10-9</p> <p style="padding-left: 20px;">User Pre-Defined (Back Referenced) Files . . . . . 10-10</p> <p style="padding-left: 20px;">New Files . . . . . 10-10</p> <p style="padding-left: 20px;">Old Files . . . . . 10-11</p> <p style="padding-left: 20px;">Input/Output Sets . . . . . 10-11</p> <p style="padding-left: 20px;">Accessing Files Already in Use . . . . . 10-12</p> <p style="padding-left: 20px;">Files on Non-Sharable Devices . . . . . 10-15</p> <p style="padding-left: 20px;">Special Considerations for Shared Files . . . . . 10-16</p> <p>Private Volumes Subsystem . . . . . 10-17</p> <p>How to Use Files . . . . . 10-17</p> <p style="padding-left: 20px;">Internal Operations for File Accessing . . . . . 10-17</p> <p>Opening Files . . . . . 10-27</p> <p style="padding-left: 20px;">Opening a New Disc File . . . . . 10-27</p> <p style="padding-left: 20px;">Opening an Old Disc File . . . . . 10-30</p> <p style="padding-left: 20px;">Foreign Disc Facility . . . . . 10-32</p> <p style="padding-left: 20px;">Opening a File on a Device other than Disc . . . . . 10-33</p> <p style="padding-left: 20px;">Issuing FREAD and FWRITE Intrinsic Calls for</p> <p style="padding-left: 40px;">\$STDIN and \$STDLIST . . . . . 10-35</p> <p>Closing Files . . . . . 10-39</p> <p style="padding-left: 20px;">Closing a New File as a Temporary File . . . . . 10-39</p> <p style="padding-left: 20px;">Closing a New File as a Permanent File . . . . . 10-40</p> <p style="padding-left: 20px;">Renaming a File . . . . . 10-43</p> <p style="padding-left: 20px;">Writing a File System Error-Check Procedure . . . . . 10-45</p> <p style="padding-left: 20px;">Reading a File in Sequential Order . . . . . 10-47</p>
---	---

# CONTENTS (continued)

Obtaining File Access Information . . . . .	10-66	Unlabeled Tapes . . . . .	10-89a
Using FFILEINFO. . . . .	10-68	Determining Tape Density. . . . .	10-89b
Obtaining File-Error Information . . . . .	10-68	Spacing on Disc or Tape Files . . . . .	10-89
Using FERRMSG . . . . .	10-69	Directing File Control Operations. . . . .	10-90
Magnetic Tape Considerations . . . . .	10-69	Resetting the Logical Record Pointer . . . . .	10-91
FWRITE . . . . .	10-71	Declaring Access-Mode Options . . . . .	10-91
FREAD. . . . .	10-71	Determining Interactive and Duplicative File Pairs . . . . .	10-92
FSPACE . . . . .	10-71	User Logging . . . . .	10-93
FCONTROL (Write EOF) . . . . .	10-71	How User Logging Works . . . . .	10-93
FCONTROL (Forward Space to File Mark) . . . . .	10-71	Effective Use of User Logging . . . . .	10-96
FCONTROL (Backward Spare to File Mark). . . . .	10-71	Suggested Log File Uses . . . . .	10-98
End-of-File Mark on Magnetic Tape . . . . .	10-72	Appendix A	
Spacing File Marks. . . . .	10-72	ASCII Character Set	
Using the FCLOSE Intrinsic with Magnetic Tape. . . . .	10-73	Appendix B	
MPE Tape Labels. . . . .	10-75	Disc File Labels	
Updating Magnetic Tape Files . . . . .	10-75	Appendix C	
Reading and Writing an Unlabeled Magnetic		End-of-File	
Tape File . . . . .	10-77	Appendix D	
Opening a Labeled Magnetic Tape File. . . . .	10-81	Magnetic Tape Labels	
Writing a Tape Label . . . . .	10-84	Appendix E	
Reading a Labeled Magnetic Tape File. . . . .	10-87	MPE Diagnostic Messages	
Writing to a Labeled Magnetic Tape File . . . . .	10-88		
Writing a User-Defined File Label on a			
Labeled Tape File . . . . .	10-88		
Reading a User-Defined File Label on a			
Labeled Tape File . . . . .	10-89		
Density Selection on Labeled and			
Unlabeled Tapes . . . . .	10-89		
Labeled Tapes. . . . .	10-89a		

# ILLUSTRATIONS

Title	Page	Title	Page
Calling the PRINTOP Intrinsic from SPL . . . . .	1-10	Using the GETDSEG and DMOVOUT Intrinsics (Program DSINIT) . . . . .	8-3
Using Numeric Values as Parameters in an Intrinsic Call . . . . .	1-10	Creating and Activating Two Son Processes (Program DSBOSS) . . . . .	8-4
Condition Code Checks . . . . .	1-12	Using the GETDSEG and DMOVIN Intrinsics (Program DSACCS) . . . . .	8-5
Item Numbers and Corresponding Items . . . . .	2-27	Array CALENDAR . . . . .	8-8
Foptions Bit Summary . . . . .	2-66	Using the GETPRIVMODE and GETUSERMODE Intrinsics . . . . .	9-4
Aoptions Bit Summary . . . . .	2-69	MPE Queue Structure . . . . .	9-6
Carriage-Control Directives . . . . .	2-117	Actions Resulting from Multiple Access of Files . . .	10-13
Carriage-Control Summary . . . . .	2-119	File Access Interface for New Disc Files . . . . .	10-18
Error Codes Returned From PROCINFO . . . . .	2-167b	File Name and Sector Address Storage . . . . .	10-21
Information Options For PROCINFO . . . . .	2-167c	File Access Interface for Old Disc File . . . . .	10-22
Data Paths Among Processes and Message Files (1) . .	3-12	Device Allocation Flowchart . . . . .	10-26
Data Paths Among Processes and Message Files (2) . .	3-16	Opening a New Disc File . . . . .	10-28
Using the MYCOMMAND Intrinsic . . . . .	4-5	Opening a Old Disc File . . . . .	10-31
Using the WHO Intrinsic . . . . .	4-11	Opening a File on a Device Other Than Disc . . . . .	10-34
Using the ASCII Intrinsic . . . . .	4-12	Opening \$STDIN and \$STDLIST . . . . .	10-36
Using the DASCII Intrinsic . . . . .	4-14	Closing a New File as a Temporary File . . . . .	10-39a
Using the BINARY Intrinsic . . . . .	4-15	Closing a New File as a Permanent File . . . . .	10-41
Using the PRINT and READ Intrinsics . . . . .	4-17	FRENAME Intrinsic Example . . . . .	10-44
Using the QUIT Intrinsic . . . . .	4-21	Error-Check Procedure Example . . . . .	10-46
Expanding and Contracting the DL to DB Area . . . . .	4-23	FREAD and FWRITE Intrinsics Example . . . . .	10-48
Using the DLSIZE Intrinsic . . . . .	4-25	FREADDIR and FREADSEEK Intrinsics Example . . . . .	10-52
Changing the DL to DB Area Size . . . . .	4-28	FWRITEDIR Intrinsic Example . . . . .	10-53
Using the XARITRAP Intrinsic . . . . .	4-33	FLOCK and FUNLOCK Intrinsics Example . . . . .	10-56
Using the XCONTRAP Intrinsic . . . . .	4-41	FUPDATE Intrinsic Example . . . . .	10-58
Using the TIMER Intrinsic . . . . .	4-43	Using the IOWAIT Intrinsic . . . . .	10-60
FMTCALENDAR, FMTCLOCK, and FMTDATE Intrinsics Example . . . . .	4-45	FWRITELABEL Intrinsic Example (Disc File) . . . . .	10-64
GENMESSAGE Intrinsic Example . . . . .	4-51	FREADLABEL Intrinsic Example . . . . .	10-65
Carriage Control Directives . . . . .	5-8	FGETINFO Intrinsic Example . . . . .	10-67
Echo Facility vs Duplex Mode . . . . .	5-12	FCHECK Intrinsic Example . . . . .	10-70
Using the FCONTROL Intrinsic to Enable and Read the Terminal Input Timer . . . . .	5-18	Using the FCLOSE Intrinsic with Unlabeled Magnetic Tape . . . . .	10-74
FCARD Intrinsic Example . . . . .	5-30	Unlabeled Magnetic Tape Example . . . . .	10-78
Using the LOCKGLORIN and UNLOCKGLORIN Intrinsics . . . . .	6-4	Opening a Labeled Magnetic Tape File . . . . .	10-82
Using the CREATE and ACTIVATE Intrinsics . . . . .	7-4	Writing a Tape Label . . . . .	10-85
Process Deletion . . . . .	7-9	User Logging Facility . . . . .	10-94
		MPE Tape Labels (Conforming to ANSI-Standard . . . .	D-2

# TABLES

Title	Page	Title	Page
Summary of MPE Intrinsic . . . . .	1-3	Intrinsic Errors . . . . .	E-6
Intrinsic That are Not Permitted with Message Files . . . . .	3-5	Run-Time Errors . . . . .	E-7
Intrinsic That are Not Permitted with Circular Files . . . . .	3-10	File System Errors . . . . .	E-8
Line Printer Differences . . . . .	5-4	Loader Errors . . . . .	E-12
Carriage-Control Directives . . . . .	5-7	CREATE Intrinsic Errors . . . . .	E-13
Terminals Supported by MPE . . . . .	5-9	ACTIVATE Intrinsic Errors . . . . .	E-13
Device Dependent Restrictions . . . . .	10-24	SUSPEND Intrinsic Error . . . . .	E-13
Classification of Devices . . . . .	10-25	MYCOMMAND Intrinsic Error . . . . .	E-13
Format of Tape Labels Written by MPE (ANSI Standard) . . . . .	D-3	LOCKGLORIN Intrinsic Errors . . . . .	E-13
Program Errors . . . . .	E-5	Private Volumes Messages . . . . .	E-14
		User-Logging Error Messages . . . . .	E-15
		CLEANUSL Error Messages . . . . .	E-16
		CREATEPROCESS Error Messages . . . . .	E-17

# 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 MPE Operating 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.

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
ACCEPT	Accepts (and completes) a request received by the preceding GET intrinsic call. (Used only with DS/3000.)	Standard
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
CLEANUSL	Deletes inactive entries from USL file.	Standard
CLOCK	Returns the actual time.	Standard
CLOSELOG	Closes access to the logging facility.	LG Capability
COMMAND	Executes an MPE command programmatically.	Standard
CREATE	Creates a process.	Process Handling
CREATEPROCESS	Provides ability to assign \$STDIN and \$STDLIST to any file	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
DATELINE	Returns date and time information.	Standard
DBINARY	Converts a number from ASCII code to a double-word binary value.	Standard
DEBUG	Calls the DEBUG facility.	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

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FATHER	Requests Process Identification Number (PIN) of father process.	Process Handling
FCARD	Drives the HP 7260A Optical Mark Reader.	Standard
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
FDELETE	Deactivates a RIO record.	Standard
FDEVICECONTROL	Adds control directives to a spooled device file.	Standard
FERRMSG	Returns message corresponding to FCHECK error number.	Standard
FFILEINFO	Provides access to file information.	Standard
FGETINFO	Requests access and status information about a file.	Standard
FINDJCW	Searches Job Control Word (JCW) table for specified JCW.	Standard
FLOCK	Dynamically locks a file.	Standard
FMTCALENDAR	Formats calendar date.	Standard
FMTCLOCK	Formats time of day.	Standard
FMTDATE	Formats calendar date and time of day.	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
FREADBACKWARD	Reads a logical record beginning at a point prior to the current record pointer	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

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FRELOCRIN	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 file label.	Standard
GENMESSAGE	Accesses MPE message system.	Standard
GET	Receives the next request from a remote master program. (Used only with DS/3000.)	Standard
GETDSEG	Creates an extra data segment.	Data Segment Management
GETJCW	Fetches contents of system job control word (JCW).	Standard
GETLOCRIN	Acquires local RIN's.	Standard
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
IODONTWAIT	Initiates completion operations for an I/O request.	Privileged Mode

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
IOWAIT	Initiates completion operations for an I/O request.	Privileged Mode
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
LOCRINOWNER	Identifies process locking a local RIN.	Standard
MAIL	Tests mailbox status.	Process Handling
MYCOMMAND	Parses (delineates and defines parameters) for user-supplied command image.	Standard
OPENLOG	Provides access to a logging facility.	LG Capability
PAUSE	Suspends calling process for a specified number of seconds.	Standard
PCHECK	Returns an integer code specifying the completion status of the most recently executed DS/3000. (Used only with DS/3000.)	Standard
PCLOSE	Terminates program-to-program communication with a remote slave program. (Used only with DS/3000.)	Standard
PCONTROL	Exchanges tag fields with a remote slave program. (Used only with DS/3000.)	Standard
POPEN	Initiates program-to-program communication with a remote slave program. (Used only with DS/3000.)	Standard
PREAD	Requests a block of data from a remote slave program. (Used only with DS/3000.)	Standard
PRINT	Prints character string on job/session list device.	Standard
PRINTFILEINFO	Prints file information display.	Standard
PRINTOP	Prints a character string on the Operator's Console.	Standard
PRINTOPREPLY	Prints a character string on the 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

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
PUTJCW	Puts value of a given JCW in JCW table.	Standard
PWRITE	Sends a block of data to a remote slave program.	Standard
QUIT	Aborts a process.	Standard
QUITPROG	Aborts the user process structure.	Standard
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
REJECT	Rejects the request received by the preceding GET intrinsic call. (Used only with DS/3000.)	Standard
RESETCONTROL	Resets terminal to accept CONTROL Y signal.	Standard
RESETDUMP	Disables the abort stack analysis facility.	Standard
SEARCH	Searches an array for a specified entry or name.	Standard
SENDMAIL	Sends mail to another process.	Process Handling
SETDUMP	Enables the abort stack analysis facility.	Standard
SETJCW	Sets the value of the system job control word (JCW).	Standard
STACKDUMP	Dumps selected parts of stack to file.	Standard
SUSPEND	Suspends a process.	Process Handling
SWITCHDB	Switches DB register pointer.	Privileged Mode
TERMINATE	Terminates a process.	Standard
TIMER	Returns job or session timer bit count.	Standard
UNLOADPROC	Dynamically unloads a library procedure.	
UNLOADGLORIN	Unlocks a global RIN.	Standard
UNLOCKLOCRIN	Unlocks a local RIN.	Standard
WHO	Returns user attributes.	Standard
WRITELOG	Writes a record to a logging file.	LG Capability
XARITRAP	Arms or disarms the software arithmetic trap.	Standard
XCONTRAP	Arms or disarms the CONTROL-Y trap.	Standard
XLIBTRAP	Arms or disarms the library trap.	Standard

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
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 2-82) 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 2-147 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 logical 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*.



PAGE 0001 HP32100A.06.0 (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1976

```
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:=160;
00012000 00004 1 PRINTOP(MESSAGE, LENGTH, CONTROL);
00013000 00010 1
00014000 00010 1 END.
PRIMARY DB STORAGE=1003; SECONDARY DB STORAGE=100012
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

PAGE 0001 HP32100A.06.0 (C) COPYRIGHT HEWLETT-PACKARD COMPANY 1976

```
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, 160);
00010000 00004 1
00011000 00004 1 END.
PRIMARY DB STORAGE=1001; SECONDARY DB STORAGE=100012
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.

The *User Logging Optional Capability* provides a flexible transaction logging capability which enables you to journalize additions and modifications to your data bases and subsystem files. User logging permits you to journalize on two mediums: tape and disc. If the data base is lost, the logging tape or disc file can be used to recover the lost transactions.

# 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 number is only significant for error diagnosis. See the Error Messages and Recovery Manual.)
- 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 1-8 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, for example, 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:

I                    LA                    IV  
*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,

I                    LA                    IV  
*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 1-13.

*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.



# ACCEPT

Accepts (and completes) a request received by the preceding GET intrinsic call and returns an optional tag field back to a remote master program.

```
IA IA IV  
ACCEPT(itag,target,tcount);
```

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

Activates a process.

INTRINSIC NUMBER 104

```
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 2-172) 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 group in which the program file resides 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 request to activate the father would result in activation of a job or session main process or a system process.

## 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 2-172) and CREATE (see page 2-19) 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.

# ACTIVATE

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

Adjusts directory space in a USL file.

INTRINSIC NUMBER 83

```
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 *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*.

# 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.

## TEXT DISCUSSION

*MPE Segmenter Reference Manual.*

Alters the size of an extra data segment.

INTRINSIC NUMBER 134

```

LV IV I
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.

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

## NOTE

Sufficient virtual space is allocated by the system when a data segment is created through GETDSEG to accommodate the original length of the data segment. This virtual space is allocated in increments of pages where the number of words per page is set when the system is configured (typically 512 words/page). For example, creation of a data segment with a length of 600 words would result in two virtual pages being allocated for the data segment (space for 1024 words).

In no case may ALTDSEG increase the size of a data segment to exceed the virtual space originally allocated through GETDSEG.

## 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 by value (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 (required)</i> A word to which is returned the new size of the data segment after incrementing or decrementing occurs.

# ALTDSEG

## 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 virtual space originally assigned by GETDSEG, was attempted. In the first case, the current size remains in effect. In the second case, the size of the virtual space is granted and this size is returned through the size parameter.
CCL	Request denied because an illegal <i>index</i> parameter was specified.

## SPECIAL CONSIDERATIONS

Data-Segment Management Capability required.

## TEXT DISCUSSION

Page 8-16





# ASCII

INTRINSIC NUMBER 63

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

```
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* (including 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*.

# BEGINLOG

INTRINSIC NUMBER 211

Marks the beginning of user logging transaction.

```
      D  L A I   I   I  
BEGINLOG (index, data, len, mode, status);
```

The BEGINLOG intrinsic posts a special record to the user logging file to mark the beginning of a logical transaction in the log file. When BEGINLOG is used, the logging memory buffer is flushed to ensure that the record gets to the logging file. BEGINLOG can be used also to post data to the logging file by using the *data* parameter. This function of BEGINLOG performs the same procedure as the WRITELOG intrinsic.

## PARAMETERS

- data*                      *array (required)*  
An array in which the actual information to be logged is passed. A log record contains 128 words of which 119 words are available to the user. Because of this, the most efficient use of log file space is a multiple of 119 words.
- len*                        *integer (required)*  
The length of the data in *data*. A positive count indicates words, and a negative count indicates bytes. If the length is greater than 119 words, the information in *data* will be divided into two or more physical log records.
- index*                     *double (required)*  
The parameter returned from OPENLOG that identifies the users access to the logging system.
- status*                    *integer (required)*  
An integer that the logging system uses to return error information to the user. Zero indicates OK status.
- mode*                      *integer (required)*  
An integer which specifies whether you want your process impeded by the logging process if the logging buffer is full. If it is not possible to log the transaction and the mode is set to *nowait*, the BEGINLOG intrinsic will return an indication in the *status* word that the request was not completed. Mode zero indicates *wait*; mode one indicates *nowait*.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

None.

# BINARY

INTRINSIC NUMBER 62

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

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

## FUNCTIONAL RETURN

This intrinsic returns the binary equivalent of the numeric string.

## PARAMETERS

*string*                      *byte array (required)*  
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.

### NOTE

*String* cannot contain blanks.

*length*                      *integer by value (required)*  
An integer representing the length (number of bytes) in the byte array 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 one-word binary value is returned to the user's process.

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

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

## TEXT DISCUSSION

Page 4-13.

# CALENDAR

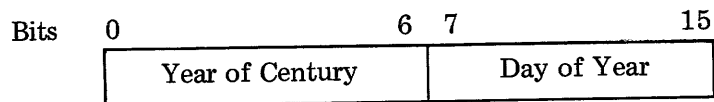
Returns the calendar date.

INTRINSIC NUMBER 43

```
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

INTRINSIC NUMBER 56

Places a session in break mode.

**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

# CLEANUSL

Deletes inactive entries from USL file

```
I          IV  BA
filename=CLEANUSL (uslfnm,filename);
```

## FUNCTIONAL RETURN

CLEANUSL deletes all inactive entries from currently managed USL files and returns the new file number. If an error occurs, the error number is returned instead of the new file number. (See Table 10-13, CLEANUSL Error Messages) The condition code, therefore, must be tested immediately on return from the intrinsic. Unpredictable results occur if an error number is used as a file number.

### NOTE

CLEANUSL requires at least 3000 words of available stack space to execute.

## PARAMETERS

*uslfnm*

*integer by value (required)*

A word identifier which supplies the file number of the file.

*filename*

*byte array (required)*

The name to be given to the cleaned file. The array must end with a blank, but it can be all blanks. If it's all blanks it purges the inactive entries.

## CONDITION CODES

CCE	Request granted. The new file number is returned.
CCG	Not returned by this intrinsic.
CCL	Request denied. (See Table 10-13, CLEANUSL Error Messages)

## TEXT DISCUSSION

None

# 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.

Closes access to the logging facility.

INTRINSIC NUMBER 212

D I I  
CLOSELOG (*index, mode, status*);

The CLOSELOG intrinsic closes access to the logging facility.

## PARAMETERS

<i>index</i>	<i>double (required)</i> The parameter returned from OPENLOG that identifies your access to the logging facility.
<i>mode</i>	<i>integer (required)</i> An integer which you use to indicate whether or not your process should be suspended if your request for service cannot be completed immediately. Enter a zero if you want to wait for service; enter a one if you do not want to wait.
<i>status</i>	<i>integer (required)</i> An integer which indicates logging system errors to you. (See table E-12, User Logging Error Messages.)

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 10-93



# COMMAND

INTRINSIC NUMBER 68

Executes an MPE command programmatically.

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

## NOTE

User-defined commands may not be used.

## 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, i.e., command interpreter error code not file system error code. If no error occurs, *error* returns zero.
- parm* *integer (required)*  
A word to which the number (index) of the erroneous parameter is returned. If no parameters are in error, *parm* returns zero. If there are errors, *parm* may be zero or some positive integer. In the case where an error refers to a file system problem, *parm* is the file system error code.

## CONDITION CODES

- CCE Request granted.
- CCG An executor-dependent error, such as an erroneous parameter, prevented execution of the command. The *error* parameter contains the numeric error code.
- CCL Request denied. The command was an undefined command.

## TEXT DISCUSSION

Page 4-9

# CREATE

Creates a process.

INTRINSIC NUMBER 100

```
          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.

# CREATE

## PARAMETERS

<i>programe</i>	<i>byte array (required)</i> Contains a string, terminated by a blank, specifying the name, and optionally, the account and group ( <i>filereference</i> format, see Section III, page 3-8) of the file containing the program to be run.
<i>entryname</i>	<i>byte array (optional)</i> 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 <i>primary</i> entry point in the program can be specified by setting the array equal to a blank character alone. <i>Default: The primary entry point is used.</i>
<i>pin</i>	<i>integer (required)</i> 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.
<i>param</i>	<i>integer by value (optional)</i> 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. <i>Default: Word is filled with zeros.</i>
<i>flags</i>	<i>logical by value (optional)</i> 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:

S	00	— System Library.
P	01	— Account Public Library, followed by System Library.
G	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.*

# CREATE

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.*

*dlsize*

*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*, *dlsize*, 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.

## 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\ e + 100)</math> modulo 128 is <i>not</i> zero, then <i>dlsiz</i>e is rounded upward so that <math>(dlsiz\ e + 100)</math> modulo 128 = 0.</li></ol>
CCL	Request denied because the <i>progrname</i> or <i>entryname</i> specified does not exist.

## SPECIAL CONSIDERATIONS

Process-Handling Capability required.

## TEXT DISCUSSION

Page 7-3.

# CREATEPROCESS

INTRINSIC NUMBER 101

Provides the ability to assign \$STDLIST and \$STDIN to any file.

I I BA IA LA O-V  
CREATEPROCESS(*error*,*pin*,*progname*,*itemnums*,*items*);

The CREATERPROCESS Intrinsic allows you to assign the system defined files, \$STDIN and \$STDLIST, to any file at process creation time. You are not limited to system defined defaults. Note that Process-Handling capability is required to call this intrinsic, and that it may not be called in split stack mode. If the intrinsic is called with the *error* parameter omitted, an invalid address for parameter *error* is returned. In split stack mode, the calling process will be aborted.

## PARAMETERS

<i>error</i>	<i>integer (required)</i> An integer indicating success or failure type. (See Table E-14.)
<i>pin</i>	<i>integer (required)</i> An integer in which the <i>PIN</i> of the newly created process is returned. If there is an error in creating the new process, i.e., parameter <i>error</i> > 0 a zero is returned.
<i>progname</i>	<i>byte array (required)</i> A byte array containing a string terminated by any non-alphanumeric character other than a period or a slash which specifies the name of the program file to be run by the new process.
<i>itemnums</i>	<i>integer array (optional)</i> An array containing the item numbers (in any order) of the options you want to use in creating a new process. This array must contain a zero as its last element to indicate the end of the option list. (See Figure 2-0).
<i>items</i>	<i>logical array (optional)</i> An array containing the items (in the same order as the item numbers in <i>itemnums</i> ), to be used in creating the new process. (See Figure 2-0).

## CONDITION CODES

CCE	No error.
CCL	Unsuccessful.
CCG	Successful. Error numbers preceded by a minus sign (-) indicate a warning only. (See Table E-14.)

## SPECIAL CONSIDERATIONS

Process-Handling capability required.

## TEXT DISCUSSION

None.

The item numbers in the array *itemnums* indicate the options to be applied in creating the new process. The corresponding items in the array *items* give the information necessary for each option to be used.

Itemnumber	Item
1	A pointer to a byte array containing the name of the entry point in the program where the new process is to begin execution. The name is specified as a string of characters terminated by a blank.
2	An integer containing a parameter to be passed to the new process (accessed through Q-4 of the outer block).
3	A logical value containing the load option flags to be used in loading the program file for the new process. This parameter has the same definition as the <i>flags</i> parameter of the CREATE intrinsic.
4	An integer specifying the initial stack size (Q - Z).
5	An integer specifying the initial DLsize (DL-DB) for the new process.
6	An integer specifying the maximum stack size (DL-Z) for the new process (i.e. MAXDATA).
7	A string of 2 ASCII characters specifying the priority class in which the new process is to be scheduled. ("CS", "DS", or "ES".)
8	A pointer to a byte array containing the definition of a file to be used as \$STDIN for the new process. (See description below).
9	A pointer to a byte array containing the definition of a file to be used as \$STDLIST for the new process (see description below).
10	A logical value indicating suspension and anticipated source of re-activation. Specification of this parameter causes the newly created process to be ACTIVATED automatically upon creation completion. The meanings of the individual bit fields of this parameter are the same as those of the <i>susp</i> parameter of the ACTIVATE intrinsic.
11	A pointer to a byte array containing a string of information to be passed to the new process. The length of the string is specified with item number 12.
12	An integer specifying the length in bytes of the string specified with item number 11.

#### NOTES

If item numbers 8 or 9 are not specified, the default \$STDIN and \$STDLIST will be used in creating the new process. These defaults are the current \$STDIN and \$STDLIST files for the creating (father) process.

Item number 8 indicates that the corresponding item in the item array is the address of a byte array which contains the definition of the file to be used as \$STDIN for the new process. This byte array must contain an ASCII string (terminated by a carriage return) which is the right hand side of a file equation specifying the file to be used as \$STDIN (i.e. everything after the ":FILE *formaldesignator*=" portion of the file equation).

Item number 9 indicates that the corresponding item in the item array is the address of a byte array which contains the definition of the file to be used as \$STDLIST for the new process. This array is defined as above for \$STDIN.

Item numbers 11 and 12 indicate that a string is to be passed to the new process. The string will be placed just after the global area of the new process's stack. A DB relative byte pointer to the string in the new process's stack will be placed at Q-5 of the stack (where Q is the initial value of the Q-register at activation time), and the length of the string in bytes will be placed at Q-6. If no string is specified to be passed to the new process, Q-5 and Q-6 will both contain 0.

Figure 2-0. Item Number and Corresponding Items



# CTRANSLATE

INTRINSIC NUMBER 61

Converts a string of characters from EBCDIC to ASCII, ASCII to EBCDIC, EBCDIK to JIS (katakana), or JIS to EBCDIK.

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

The CTRANSLATE intrinsic is used for character code translating, whether between the standard computer character codes or with a user defined code. It permits you to obtain character code conversions within programs of your own design. In the code parameter of CTRANSLATE, the following values specify the translation table to be used:

## PARAMETERS

- code* *integer by value (required)*  
An integer identifying a specific translation to be used as follows:
- 0 = The user supplied table specified in the parameter, *table*.
  - 1 = EBCDIC to ASCII.
  - 2 = ASCII to EBCDIC.
  - 3 = Reserved for future use.
  - 4 = Reserved for future use.
  - 5 = EBCDIK to JIS (katakana data).
  - 6 = JIS to EBCDIK.
- 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 value in the source string to be translated; for example, the fifth byte in the table gives the code to be substituted for source bytes whose value is 4.

# CTRANSLATE

## CONDITION CODES

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

## TEXT DISCUSSION

Page 4-13

# DASCII

INTRINSIC NUMBER 75

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

```
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.

## **CONDITION CODES**

The condition code remains unchanged.

## **TEXT DISCUSSION**

Page 4-13.

# DATELINE

Returns date and time information.

```
BA
DATELINE (datebuf);
```

## PARAMETERS

*datebuf* *byte array (required)*  
A byte array reference, (minimum 27 characters), to which the date and time information is returned.

byte string	F	r	i	,	M	a	y		2	5	,		1	9	7	9	,		1	2	:	0	6		P	M	
byte index	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

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

None

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.

# DEBUG

INTRINSIC NUMBER 99

Invokes the DEBUG facility.

DEBUG;

## PARAMETERS

None.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

*MPE DEBUG/Stack Dump Reference Manual.*





# DLSIZE

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 conditions.

## TEXT DISCUSSION

Page 4-22.



# DMOVIN

## 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

Copies data from stack to extra data segment.

INTRINSIC NUMBER 133

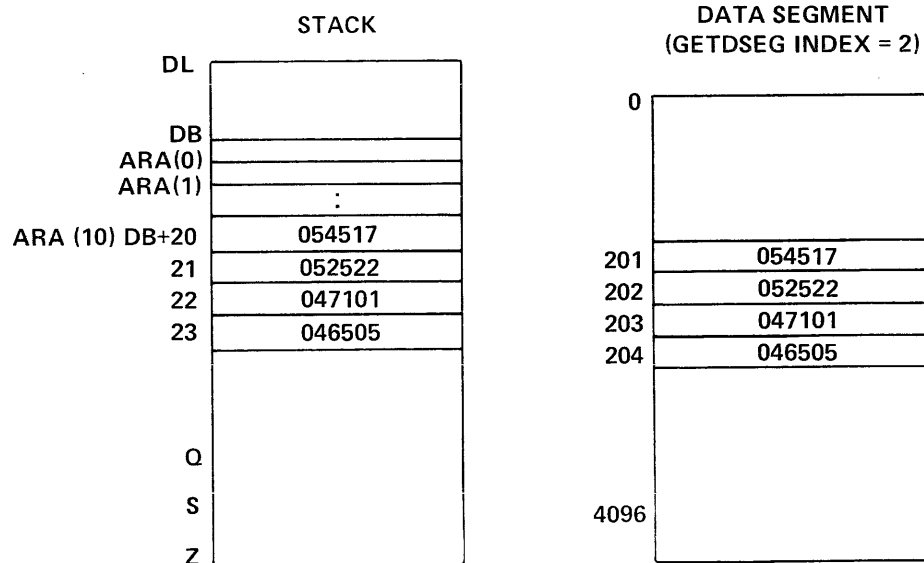
```
LV IV IV LA
DMOVOUT(index, disp, number, location);
```

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 2-111), 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.



# ENDLOG

INTRINSIC NUMBER 211

Marks the end of a user logging transaction.

```
      D L A I I I  
ENDLOG (index, data, len, mode, status);
```

The ENDLOG intrinsic posts a special record to the logging file to mark the end of a logical transaction in the logging file. When the record is posted, ENDLOG flushes the user logging memory buffer to ensure that the record gets to the logging file.

The *data* parameter of the intrinsic can be used to post user data to the log file. This function of the procedure is identical to the WRITELOG intrinsic.

## PARAMETERS

<i>data</i>	<i>array (required)</i> An array in which the actual information to be logged is passed. A log record contains 128 words of which 119 words are available to the user. Because of this, the most efficient use of log file space is a multiple of 119 words.
<i>len</i>	<i>integer (required)</i> The length of the data in <i>data</i> . A positive count indicates words and a negative count indicates bytes. If the length is greater than 119 words, the information in data will be divided into two or more physical log records.
<i>index</i>	<i>double (required)</i> The parameter returned from OPENLOG that identifies the user's access to the logging file.
<i>mode</i>	<i>integer (required)</i> An integer which specifies whether you want your process impeded by the logging process if the logging buffer is full. If it is not possible to log the transaction and the mode is set to nowait, the ENDLOG intrinsic will return an indication in the status word that the request was not completed. Mode zero indicates wait; mode one indicates nowait.
<i>status</i>	<i>integer (required)</i> An integer that the logging system uses to return error information to the user. Zero indicates no errors.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

None.

# DMOVOUT

## 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

Changes length of a USL file.

INTRINSIC NUMBER 84

```
      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

<i>uslfnm</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file.
<i>records</i>	<i>integer by value (required)</i> A signed integer specifying the number of records by which the length of the USL file is to be changed. If <i>records</i> is positive, the new USL file is longer than the old USL. If <i>records</i> 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 <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> .
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.

# EXPANDUSLF

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 (save) the new USL file.
10	The intrinsic was unable to close \$NEWPASS.
11	The intrinsic was unable to open \$OLDPASS.

## TEXT DISCUSSION

*MPE Segmenter Reference Manual.*



Requests PIN of father process.

INTRINSIC NUMBER 109

```
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.

# FCARD

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.

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.

# FCARD

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 or responded to the read, select, retransmit, or temporary suspend request. 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*).

- 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).

## CONDITION CODE

The condition code remains unchanged.

## TEXT DISCUSSION

Page 5-28.

# FCHECK

INTRINSIC NUMBER 10

Requests details about file input/output errors.

```
IV      I  I  D  I  0-V  
FCHECK(filenum,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 *filenum* 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 *filenum*. Therefore, when an FOPEN fails, a *filenum* 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 *filenum* of zero is used for a file which has been previously FOPENed, but not yet FCLOSEd, the returned *errorcode* will be meaningless.

## PARAMETERS

<i>filenum</i>	<i>integer by value (optional)</i> A word identifier supplying the file number of the file for which error information is to be returned. If omitted, FCHECK assumes you want the last FOPEN error.
<i>errorcode</i>	<i>integer (optional)</i> A word to which is returned an 8-bit code (16 bits for KSAM) specifying the type of error that occurred. If the previous operation was successful or an EOF is encountered, 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 on the last data transfer. This word specifies the number of words actually read or written if an input/output error occurred. <i>Default: The transmission log value is not returned.</i>
<i>blknum</i>	<i>double (optional)</i> The physical record count, if the file is not a spool file; the logical record count if it is a spool file. The physical count is the number of physical records transferred to or from the file since a) FOPEN, for fixed and undefined length record files; b) the last rewind, rewind/unload space forward or backward to tape mark, for variable length record files.
<i>numrecs</i> <i>numrecs</i>	<i>integer (optional)</i> A word to which is returned the number of logical records in the bad block (blocking factory). <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. (Non KSAM access.)

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.
3	Parameter omitted in IOWAIT.
5	DRT number > 255.
8	Illegal parameter value.
9	Undefined file type.
10	Invalid record size.
11	Invalid block size.
12	Record number out of range.
16	More than 255 FOPENS applied against one file.
17	Magnetic tape runaway (tape is blank).
18	Device did a power-up reset.
19	Line printer did a VFC reset.
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 (No defective track table entry is made on foreign discs).
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>recsize</i> , 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.

# FCHECK

Code (Decimal)	Meaning
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.
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.
70	Header or trailer I/O error.
71	Process file access information area exhausted. (Try preparing with NOCB.)
72	Invalid file number.
73	Bounds check violation.
76	Input buffer absent in IOWAIT.
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.
88	Spooling error.
89	Power failure.



Code (Decimal)	Meaning
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).
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 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.
111	User lacks save files (SF) capability.
112	User lacks private volumes (UV) capability.
113	Volume set not mounted — mount problem.
114	Volume set not dismounted — dismount problem.
115	Attempted rename across volume sets.
116	Invalid tape label FOPEN parameters.
117	Attempted to write on an unexpired tape file.
118	Invalid header or trailer tape label.
119	Input/output error positioning tape for tape labels.
121	Tape label lockword violation.
122	Tape label table overflow.
123	End of tape volume set.
124	Append request to labelled tape.
126	Character set number must be between 0 and 31.
127	Form number must be between 0 and 31.
128	Logical page number must be between 0 and 31.
129	Vertical format number must be between 0 and 31.
130	Number of copies must be between 1 and 32767.
131	Number of overlays must be between 1 and 8.
132	Page length parm must be between 12 (=3”) and 68 (=17”).
139	Deleted sectors on IBM diskette.
148	Inactive RIO record accessed.

# FCHECK

Code (Decimal)	Meaning
149	Missing item number or return variable.
150	Invalid item number.
151	Undefined file type (invalid file type in FOPTION of FOPEN).
152	Unrecognized key word in FOPEN <i>device</i> parameter.
153	Expecting “;” or “cr” in FOPEN <i>device</i> parameter.
154	Environment file open error.
155	File not environment file. Check file code or record size.
156	Header record incorrect.
157	Uncompiled environment file.
158	Error reading environment file.
159	Error closing environment file.
160	Error doing FDEVICECONTROL from environment file.
161	Too many parameters in device string—overflow.
162	Expecting “=” after keyword in <i>device</i> parameter.
163	“ENV” back reference in file equation incorrect.
164	<i>Device</i> parameter too large or missing carriage return.
165	Invalid density specification.
166	FFILEINFO failed in accessing remote spoolfile.
167	Spoolfile label error, can’t insert ENV file name.
170	The record is marked deleted. FPOINT positioned pointer to a record that was marked for deletion.
171	Duplicate key value (KSAM error).
172	No such key (KSAM error).
173	<i>Tcount</i> parameter larger than record size (KSAM error).
174	Cannot get extra data segment (KSAM error).
175	Internal KSAM error.
176	Illegal extra data segment (KSAM error).
177	Too many extra data segments for this process (KSAM error).
178	Not enough virtual memory for extra data segment (KSAM error).
179	File must be locked before issuing this intrinsic (KSAM error).
180	The KSAM file must be rebuilt because this version of KSAM does not handle the file built by previous version.
181	Invalid key starting position (KSAM error).
182	File is empty (KSAM error).
183	Record does not contain all keys (KSAM error).
184	Invalid record number (KSAM FFINDN intrinsic error).
185	Sequence error in primary key (KSAM error).
186	Invalid key length (KSAM error).
187	Invalid key specification (KSAM error).
188	Invalid device specification (KSAM error).
189	Invalid record format (KSAM error).
190	Invalid key blocking factor value (KSAM error).
191	Record does not contain search key for deletion. Specified key value points to record which does not contain that value.
192	System failure occurred while KSAM file was opened.
201	Invalid ID sequence (CS error).
202	Invalid telephone number (CS error).
203	No telephone list specified (CS error).

Code (Decimal)	Meaning
204	Unable to allocate an extra data segment for DS/3000.
205	Unable to expand the DS/3000 extra data segment.
212	File number returned from IOWAIT is not a DS line number.
214	The requested DS line has not been opened with a user :DSL command.
216	Message rejected by remote computer (DS error).
217	Insufficient amount of user stack available (DS error).
221	Invalid DS message format. (Internal DS error.)
240	Local communication line not opened by operator (DS error).
241	DS line in use exclusively or by another subsystem.
242	Internal DS software malfunction.
243	Remote computer not responding (DS error).
244	Communications interface error. Remote computer reset the line.
245	Communications interface error. Receive timeout.
246	Communications interface error. Remote computer has disconnected.
247	Communications interface error. Local timeout.
248	Communications interface error. Connect timeout.
249	Communications interface error. Remote computer rejected connection.
250	Communications interface error. Carrier lost.
251	Communications interface error. The local data set for the DS line went not ready.
252	Communications interface error. Hardware failure.
253	Communications interface error. Negative response to the dial request by the operator.
254	Communications interface error. Invalid input/output configuration.
255	Communications interface error. Unanticipated error condition.

## 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 10-68

# FCLOSE

INTRINSIC NUMBER 9

Closes a file.

```
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. All magnetic tape files are left offline after an FCLOSE to indicate to the console operator that they may be removed.

The FCLOSE intrinsic can be used to maintain position when creating or reading a labeled tape file that is part of a volume set. If you close the file with a disposition code of 3, the tape does not rewind, but remains positioned at the next file. If you close the file with a disposition code of 2, the tape rewinds to the beginning of the file but is not unloaded. A subsequent request to open the file does not reposition the tape if the sequence (*seq*) subparameter is NEXT, or default (1). A disposition code of 1 (rewind and unload) implies the close of an entire volume set.

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. When a file is read that spans more than one volume of labeled magnetic tape, the user program is suspended until the operator has completed mounting the next tape. CCG is not returned when end-of-tape is encountered. 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.

## 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 (ignored by Foreign Disc Facility). 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. An unlabeled tape file is rewound. If the file resides on a labeled tape, the tape is rewound and unloaded.

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, and the file remains open, 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.

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. If the file is a disc file, the uniqueness of the file name is checked. If a file of the same name in the temporary file domain exists already, an error code is returned and the file remains open. If the file resides on unlabeled magnetic tape, the tape is rewound. If the file resides on labeled magnetic tape, the tape is backspaced to the beginning of the presently opened file.

3 = Temporary job file (not rewind). This option has the same effect as disposition code 2, except that tape files are *not* rewound. In the case of unlabeled magnetic tape, if this FCLOSE is the last done on the device (no other OPENS outstanding) the tape is rewound. If the file resides on a labeled magnetic tape, the tape is positioned to the beginning of the next file on the tape. Note that this disposition does not apply to disc files.

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 X for special considerations not here.

(12:1) Disc Space Disposition (for fixed length and undefined format files only).

1 = Returns to the system any disc space allocated beyond the end-of-file indicator. The EOF becomes the file limit. No records may be added to the file beyond this new limit.

Note: No space will be returned to the system if used with variable length files.

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

# FCLOSE

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 (ignored by Foreign Disc Facility). 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.

## 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. Additionally, an illegal disposition, 5, 6, or 7, was specified. This can be detected by FCHECK returning an error 49.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 10-39



# FCONTROL

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 on the user's stack, indicated by *param*. If this interval is zero, any previously established interval is cancelled, and no time out occurs. Controlcode 4 is ignored if the addressed file is 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 not valid for files accessed with append-only. Note that on a labeled magnetic tape file, the tape is positioned to the beginning of the opened file, and not necessarily to the beginning of the volume.

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 disc 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. This *controlcode* is not allowed for labeled magnetic tape files.

7 - Space Forward to Tape Mark. This moves a magnetic tape forward until a tape mark is encountered. If used on labeled magnetic tapes, the tape is positioned to the beginning of user trailer labels, if any.

8 - Space Backward to Tape Mark. On unlabeled tapes, this moves a magnetic tape backward until a tape mark is encountered. If used on labeled tapes, the tape is positioned to the beginning of user header labels, if any.

9 - Rewind and Unload Tape. This repositions a magnetic tape file at its beginning and places the tape offline. Not allowed for labeled tapes.

## NOTE

Control codes 0 and 3 will be rejected for spooled devicefiles. Control codes 5 through 9 (magnetic tape control) will be rejected for spooled :DATA tapes. Control codes 6 and 9 will be rejected for labeled magnetic tape files.

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.
- 45 = Enable/Disable extended wait.
- 46 = Enable/Disable reading writer's ID.
- 47 = Nondestructive read.

\*In Series 30/33 environment, FCONTROL code 36 only sets parity sense. You must additionally use control code 23/24 to disable/enable parity checking.

# FCONTROL

*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.

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 5-1 and 10-79

# FDEVICECONTROL

INTRINSIC NUMBER 53

Adds a variety of control directives to a spooled device file, currently only device files of the 2680 page printer.

```
IV    LA    IV    IV
FDEVICECONTROL (filenum, target, tcount, controlcode,
                LV    LV    I
                param 1,param2,errnum );
```

FDEVICECONTROL may be used to download character sets, forms and internal or control tables used in printing. It may also be used to control the page size, pen positioning, use of character sets and forms, and the number of copies of each page to be printed, along with other characteristics of the printing environment. The IFS/3000 intrinsics (which perform the same functions as FDEVICECONTROL), together with the layout of character set and form set load records and the Logical Page Table, are discussed in the IFS/3000 manual, part number 36580-90001.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the spoolfile. This value is obtained from FOPEN.
<i>target</i>	<i>logical array (required)</i> This array contains data to be passed to the 2680 printer. In general, it contains character sets, forms, or VFC information.
<i>tcount</i>	<i>integer by value (required)</i> The length of <i>target</i> in words or, if the value is negative, in bytes.
<i>controlcode</i>	<i>integer by value (required)</i> The code number of the operation to be performed.
<i>param 1, param 2</i>	<i>logical by value (required)</i> For each value of <i>controlcode</i> , there may be several possible values for <i>param 1</i> and <i>param 2</i> , which define the operation in more detail.
<i>errnum</i>	<i>integer (required)</i> The File System error code returned if an error occurs. Otherwise set to zero.

# FDEVICECONTROL

The following is a summary of the *controlcodes* described below:

<i>Controlcode</i>	<i>Meaning</i>
128	Character Set Selection
129	Logical Page Activation/Deactivation Request
130	Relative Pen Displacement
131	Absolute Pen Move
132	Define Job Characteristics
133	Define the Physical Page
134	Download/Delete Character Set
135	Download/Delete Form
136	Download Logical Page Table
137	Download Multi-copy Form Overlay Table
138	Download/Delete Vertical Format Control
140	Page Control
141	Clear Environment
142	Reserved
143	Load the Default Environment

*Controlcode* = 128          Character Set Selection

*param1* (8:8)      primary character set identification

*param2* (8:8)      secondary character set identification

The 2680 can contain up to 32 character sets, thus allowing the use of a variety of fonts, styles, print rotations, and languages. Use *controlcode* 134 to download character sets to the printer. Use this *controlcode* to select any two downloaded character sets to be the current primary and secondary character sets.

To change the secondary character set a character at a time, set the eighth bit of the byte coding for the desired ASCII character. The 2680 will strip out this bit and print, in the secondary character set, the character represented by the remaining 7 bit value. To change to the secondary character set for a number of characters and over several lines, insert a shift-out control character (control N) in the data. Insert a shift-in control character (control O) where you again wish to use the primary character set.

# FDEVICECONTROL

*Controlcode* = 129                      Logical Page Activation/Deactivation Request

- param1* (0:1)      deactivates the Logical Page Table entry identified in the left byte of *param2*.
- (1:1)      activates the Logical Page Table entry identified in the right byte of *param2*.
- param2* (0:8)      Logical Page Table entry (from 0 to 31) to be deactivated.
- (0:8)      Logical Page Table entry (from 0 to 31) to be activated.

Every physical page is composed of one or more logical pages. When the 2680 begins to print each physical page, it scans the Logical Page Table for the first logical page labeled as active. The printer then continues searching the table sequentially for active pages and printing them until it has printed the last active page. At this point the 2680 performs a physical page eject and starts the sequence again. There must be at least one active LPT entry while the 2680 is printing.

This *controlcode* allows you to cancel or enable the printing of logical pages during a job through the activation or deactivation of those pages.

*Controlcode* = 130                      Relative Pen Displacement

- param1*    is a 16 bit signed integer containing the desired X displacement of the pen from its current position.
- param2*    is a 16 bit signed integer containing the desired Y displacement of the pen from its current position.

No pen movement will result from requests to move the pen off of the logical page. As the coordinate system is based upon the current logical page itself and not upon the page's orientation with respect to the printer, you need not consider how the page has been rotated when assigning displacement values to *param1* and *param2*.

# FDEVICECONTROL

*Controlcode* = 131            Absolute Pen Move

*param1* is an integer containing the X coordinate of the point to which you wish to move the pen.

*param2* is an integer containing the Y coordinate of the point to which you wish to move the pen.

The values in *param1* and *param2* are measured from the upper left corner of the logical page. As with *controlcode* 130, you need not take page rotation into account when assigning coordinates, and the printer will not move the pen if the location you specify is off the logical page.

*Controlcode* = 132            Define Job Characteristics

*param1* (0:1) 1 — the printer will print no job separation marks until the next job is open.

(1:1) 1 — *param2* contains the maximum allowable number of copies of each page.

*param2* is significant only if *param1* (1:1) is set, and is the maximum number of copies the printer will make of any one page for the current job. The default maximum is 32,767.

The Spooler calls FDEVICECONTROL with this value of *controlcode* to set the maximum allowable number of copies per page. You may request any number less than or equal to this number by using *controlcode* 133.

# FDEVICECONTROL

*Controlcode* = 133            Define the Physical Page

- Param1* (0:1)    1— turn ON Multi Copy Form Overlay feature.
- (1:1)    1— turn OFF Multi Copy Form Overlay feature.
- (2:1)    1— reserved
- (3:1)    1— redefine the physical page length.
- (4:1)    1— redefine the number of copies of each page desired.
- (5:1)    1— reserved
- (6:1)    1— reserved
- (7:1)    1— reserved
- (8:8)    New physical page length in units of .25 inches. The length may not be less than 3.0 inches (a value of 12) or greater than 17.0 inches (a value of 68).

*param2* contains the number of copies of each page you want to print. If this number exceeds the maximum defined in *param2* of *controlcode* 132, only the maximum number of copies is printed.

Although FDEVICECONTROL will accept page length values that are multiples of .25 inches, the 2680 printer is able to produce only pages that are multiples of .5 inches. For this reason, only use even number values in *param1* (8:8). In other words, bit 15 must always be zero.

# FDEVICECONTROL

*Controlcode* = 134      Download/Delete Character Set

*Param1* (0:1)    0— Download of character set into the 2680.  
1— Purge the character set identified in the right hand byte of *param2* from the 2680.

*Param2* (0:1)    0— This is the first record of a load.  
1— This record is a continuation of the previous record.

(8:8)            Character set identifier (0-31)

If you attempt to download a character set having the same identifier as one already present in the printer, then the 2680 will purge the already present character set and repack the user area before loading the new font. However, before the modification of the user area, the 2680 prints all data currently in its buffer, as it does whenever you load, overlay, or delete a character set, form, or Vertical Format Control set.

*Controlcode* = 135      Download/Delete Form

*Param1* (0:1)    0— Load the form set identified in the right hand byte of *param2*.  
1— Purge the identified form set from the 2680 printer's memory.

*Param2* (0:1)    0— This is the first record of a load.  
1— This record is logically a continuation of the previous record.

(8:8)            Form set identifier (0:31).

FDEVICECONTROL will treat form sets with the same identifying integer in a way analogous to its treatment of character sets with the same identifiers. See *controlcode* 134.



# FDEVICECONTROL

*Controlcode* = 136

Download Logical Page Table

*Param1* is not used.

*Param2* (0:1) 0— This is the first record of a load.

1— This record is logically a continuation of the previous record.

A logical page is a page of data that may or may not take up an entire sheet of paper. It is possible to print up to 8 logical pages on one physical page. The Logical Page Table, 513 words long, contains some of the information needed to print up to 32 logical pages, so that the set of up to 8 logical pages printed on any one physical page may be varied.

*Controlcode* = 137

Download Multi-copy Form Overlay Table

*Param1* is not used.

*Param2* is not used.

This operation allows you to emulate a multi-part carbon by printing up to 8 copies of a page, each on one or two different forms. FDEVICECONTROL downloads into the printer's memory a table containing one word of information for each of the 8 possible copies to be overlaid with a form. The format of each word of the table is:

- Bit (0:1) — Form1 is to be overlaid on the physical page.
- (1:1) — Form2 is to be overlaid on the physical page.
- (2:4) — Reserved
- (6:5) — Form1 identifier — an integer from 0 to 31.
- (11:5) — Form2 identifier — an integer from 0 to 31.

# FDEVICECONTROL

*Controlcode* = 138            Download/Delete Vertical Format Control

*Param1* (0:1)    0— Load a VFC  
                  1— Delete a VFC

*Param2* (0:1)    0— This is the first record of a load.  
  
                  1— This record is logically a continuation of the  
                  previous record.

(8:8)    VFC set identifier (0-31).

The Vertical Format Control table is an ASCII file downloaded to the 2680 printer in order to give specific instructions as to the print density, location of top and bottom of page, and other specifications of the printed page. This table is further described and illustrated in chapter 4 of the Console Operator's Guide (part number 32002-90004).

The 2680 expresses the height of a printed line in dots, and the system uses this value to compute line positions on the page. Because these space measurements are relative to the top of the *logical* page, as opposed to the physical page, you may use the same or different Vertical Format Control tables for logical pages of different rotations.

*Controlcode* = 140            Page Control

*Param1* (15:1)    1— Do a physical page eject before going to the  
                  specified logical page. This bit has no effect if  
                  this is the first record since an environment load,  
                  FOPEN or FCLOSE.

(13:2)            Auto eject mode.  
                  0— Use auto eject flag of last data record (default at  
                  start of job is auto eject enabled).  
  
                  1— Enable auto eject (select VFC channel 1 on new  
                  page).  
  
                  2— Disable auto eject (position pen at top of page.)

*Param2* (8:8)    Logical page number (0 to 31).

The logical page identified in *param2* becomes the current logical page even if other logical pages have entries which precede it in the Logical Page Table. FDEVICECONTROL activates the specified page if it is inactive, and the 2680 performs a physical page eject if bit (15:1) of *param1* is set.

# FDEVICECONTROL

*Controlcode* = 141          Clear Environment

*param1* (0:1) 1 — clear all character sets  
(1:1) 1 — clear all forms  
(2:1) 1 — clear all Vertical Format Controls (VFCs)

*param2* is not used.

The printer will flush all data currently in its buffers, and then perform the indicated clears, if any.

*Controlcode* = 142          Reserved

*Controlcode* = 143          Load the Default Environment

*param1* is not used.

*param2* is not used.

The 2680 printer flushes all data, erases the user area, and loads the default character set, the Vertical Format Control (VFC), and the Logical Page Table (LPT).

## CONDITION CODES

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

## TEXT DISCUSSION

None.

# FDELETE

Deactivates a RIO record

```
IV      DV      O-V
FDELETE (filenum,recnum);
```

FDELETE deactivates a specified logical record. If no record is specified (or the *recnum* is negative), the *next* random access logical record becomes inactive. If the selected record has already been deactivated a CCE condition code is returned. The condition can be detected by calling the FCHECK intrinsic. The “inactive record” error indicates that the record selected for this FDELETE was already inactive.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the file to be deactivated.
<i>recnum</i>	<i>double by value (optional)</i> A positive double integer representing the relative logical record to be modified.

## CONDITION CODES

CCE	Request granted. No error (although inactive record may have been encountered).
CCG	Request denied. End of file.
CCL	Request denied. Access error.

## TEXT DISCUSSION

Page 10-9

# FERRMSG

INTRINSIC NUMBER 307

Returns message corresponding to FCHECK error number.

```
      I   LA   I  
FERRMSG(errorcode,msgbuf,msglgth);
```

The FERRMSG intrinsic causes a message to be returned to *msgbuf* that corresponds to an FCHECK error number. This makes it possible to display an error message from your program. The message describes the error associated with the error number provided in the *errorcode* parameter.

## PARAMETERS

<i>errorcode</i>	<i>integer (required)</i> A word identifier containing the error code for which a message is to be returned. It should contain an error number returned by FCHECK.
<i>msgbuf</i>	<i>logical array (required)</i> A logical array to which the message associated with <i>errorcode</i> is returned by FERRMSG. In order to contain the message string, <i>msgbuf</i> must be a maximum of 72 characters long.
<i>msglgth</i>	<i>integer (required)</i> A word identifier to which is returned the length of the <i>msgbuf</i> string. The length is returned as a positive byte count.

## CONDITION CODES

CCE	Request granted.
CCL	Request not granted because no error message exists for this <i>errorcode</i> or because of a message system error.
CCG	Request not granted. <i>msgbuf</i> address may be out of bounds, <i>msgbuf</i> may not be large enough, or <i>msglgth</i> address is out of bounds.

## TEXT DISCUSSION

Page 10-69

Provides access to file information.

```
IV      IV      BA
FFILEINFO (filename [,itemnum1, itemvalue1]
           [,itemnum2, itemvalue2]
           [,itemnum3, itemvalue3]
           [,itemnum4, itemvalue4]
           [,itemnum5, itemvalue5] );
```

## NOTE

*Itemnum/itemvalue* parameters must appear in pairs. Up to five items of information can be retrieved by specifying one or more *itemnum/itemvalue* pairs.

FFILEINFO provides access to file information. It is designed to be extensible so that new file information can be defined and accessed.

## PARAMETERS

<i>filename</i>	<i>integer by value (required)</i> MPE file number returned by FOPEN
<i>itemnum</i>	<i>integer by value (optional)</i> Cardinal number of the item desired; this specifies which item value is to be returned. (See item #, Figure 2-1a)
<i>itemvalue</i>	<i>byte array (optional)</i> Value of the item specified by the corresponding itemnum; the data type of the item value depends on the item itself. (See item, Figure 2-1a)

## CONDITION CODES

CCE	No error
CCG	Not used
CCL	Access or calling sequence error

## TEXT DISCUSSION

# FFILEINFO

ITEM #	TYPE	ITEM	UNITS
33	I	label type	(see Label Tapes)
34	I	current number of writers	(see IPC)
35	I	current number of readers	(see IPC)
36	L	File Allocation Date (CALENDAR format)	
37	D	File Allocation (CLOCK format)	
38	L	SPOOFLE Device file number (#O or #I number)	(see File Code)
39		RESERVED	
40	D	disc or diskette device status	
41	I	device type	
42	I	device subtype	
43	BA	environment file name	
44	I	last disc extent allocated	
45	BA	file name from labeled tape HDR1 record	
46	I	tape density	BPI
47	i	DRT number	
48	I	UNIT number	
49	I	software interrupt PLABEL	

Figure 2-1a. Item Values Returned by FFILEINFO (Continued)

# FFILEINFO

ITEM #	TYPE	ITEM	UNITS
1	BA	filename	(see FGETINFO)
2	L	foptions	(see FGETINFO)
3	L	aoptions	(see FGETINFO)
4	I	resize	(see FGETINFO) words/bytes
5	I	devtype	(see FGETINFO)
6	L	ldnum	(see FGETINFO)
7	L	hdaddr	(see FGETINFO)
8	I	filecode	(see FGETINFO)
9	D	recpt	(see FGETINFO)
10	D	eof	(see FGETINFO)
11	D	flimit	(see FGETINFO) records
12	D	logcount	(see FGETINFO) records
13	D	physcount	(see FGETINFO) records
14	I	blksize	(see FGETINFO) words/bytes
15	L	extsize	(see FGETINFO) sectors
16	I	numextents	(see FGETINFO)
17	I	userlabels	(see FGETINFO)
18	BA	creatorid	(see FGETINFO)
19	D	labaddr	(see FGETINFO)
20	I	blocking factor	(see FOPEN)
21	I	physical block size	words
22	I	data block size	words
23	I	offset to data in blocks	words
24	I	offset to Active Record Table within the block	(R10 files) words
25	I	size of Active Record Table	words
26	BA	vol. ID (label tape)	(see Label Tapes)
27	BA	vol. set ID (label tape)	(see Label Tapes)
28	I	expiration date (Julian format)	(see Label Tapes)
29	I	file sequence number	(see Label Tapes)
30	I	reel number	(see Label Tapes)
31	I	sequence type	(see Label Tapes)
32	I	creation date (Julian format)	(see Label Tapes)

Figure 2-1a. Item Values Returned by FFILEINFO



Requests access and status information about a file.

INTRINSIC NUMBER 11

```

      IV  BA  L  L  I  I  L  L
FGETINFO(filenum,filename,foptions,aoptions,recsize,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 seven 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	(0:3)	(3:1)	(4:1)	(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	(RESERVED)	RELATIVE I/O	KSAM FILE	DISALLOW :FILE	MPE TAPE LABELS	CARRIAGE CONTROL	RECORD FORMAT	DEFAULT DESIGNATOR	ASCII/BINARY	DOMAIN
MEANING		0 = Non-RIO file  1 = RIO file	0 = Not a new KSAM file (default)  1 = New KSAM file or existing KSAM file opened as an MPE file	1 = No :FILE  0 = :FILE	1 = LABELED TAPE  0 = NON-LABELED TAPE	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

NOTE: Double lines indicate octal digit boundaries.

# FGETINFO

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) — MPE Tape Label *Foption*.  
0 = Non-labeled tape.  
1 = Labeled tape.

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.

Bits (4:1) — KSAM file *Foption*  
0 = Not a new KSAM file (default)  
1 = New KSAM file or existing file opened as an MPE file.

Bits (3:1) — Relative I/O *Foption*  
0 = Non-RIO file will be created. (default)  
1 = RIO file will be created.

# FGETINFO

*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)	(3:1)	(4:1)	(5:1)	(6:1)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)
FIELD	(RESERVED)	KSAM ACCESS	NO WAIT I/O	(RESERVED)	MULTI ACCESS	INHIBIT BUFFERING	EXCLUSIVE ACCESS	DYNAMIC LOCKING	MULTI-RECORD ACCESS	ACCESS TYPE
MEANING		0 = KSAM access expected 1 = Non-KSAM access expected	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	0 000 = Read only 0 001 = Write only 0 010 = Write (save) only 0 011 = Append only 0 100 = Read/write 0 101 = Update 0 110 = Execute

Figure 2-2. Aoptions Bit Summary

NOTE: Double lines indicate octal digit boundaries.

# 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 (3:1) — KSAM Access *Aoption*

0 = KSAM access  
1 = Non-KSAM access expected; KSAM key file or data file is treated as standard MPE file. For this setting to be meaningful, file must be a KSAM file (*foptions* 4:1 = 1).

Bits (0:3) — Reserved for MPE.

*Default: Aoptions are not returned.*

- resize*                    *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), and 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. (See the System Manager/System Supervisor Manual.) If you have opened a serial disc tape and type returned in bits (8:8) is 31 (%37) even though the real device type is as specified in table 8-2 Classification of Devices. Device type %07 is returned by FGETINFO for foreign discs.  
*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. If the file is located on a remote computer, the left eight bits are the logical device number of the DS device and the right eight bits are the logical device number on the remote computer.  
*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. (See limitation under special considerations).  
 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.*

10-2





*creatorid*                    *byte array (optional)*  
A byte 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.  
*Default: The user name is not returned.*

*labaddr*                    *double (optional)*  
A double word to which is returned the sector address of the label of the file. The high-order eight bits show the logical device number. The remaining 24 bits show the absolute disc address. If the file is not a disc, zero is returned.  
*Default: The sector address is not returned.*

## CONDITION CODES

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

## SPECIAL CONSIDERATIONS

HDADDR Error Code 5 as obtained by FCHECK:

For large system configurations such as Series 64, if a DRT larger than 255 was requested by FGETINFO the request will be denied with a CCL condition code 5. To obtain a proper value for DRT and/or UNIT number, refer to the table of parameters in FFILEINFO.

## TEXT DISCUSSION

Page 10-66

# FINDJCW

Searches the Job Control Word table for a specified Job Control Word.

```
BA    L    I
FINDJCW (jcwname,jcwvalue,status);
```

## PARAMETERS

- jcwname*                    *byte array (required)*  
A byte array containing the name of the Job Control Word (JCW) to be found. May contain up to 255 alphanumeric characters, starting with a letter and ending with a non-alphanumeric character such as a blank.
- jcwvalue*                   *logical (required)*  
A word identifier to which is returned the value of *jcwname*, if *jcwname* is found. If *jcwname* is not found, *jcwvalue* is unchanged.
- status*                     *integer (required)*  
A word identifier to which is returned a value denoting the execution status of the intrinsic, as follows:
- 0 - Successful execution, *jcwname* found.
  - 1 - Error, *jcwname* greater than 255 characters long.
  - 2 - Error, *jcwname* does not start with a letter.
  - 3 - Error, *jcwname* not found in JCW table.

## CONDITION CODES

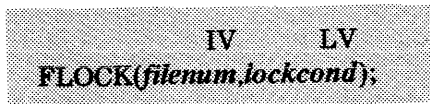
The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-47

Dynamically locks a file.

INTRINSIC NUMBER 15



The FLOCK intrinsic provides a means of signaling that the caller wants temporary exclusive use of a file.

## PARAMETERS

- filename*                      *integer by value (required)*  
A word supplying the file number of the file to be locked.
- lockcond*                      *logical by value (required)*  
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 *lockcond* = TRUE.
- CCL                      Request denied because this file was not opened with the *dynamic locking aoption* 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 *dynamic locking aoption* 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.

# FLUSHLOG

INTRINSIC NUMBER 213

Obtains information about the opened logging file.

D I  
FLUSHLOG ( *index, status* )

The FLUSHLOG intrinsic is used to write the contents of the user logging memory buffer to the disc destination file. This helps to preserve the contents of the memory buffer in the event of a system failure. This intrinsic writes no special records.

## PARAMETERS

<i>index</i>	<i>double (required)</i> The parameter returned from OPENLOG that identifies the user's access to the logging system.
<i>status</i>	<i>integer (required)</i> An integer in which error information is returned to the caller. Zero indicates OK status.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

None.

# FLOCK

## 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 10-55

# FMTCALENDAR

Converts any calendar date with the same format as the CALENDAR intrinsic into a format as follows:

FRI, AUG 5, 1977

```
LV BA  
FMTCALENDAR(date,string);
```

## PARAMETERS

<i>date</i>	<i>logical by value (required)</i> A logical value representing any calendar date with the same format as the CALENDAR intrinsic.
<i>string</i>	<i>byte array (required)</i> A 17-character byte array in which the formatted calendar date is returned.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-45

# FMTCLOCK

Converts the time of day with the same format as the CLOCK intrinsic into a format as follows:

7:39 AM

```
DV  BA
FMTCLOCK(time,string);
```

## PARAMETERS

<i>time</i>	<i>double by value (required)</i> A doubleword value representing the time of day with the same format as the CLOCK intrinsic.
<i>string</i>	<i>byte array (required)</i> An 8-character byte array in which the formatted time of day is returned.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-45

Converts calendar date and time of day with the same format as the CALENDAR and CLOCK intrinsics to a format, as follows:

FRI, AUG 5, 1979 7:39 AM

```
LV DV BA  
FMTDATE(date,time,string);
```

## PARAMETERS

<i>date</i>	<i>logical by value (required)</i> A logical value with the same format as the CALENDAR intrinsic.
<i>time</i>	<i>double by value (required)</i> A doubleword value with the same format as the CLOCK intrinsic.
<i>string</i>	<i>byte array (required)</i> A 27-character byte array in which the formatted date and time are returned.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-45



# FOPEN

INTRINSIC NUMBER 1

Used to establish access to a file and optionally define the physical characteristics of the file prior to setting up access to it.

```
      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 (\*). New KSAM files unlike standard files must be opened with a unique name.

*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 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 except mag tape and serial disc files are treated as ASCII files. Foreign disc files are binary records.

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. (Note that a leading \* in a formal designator can effectively override the disallow file equation *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. The file is composed of logical records of uniform length. Foreign discs always have fixed-length records.

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.

The actual physical record size used is determined by multiplying the *recsize* (specified or default) plus one by the *blockfactor*, and adding one word for the end-of-block indicator. 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. The file system makes no assumption about the amount of data that is useful. The user must determine how much data is good. For undefined length records, only the data supplied is written with no information about its length.

Note: Undefined-length records are supported by all devices; fixed- and variable-length records are supported by disc and 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.

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.

A record written with one of the above controls and no data (count = 0, or imbedded control and count = 1) will not cause physical I/O of any sort.

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 *resize* of 133 characters.

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

Bit (6:1) — Labeled Tape *Foption*.

1 = Labeled tapes.

0 = No labeled tapes.

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). Note that a leading \* in a formal designator can effectively override the disallow file equation *foption*.

0 = Allow :FILE.

1 = Disallow :FILE.

Bits (2:3) File Type *Foption*

Determines the type of file to create for a new file. If the file is old, this field is ignored.

000 - Ordinary file

001 - KSAM file

010 - Relative I/O file

100 - Circular file (discussed in Section III)

110 - Message file

Note: The Default Designator FOPTION, bits 10 through 12, offers several choices for default file designators. Any value used other than 0 for "filename" will override the File Type field.

Specification of both the KSAM and RIO options result in an access violation communicated by a CCL being returned. Specifying RIO in a :FILE command will override the KSAM option in the FOPEN call.

Bits (0:2) — Reserved for MPE. Should be 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.

# FOPEN

## 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. The end-of-file is not changed; the record pointer starts at 0.
- 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. The end-of-file is set to 0; the record pointer starts at 0. On magnetic tape an EOF mark will be written to the tape when the file is FCLOSED even if no data is written.
- 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. The end-of-file pointer is not changed, the record pointer starts at 0. Therefore, data will be overwritten if a WRITE is done.
- 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. The end-of-file pointer is used to set the record pointer prior to each FWRITE. For disc files it is updated (in an internal file system table) after each FWRITE. Thus, data in the file cannot be overwritten.
- 0100 = Input/output access. Any file intrinsic except FUPDATE can be issued for this file. The end-of-file pointer is not changed, the record pointer starts at 0.
- 0101 = Update access. All file intrinsics, including FUPDATE, can be issued for file. The end-of-file pointer is not changed; the record pointer starts at 0.
- 0110 = Execute access. Allows user with Privileged Mode Capability input/output access to any loaded file. The end-of-file pointer is not changed, the record pointer starts at 0.

## 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* (disc file only).

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.

# FOPEN

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. You must have LOCK access at account, group and file levels for FOPEN to grant the dynamic locking *option*. (LOCK Access is available if LOCK, APPEND, or WRITE access is set for you at these levels.)

0 = Disallow dynamic locking/unlocking.

1 = Allow dynamic locking/unlocking. A disc 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. This bit is ignored for non-disc files.

Bits (8:2) — Exclusive *Option*.

This *option* 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 *option* is in effect, an error code is returned to that calling process. The exclusive access *option* 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 *option* 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 *option* 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 *option* is selected, share access (11) takes effect. Otherwise, exclusive access (01) takes effect. Regardless of which access is selected, FGETINFO will report 00.

## 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 *recsize* 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.

Non-RIO access to a RIO file can be indicated by specifying the NOBUF option. In this case the physical blocksize (item #21) from FFILEINFO should be used to determine the maximum transfer length. (The FGETINFO "blksize" parameter may also be used.)

## Bits (5:2) — Multi-Access Mode *Aoption*

This feature permits processes located in different jobs or sessions to open the same file.

00 - No multi-access.

01 - Only intra-job multi-access allowed; this is the same as specifying the MULTI option in a FILE command.

10 - Inter-job multi-access allowed; this is the same as specifying the GMULTI option in a FILE command.

11 - Undefined. If this is specified, the FOPEN will be rejected with an error code of 40: ACCESS VIOLATION.

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. The No-Wait I/O *aoption* implies the NOBUF *aoption*; if you do not specify NOBUF, the file system does it for you. Also, multirecord access is not available. This option is not available if the file is located on a remote computer.

## NOTE

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

Bits (3:1) — File Copy *Aoption*

This feature permits any file to be treated as a standard sequential file, rather than as a file of its own type.

- 0 - The file will be accessed in its native mode; that is, a message file will be treated as a message file, a KSAM file as a KSAM file, etc.
- 1 - The file is to be treated as a standard, sequential file with variable-length records.

Note: In order to access a message file in copy mode, a process must have exclusive access to the file.

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.

When a foreign disc is opened, *resize* is forced to 128 words. (Note: IBM diskettes are forced to 64 words.)

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



To specify density when writing to the tape files, the keyword "DEN=" is used. The DEN keyword must be preceded by a semicolon (;), which indicates to the system that a keyword follows. Note that if the device parameter is specified, a semicolon must terminate the device string. If device is not specified, then a semicolon must be the first character of the device parameter. The keyword string must be terminated by a carriage return.

The density keyword is applicable only when writing to tape. When reading from a tape, the density selected by the user at FOPEN time will be ignored. For example, when reading from a tape, a 1600 BPI tape will satisfy an FOPEN request which specified ";DEN=6250", and vice-versa. The following examples show the correct syntax for the DEN= keyword:

```

BYTE ARRAY DEVICE(0:13):="TAPE;DEN=6250",%13;

BYTE ARRAY DEVICE(0:8):=";DEN=6250",%13;
      •
      •
      •
NUM:=FOPEN(FILEX,%4,%4,,DEVICE);

```

For more information on density selection, see Density Selection on Labeled and Unlabeled Tapes in Section X.

If you are opening a 2680 page printer, you may specify an optional printing environment for your job. The printing environment is defined as all of the characteristics of the printed page which are not part of the data itself, and thus would include the page size, the margin width, the character set, the orientation (horizontal or vertical), and the name of any forms you wish to use. Such information is contained in the *environment* file.

If you do not specify an environment file, FOPEN assumes that you want to use the default printer environment. HP provides a number of prepared environment files, which reside in the ENV2680A group of the SYS account. For information on how to build your own printing environments, see the IFS/3000 manual, part number 36580-90001.

To specify your own printer environment, you must also assign a keyword, ENV=, followed by the name of your environment file, to the *device* array, in the form, ENV=*environmentfilename*, and terminate the array with a carriage return. You must also include a semicolon between the device class name and the ENV keyword. For example, if PP is the device class name configured for your 2680 printer,

# FOPEN

```
EQUATE CR =13;  
BYTE ARRAY DEVICE(0:50) :=‘PP;ENV=MYENVFLE’,CR;  
•  
•  
•  
NUM:=FOPEN (FILEX,%4,%4,,DEVICE);
```

Any environment you select remains active until replaced by a new environment or until you FCLOSE the printer. If the printer has been opened with the multi-access AOPTION (e.g. as \$STDLIST), a selected environment remains active until replaced or until the final FCLOSE of the printer.

*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 the terminating period. Arrays with more than 49 characters are truncated by MPE.

# FOPEN

*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, and optionally specifying density for tape files (DEN = parameter), and/or environment file for the 2680 page printer (ENV = parameter). This parameter may be specified in one of the following forms:

*devclass*

*ldn*

\*

*\*vname*

*\*\*volname*

The *devclass* form represents a device class name of up to eight alphanumeric characters beginning with a letter, as for example, DISC or TAPE. If *devclass* is specified, the file is allocated to any available device in that class. If you are opening a file which is to reside on a private volume, you must specify device class DISC; the file then is allocated to any of the home volume set's volumes that fall within that device class.

The logical device number (*ldn*) consists of a three-byte numeric string specifying a particular device. If you are opening a file which is to reside on a private volume, you must specify a disc drive on which one of the volumes in the home volume set resides.

If you open a foreign disc file, *device* must be either a foreign disc class name or the *ldn* of a disc in a foreign disc class. When opening a foreign disc by logical device, the disc should be mounted on the drive, prior to the FOPEN. Otherwise it may be assumed to be a serial disc by the system.

The forms \*, *\*vname*, and *\*\*volname* are used only if you are opening a file which is to reside on a private volume.

If \* is specified, the file is allocated to any of the volumes of the home volume set.

If *\*vname* (volume class name) is specified, *vname* must be a member of the home volume set. The file then is allocated to any of the volumes within the volume class.

If *\*\*volname* (volume name) is specified, *volname* must be a member of the home volume set. The file then is allocated to that volume.

Any of the forms may be used to reference files on a remote computer by preceding the device or volume specification with DSDEVICE#.

This array also is used for tape label information if bit 6 of the *foptions* parameter is set. The 49-character limit does not apply in this case. The tape label information is formatted as follows:

```
. [volumeid] [, type] [, expdate] [, seq];
```

where

*volumeid* - Consists of six or fewer printable characters that identify the volume. In a multi-volume set, only the first *volumeid* can be specified.

*type* - Three alphabetic characters that identify label type information. Options are:

ANS - ANSI standard labels. (Default)  
IBM - IBM standard labels.

*expdate* - Month/day/year of the expiration date of the file or the date after which the information contained in the file is no longer useful. The file can be overwritten after this date. Default is 00/00/00, meaning that the file can be overwritten immediately. In a volume set, file expiration dates must always be equal to or earlier than the date on the previous file.

*seq* - Up to four characters that denote the position of the file in relation to other files on the tape. Default is "NEXT." A zero will cause a search of all volumes until file is found. If *seq*='ADDF,' then the tape will be positioned to add a new file on the end of the volume or last volume in a multi-volume set. If *seq*= NEXT, then the tape will be positioned to the next file on the tape. If this is the first FOPEN, then NEXT will cause the tape to be positioned to the first file on the tape. If the previous FCLOSE specified REWIND backspace to last file. The position will remain as it was on the previous file. (You cannot append a file on a labeled tape.)

*userlabels*

*integer by value (optional)*

An integer specifying the number of user-label records to be written for this file. Applicable to new disc files only. The maximum number of user labels allowed varies from file to file. It depends on the final blockfactor and reconfig used, as well as whether you have specified fixed, variable or undefined length records. If you specify more user labels than will fit in the 254 sectors following the MPE file label, an error occurs and the FOPEN fails.

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

*blockfactor*

*integer by value (optional)*

An integer containing the size of the 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 255. *Blockfactor* establishes the physical record size on disc and magnetic tape files.

The *blockfactor* for foreign disc files is  $> = 1$ .

*Default: Calculated by dividing the specified resize (in words) into the configured blocksize. This value is rounded downward to an integer that is never less than 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, because 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 blocks 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 X.

The *filesize* for foreign disc files is set to the maximum physical size of the disc as determined by its subtype.

Because of spare tracks, remapped tracks, etc., the logical size will usually be smaller than the physical size.

*Default: 1023 logical records.*

*numextents*

*integer by value (optional)*

An integer specifying the maximum 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 always calculated in terms of physical records. When the file is type F (fixed) *filesize* is the number of logical records; thus it will be divided by the blockfactor to determine the number of physical records (blocks). If the file is variable length or undefined length, *filesize* is the number of physical records. Then, the number of physical records required for the system file label, and user labels (if any), are added to the number of physical records required for data. To determine extent size, the number is divided by the requested *numextents*. The result rounded up, is the number of physical records per extent. This is then used to determine the actual number of extents and the size of each. If specified, *numextents* must be an integer from 1 to 32. A zero or negative value results in the default setting. Any value >32 will automatically be set to 32.

*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. For this parameter, any user can specify a positive integer ranging from 0 to 1023. This parameter is used for new files only when it is positive and

# FOPEN

the process is running in user mode. 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 “privileged” file must be requested in privileged mode. A process running in user mode cannot open a file that has a negative *filecode*. Also, if the process supplies a non-zero parameter the *filecode* must match the one originally specified for the file. Certain integers have particular HP-defined meanings, as follows:

Mnemonic	Integer	Meaning
USL	1024	USL file.
BASD	1025	BASIC data file.
BASP	1026	BASIC program file.
BASFP	1027	BASIC fast program file.
RL	1028	RL file.
PROG	1029	Program file.
SL	1031	SL file.
VFORM	1035	VIEW formsfile.
VFAST	1036	VIEW fast forms file.
XLSAV	1040	Cross Loader ASCII file (SAVE).
XLBIN	1041	Cross Loader relocated binary file.
XLDSP	1042	Cross Loader ASCII file (DISPLAY).
EDITQ	1050	Edit KEEPQ file (non-COBOL).
EDTCQ	1051	Edit KEEPQ file (COBOL).
EDTCT	1052	Edit TEXT file (COBOL).
RJEPN	1060	RJE punch file.
QPROC	1070	QUERY procedure file.
	1071	QUERY work file.
	1072	QUERY work file.
KSAMK	1080	KSAM key file.
LOG	1090	User Logging logfile.

Integer	Meaning
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 } 1072 }	QUERY work files.
1080	KSAM Key file.
1090	User Logging

*Default: 0.*

## 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 10-27



# FPOINT

INTRINSIC NUMBER 6

Sets the logical record pointer for a disc file.

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

The FPOINT intrinsic sets the logical record pointer for a disc file, containing fixed-length or undefined 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

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

*recnum*                    *double by value (required)*  
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.

### NOTE

On disc files, the end-of-file indicator is the file limit. Magnetic tape files are delimited by the end-of-file marker.

## CONDITION CODES

CCE                    Request granted.

CCG                    Request denied. The logical record pointer position is unchanged. Positioning was requested at a point beyond the file limit.

CCL                    Request denied. The logical record pointer position is unchanged because of one of the following:  
*recnum* was <0.  
Invalid *filenum* 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 or undefined-length records.  
Not allowed with append access.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 10-91

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

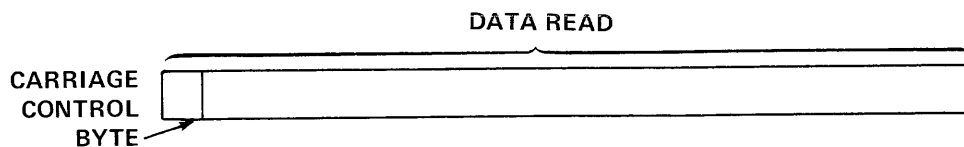
INTRINSIC NUMBER 2

```
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. When a file is read that spans more than one volume of labeled magnetic tape, the user program is suspended until the operator has completed mounting the next tape. CCG is not returned when end-of-tape is encountered. 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

- |                |   |
|----------------|---|
| <i>filenum</i> | <i>integer by value (required)</i><br>A word identifier supplying the file number of the file to be read.   |
| <i>target</i>  | <i>logical array (required)</i><br>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. |

# FREAD

*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 multi-record *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. When reading a labeled magnetic tape file that spans more than one volume, CCG is not returned when end-of-tape (EOT) is encountered. Instead, CCG is returned at actual end-of-file, with a transmission log of 0 if an attempt is made to read past end-of-file.
CCL	The information was not read because an error occurred, a terminal read was terminated by a special character or timeout interval 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 10-47

# FREADBACKWARD

Reads a logical record beginning at a point prior to the current record pointer.

INTRINSIC NUMBER 2

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

The FREADBACKWARD intrinsic reads a logical record from a tape to the user's stack. The record read is determined by the current position of the record pointer. This intrinsic permits access to the Read Reverse capability of HP-IB Magnetic Tape Drives, and can be used to recover tape errors when handling I/O management and data recovery routines.

Presently two substantial restrictions are associated with the use of this intrinsic:

1. It may only be used with Magnetic Tape Drives on HP-IB Systems (i.e., Series 30, 33, 44, and Series III with HP-IB Interface Module).
2. The magnetic tape must be accessed NOBUF.

## FUNCTIONAL RETURN

The FREADBACKWARD intrinsic returns a positive integer value showing the length of the information transferred. If the *tcount* parameter in the FREADBACKWARD 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. FREADBACKWARD 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

<i>filenum</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 the length in words; if it is negative, it signifies the length in bytes; 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 *tcount* is larger than the size of the logical record, and the multi-record *aoption* was not specified in FOPEN, transfer is limited to the length of the logical record. If the multi-record *aoption* was specified in FOPEN, transfer continues until either *tcount* is satisfied or the beginning-of-data is encountered, and each transfer will begin at the end of the next physical record (i.e., block). Any data remaining in the last physical record read will be inaccessible.

# FREADBACKWARD

## CONDITION CODES

CCE	The information was read.
CCG	The logical beginning-of-data was encountered during reading. When reading a labeled magnetic tape file that spans more than one volume, CCG is not returned when beginning-of-tape (BOT) is encountered. Instead, CCG is returned at actual beginning of file, with a transmission log of 0 if an attempt is made to read past beginning of file.
CCL	The information was not read because a tape error occurred, 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

None.

# FREADDIR

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

INTRINSIC NUMBER 7

```
IV LA IV DV
FREADDIR(filenum,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. If RIO access is used, FREADDIR will input the specified logical record. If the record is inactive, the contents of the inactive record will be transmitted and a CCE will be returned. (FCHECK returns a non-zero error number to distinguish active and inactive records. If a RIO file is accessed using the non-RIO method, (NOBUF) FREADDIR will input the specified block. In this case there is no indication whether the block contains some inactive 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>filenum</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).

# **FREADDIR**

## **CONDITION CODES**

CCE	The information was read.
CCG	End of file was encountered.
CCL	The information was not read because an error occurred.

## **SPECIAL CONSIDERATIONS**

Split stack calls permitted.

## **TEXT DISCUSSION**

Page 10-49

# FREADLABEL

Reads a user file label.

INTRINSIC NUMBER 19

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

The FREADLABEL intrinsic reads a user-defined label from a disc file or magnetic tape file. Once a disc file has been opened, user labels may be read from or written to regardless of the opener's access to the rest of the file. A disc file can have up to 254 128-word user labels. A magnetic tape file must be labeled with an ANSI-standard or IBM-standard label. 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. If the file is on labeled magnetic tape, the user-defined label must be 40 words in length to conform to the length of the ANSI or IBM-standard label. Refer to Appendix D for the format of magnetic tape labels.

## 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: Zero 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



# FREADSEEK

INTRINSIC NUMBER 12

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

```
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 10-50

Releases an extra data segment.

INTRINSIC NUMBER 131

```
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

<i>index</i>	<i>logical by value (required)</i> A word containing the logical index assigned to the data segment, obtained from the GETDSEG intrinsic call.
<i>id</i>	<i>logical by value (required)</i> 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 <i>index</i> is invalid or <i>index</i> and <i>id</i> do not specify the same extra data segment.

## SPECIAL CONSIDERATIONS

Data-Segment Management Capability required.

## TEXT DISCUSSION

Page 8-15.

# **FREELOCRIN**

INTRINSIC NUMBER 31

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

## **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.

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 0 = 1, *infilenum* and *listfilenum* form a duplicative pair.
- If bit 0 = 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.

# FRELATE

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 10-92

# FRENAME

Renames a disc file.

INTRINSIC NUMBER 17

IV	BA
FRENAME( <i>filenum</i> , <i>newfilereference</i> );	

The FRENAME intrinsic changes the actual designator (including *lockword*, if any) of an open disc file. The home volume set of *newfile reference* must be the same as that of the file being renamed. (Volume sets cannot be spanned when renaming files.)

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). If the file is a permanent file, you must be the creator.

## 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.

# FRENAME

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 10-43





# FSETMODE

## 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 10-91

Spaces forward or backward on a file.

INTRINSIC NUMBER 5

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. If RIO access is used, the displacement includes all records regardless of activity state (i.e., active deleted). Attempts to backspace beyond the beginning of the file will be ignored by the system. The logical record pointer will point to record 0 ( the first record) and no error codes will be returned.

## CONDITION CODES

CCE Request granted.

CCG An end-of-file indicator was encountered during spacing. For disc files, this is the file limit, and the logical record pointer is not changed. For magnetic tape files, it is the end-of-file mark, and 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. For labeled tape the logical record pointer is at the file mark.

CCL Request denied because an error occurred; for example file resides on a device that prohibits spacing.  
Not allowed with append access.

# FSPACE

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 10-89

Dynamically unlocks a file.

INTRINSIC NUMBER 16

**IV**  
**FUNLOCK(*filenum*);**

The FUNLOCK intrinsic dynamically unlocks a file (Resource Identification Number) that has been locked with the FLOCK intrinsic.

## PARAMETERS

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

## CONDITION CODES

CCE	Request granted
CCG	Request denied because the file had not been locked by the calling process.
CCL	Request denied because the file was not opened with the dynamic locking <i>aoption</i> of the FOPEN intrinsic, or the <i>filenum</i> parameter is invalid.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 10-55

# 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. If RIO access is used, the modified record is set to the active state.

Note that FUPDATE is functionally equivalent (but faster) to FSPACE(FILENUM, -1); followed by an FWRITE to FILENUM.

## 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 update more than one block in multirecord mode.

## 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 block when multirecord mode is not in effect.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 10-57

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 or physical 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. If RIO access is used, the modified record is set to the active state.

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 or when an extent is allocated. If the physical bounds of the file are reached, the CCG condition code is returned.

If a magnetic tape is unlabeled (as specified in the FOPEN intrinsic or :FILE command) and a user program attempts to write over or beyond the physical EOT 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).

If a magnetic tape is labeled (as specified in the FOPEN intrinsic or :FILE command), a CCL condition code is not returned when the tape passes the EOT marker. Attempts to write to the tape after the EOT marker is encountered cause end of volume (EOV) markers to be written. A message then is printed on the operator's console requesting another volume (reel of tape) to be mounted.

## 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>resize</i> parameter associated with the record, only the first <i>tcount</i> words or bytes are written.

If *tcount* is larger than the logical record size and the NOBUF *option* is not specified in FOPEN, the FWRITE request is refused and condition code CCL is returned. If NOBUF is specified, *tcount* may not exceed the physical record size unless the multirecord option is specified.

# FWRITE

If the multirecord *aoption* is specified in FOPEN, the excess words or bytes are written to succeeding physical records. For files for which carriage control is specified, the actual data transferred is limited to *recsize* minus one byte.

## *control*

*logical by value (required)*

A logical value representing a carriage control code, effective if the file is transferred to a line printer or terminal (including 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.

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 satisfy 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/ NOCCTL 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.

Sometimes it is necessary to set the pre-space/post space control and the auto/no auto page eject control with the FWRITE instead of FCONTROL. You may use control codes %100 through %103 and %400 through %403 for this. If you specify one of the above controls with + count = 0, no physical I/O will occur.

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 NOCCTL. 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 *tcount* 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. If the file is being written to a multi-volume magnetic tape set, CCL is not returned when the end-of-tape marker is sensed. Instead, end-of-volume labels are written, and a request is issued to mount the next volume.

OCTAL CODE	ASCII SYMBOL	CARRIAGE ACTION
%40	" "	Single space (with or without automatic page eject).
%53	"+"	No space, return (next printing at column 1). Not valid on 2607/2608 (results in single space without automatic page eject).
%55	"_"	Triple space (with or without automatic page eject).*
%60	"0"	Double space (with or without automatic page eject).*
%61	"1"	Page eject (form feed). Selects VFC Channel 1. Ignored if:  Post-space mode: The current request has a transfer count of 0 and the previous request was an FOPEN or FCLOSE or an FWRITE which specified a carriage-control directive of %61.  Pre-space mode: Both the current request and the previous request have transfer counts of 0, and the current request and previous request are any combination of FOPEN, FCLOSE or an FWRITE specifying a carriage-control directive of %61.
%2nn (nn is any octal number from 0 through 77)		Space nn lines (no automatic page eject). %200 not valid for 2607 (results in single space without automatic page eject).
%300 - %307		Select VFC Channel 1-8 (2607)
%300 - %313		Select VFC Channel 1-12 (2613, 2617, 2618, 2619)
%300 - %317		Select VFC Channel 1-16 (2608)

**\*Note:**

*Series 30/33/44:* If these codes are selected with automatic page eject in effect (by default or following an Octal Code of %102 or %402), the resulting skip is to a location absolute to the page. A code of %60 is replaced by %303 and %61 is replaced by %304. Thus the resulting skip may be less than two or three lines, respectively.

If automatic page eject is not in effect, a true double or triple space results, but the perforation between pages is not automatically skipped.

*Series 11/111:* If these codes are selected with automatic page eject in effect, %60 and %61 are replaced by two or three %302 codes, respectively. This results in true double or triple spacing, and also skips the perforation.

If automatic page eject is not in effect, the behavior is the same as for Series 30/33/44.

Figure 2-3. Carriage-Control Directives (Sheet 1 of 2)



# FWRITE

OCTAL CODE	ASCII SYMBOL	CARRIAGE ACTION
		NOTE Channel assignments shown below are the HP standard defaults.
%300		Skip to top of form (page eject).
%301		Skip to bottom of form.
%302		Single spacing with automatic page eject.
%303		Skip to next odd line with automatic page eject.
%304		Skip to next third line with automatic page eject.
%305		Skip to next 1/2 page.
%306		Skip to next 1/4 page.
%307		Skip to next 1/6 page.
%310		Skip to bottom of form.
%311		User option (2613/17/18/19), skip to one line before bottom of form (2608)
%312		User option (2613/17/18/19), skip to one line before top of form (2608)
%313		User option (2613/17/18/19), skip to top of form (2608)
%314		Skip to next seventh line with automatic page eject.
%315		Skip to next sixth line with automatic page eject.
%316		Skip to next fifth line with automatic page eject.
%317		Skip to next fourth line with automatic page eject.
%320		No space, no return (next printing physically follows this).
%2 - %37		
%41 - %52		
%54		
%56 - %57		
%62 - %77		
%104 - %177		
%310-%317	(2607)	
%314-%317	(2613/17/18/19)	
%321 - %377		
%400 or %100		Sets post-space movement option; this first prints, then spaces. If previous option was pre-space movement, the driver outputs a line with a skip to VFC channel 3 to clear the buffer.
%401 or %101		Sets pre-space movement option; this first spaces, then prints.
%402 or %102		Sets single-space option, with automatic page eject (60 lines per page).
%403 or %103		Sets single-space option, without automatic page eject (66 lines per page).

Figure 2-3. Carriage-Control Directives (Sheet 2 of 2)

FOPEN OR :FILE	FWRITE Control Parameter		
	= 0	= 1	= Greater than 1
Carriage Control Foption Specified or CCTL	<p style="text-align: center;">Data output contains 132 characters; the prefix byte is added and contains 0</p>	<p style="text-align: center;">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 style="text-align: center;">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 style="text-align: center;">Data output contains 132 characters.</p>	<p style="text-align: center;">Data output contains 132 characters</p>	<p style="text-align: center;">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 10-49

# FWRITEDIR

INTRINSIC NUMBER 8

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

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

The FWRITEDIR intrinsic writes a specific logical record (or physical record if NOBUF is specified), 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.

If RIO access is used, the modified record is set to the active state.

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 new record is in the same extent.

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

<i>filenum</i>	<i>integer by value (required)</i> A word identifier specifying the file number of the file to be written on.
<i>target</i>	<i>logical array (required)</i> Contains the record to be written. 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 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>recsize</i> parameter associated with the record, and NOBUF was specified, only the first <i>tcount</i> words or bytes are written.

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

*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 10-51

# FWRITELABEL

INTRINSIC NUMBER 20

Writes a user file label.

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

The FWRITELABEL intrinsic writes a user-defined label onto a disc file or labeled magnetic tape file that is labeled with an ANSI-standard or IBM-standard label. This intrinsic overwrites old user labels. Once a disc file has been opened, user labels may be read from or written to regardless of the user's access to the rest of the file. If the file is on labeled magnetic tape, the user-defined label must be 40 words in length to conform to the length of the ANSI or IBM-standard label. Refer to Appendix D for the format of magnetic tape 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. If the file is a labeled magnetic tape file, this label must be 40 words in length.
<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: Zero 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 10-62

# GENMESSAGE

Accesses message system.

```
      I           IV   IV   IV  BA   IV   LV
msglen:=GENMESSAGE(filenum,setnum,msgnum,buff,buffsize,parmask,
                    LV  LV  LV  LV  LV   IV   I   O-V
                    parm1,parm2,parm3,parm4,parm5,msgdest,errnum);
```

The GENMESSAGE intrinsic accesses the MPE message system. A message number is passed by GENMESSAGE to the message system. The message system fetches the message from a message catalog (opened by the calling program), inserts parameters supplied by GENMESSAGE into the message, then routes the message to a file or returns the message to the calling program. (If *msgdest* is specified, the message is routed to a file. If *buff* is specified, the message is returned. If both *msgdest* and *buff* are specified, the message is both routed to a file *and* returned.)

## NOTE

The catalog file must be opened with *foptions* old, permanent, ASCII (*foptions* 5), and *aoptions* nobuf and multi-record access (*aoptions* %420).

## FUNCTIONAL RETURN

The length of the message is returned (in positive bytes) to *msglen*.

## PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word identifier supplying the file number of the message catalog.
<i>setnum</i>	<i>integer by value (required)</i> A positive integer no greater than 62 specifying the message set number within the catalog.
<i>msgnum</i>	<i>integer by value (required)</i> A positive integer, specifying the message number within the message set.
<i>buff</i>	<i>byte array (optional)</i> A byte array to which the assembled message is returned. <i>Default: Message is not returned to calling program.</i>
<i>buffsize</i>	<i>integer by value (optional)</i> When <i>buff</i> is specified, <i>buffsize</i> is the size, in bytes, of the buffer. When <i>buff</i> is not specified, <i>buffsize</i> is the length, in bytes, of the records written to the destination file. <i>Default: 72 bytes.</i>
<i>parmask</i>	<i>logical by value (optional)</i> A 16-bit logical mask indicating parameter types for <i>parm1</i> , <i>parm2</i> , <i>parm3</i> , <i>parm4</i> , and <i>parm5</i> . The bit settings are as follows:

# GENMESSAGE

Bit (0:1)=1. Ignore rest of word and parameters *parm1* through *parm5*.

=0. Rest of word, in 3-bit groupings, will specify parameter types for *parm1*, *parm2*, *parm3*, *parm4*, and *parm5*, as follows:

Bits (1:3)=*parm1* type.

0 - parameter is a string, terminated by an ASCII null (0).

1 - parameter is integer.

2 - parameter is double by reference.

3 - ignore the parameter.

Bits (4:3) = *parm2* type (types same as for *parm1*).

Bits (7:3) = *parm3* type (types same as for *parm1*).

Bits (10:3)= *parm4* type (types same as for *parm1*).

Bits (13:3)= *parm5* type (types same as for *parm1*).

*Default: Parameters parm1 through parm5 will be ignored.*

*parm1*

*logical by value (optional)*

Parameter to be inserted into message. If *parmask* specifies a type of 0 (string), *parm1* must pass the byte address (i.e., @stringarray) of the byte array containing the string. If *parmask* specifies type 2 (double by reference), *parm1* must pass the word address (i.e., @doubleword) of the doubleword identifier containing the value.

*parm2*

*logical by value (optional)*

Parameter to be inserted into message. Description is the same as for *parm1*.

*parm3*

*logical by value (optional)*

Parameter to be inserted into message. Description is the same as for *parm1*.

*parm4*

*logical by value (optional)*

Parameter to be inserted into message. Description is the same as for *parm1*.

*parm5*

*logical by value (optional)*

Parameter to be inserted into message. Description is the same as for *parm1*.

*msgdest*

*integer by value (optional)*

Integer value specifying the destination of the assembled message, as follows:

0 = \$STDLIST.

>2 = File number of destination file.

*Default: \$STDLIST if buff is not specified, no file if buff is specified.*

# GENMESSAGE

*errnum*

*integer (optional)*

Integer identifier to which an error number is returned. Values returned are as follows:

- 0 = Successful execution.
- 1 = FREADLABEL failed on catalog file.
- 2 = FREAD failed on catalog file.
- 3 = Specified *setnum* not found in catalog.
- 4 = Specified *msgnum* not found in catalog.
- 6 = Assembled message overflowed buffer (if *msgdest* was specified, however, message routed correctly).
- 7 = write failed to destination file.
- 8 = Catalog file opened with improper access options.
- 11 = *filenum* parameter not specified.
- 12 = *setnum* parameter not specified.
- 13 = *msgnum* parameter not specified.
- 14 = *setnum*  $\leq 0$ .
- 15 = *setnum*  $> 62$ .
- 16 = *msgnum*  $\leq 0$ .
- 17 = *bufsize*  $\leq 0$ .
- 18 = *msgdest*  $< 0$ .

## CONDITION CODES

CCE	Successful execution.
CCL	Intrinsic did not execute because of file system error.
CCG	Intrinsic did not execute. May have missing required parameter, invalid parameter, or invalid file number of catalog or destination file. CCG is also returned if SET or MSG is not found.

## TEXT DISCUSSION

Page 4-50



# GET

Receives the next request from the remote master program.

```
I      IA I      I  
ifun.=GET(itag,il,ionumber);
```

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

Creates an extra data segment.

INTRINSIC NUMBER 150

```
      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. If the intrinsic is called in user mode, then the data segment is initially filled with zeros.

## PARAMETERS

<i>index</i>	<i>logical (required)</i> A word to which the logical index of the data segment, assigned by MPE, is returned.
<i>length</i>	<i>integer (required)</i> 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.
<i>id</i>	<i>logical by value (required)</i> 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, <i>id</i> is specified as a non-zero value. If a data segment with the same <i>id</i> exists already, it is made available to the calling process. Otherwise, a new data segment, sharable within the job/session, is created with this <i>id</i> . For a private data segment, an <i>id</i> 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.

# GETDSEG

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), sufficient storage was not available for the data segment (*index* is set to %2002) or a stack expansion necessary to satisfy the request could not be done because the stack was frozen (*index* set to %2003). Note that a stack expansion is usually not necessary to get an extra data segment.

## **SPECIAL CONSIDERATIONS**

Data Segment Management Capability required.

## **TEXT DISCUSSION**

Page 8-6.

Fetches content of system job control word (JCW).

INTRINSIC NUMBER 73

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

The GETJCW intrinsic returns the complete job control word (JCW) 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 or PUTJCW intrinsic.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-46

# GETLOCRIIN

INTRINSIC NUMBER 30

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

```
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

Determines source of activation call.

INTRINSIC NUMBER 105

```
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.
- 0 = Activated from some other source.

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-14.

# GETPRIORITY

INTRINSIC NUMBER 120

Reschedules a process.

```
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.

The GETPRIORITY intrinsic will abort the calling process if the requested *priorityclass* exceeds the maximum allowable *priorityclass* of the rescheduled process or specifies an invalid priority class.

## 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 running in Privileged Mode, the *priorityclass* parameter may be specified as an absolute number 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. Note that an absolute priority must be specified in order to overcome the MAXPRI setting of an account.

## NOTE

Scheduling a process into the "AS" or "BS" priority class (assuming the process's maximum priority allows such a specification) can result in the rescheduled process deadlocking the system or locking out system and user processes from execution.

- 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.  
Must be running in Privileged Mode to specify absolute priority.

## 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.

# GETPROCID

Requests PIN of a son process.

INTRINSIC NUMBER 112

```
I      IV
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*

*integer 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	An unexpected end-of-file was encountered when writing to <i>uslfnm</i> .
1	Unexpected input/output error occurred.

## TEXT DISCUSSION

See the *MPE Segmenter Reference Manual*

# IODONTWAIT

INTRINSIC NUMBER 22

Initiates completion operations for an I/O request.

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

The IODONTWAIT intrinsic behaves the same as IOWAIT (see page 2-128) with one exception: If IOWAIT is called and no I/O has completed, then the calling process is suspended until some I/O completes; if IODONTWAIT is called and no I/O has completed, then control is returned to the calling process (CCE is returned and the result of IODONTWAIT is zero.)

## FUNCTIONAL RETURN

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

## PARAMETERS

- filenum* *integer by value (required)*  
A word identifier specifying the file number for which there is a pending I/O request. If zero is specified, the IODONTWAIT intrinsic will check for any I/O completion.
- target* *logical array (optional)*  
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. It should be the same buffer specified in the original I/O request if that request was a read. This allows for proper recognition of :EOD (AND :) where applicable.
- tcount* *integer (optional)*  
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 *tcount* parameter of IODONTWAIT.  
*Default: The length of the record is not returned.*
- cstation* *logical (optional)*  
Used for distributed systems to return the number of the calling station which completed.

# IODONTWAIT

## 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.

## TEXT DISCUSSION

Page 10-62



# IOWAIT

INTRINSIC NUMBER 22

Initiates completion operations for an I/O request.

```
I          IV  LA  I  L  O-V  
fnum:=IOWAIT(filename,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>filename</i>	<i>integer by value (optional)</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. It should be the same buffer specified in the original I/O request if that request was a read. This allows for proper recognition of :EOD (and:) where applicable.
<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> Used for distributed systems to return the number of the calling station which completed.

## 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.

## TEXT DISCUSSION

Page 10-59

# New Intrinsic JOBINFO

By Larry Cargnoni

The JOBINFO intrinsic is new for MPE V/E, and will execute only on an HP 3000 supporting MPE V/E. It enables a standard user to access session- and job-related information. Previously, most of this information was accessible only through MPE commands and the WHO intrinsic. The JOBINFO intrinsic is patterned after the FFILEINFO and CREATEPROCESS intrinsics. It is callable from any language, and may be used in software that performs security checks, job stream polling, system accounting, and job/session communication. The JOBINFO intrinsic provides access to information related to any job/session that is current to the system. This intrinsic is expandable, and is written so that the addition of further functionality will be straightforward.

## SYNTAX

IV	D	LA	IV	LA	I	OV
JOBINFO	( <i>jsind</i> ,	<i>JS#nnn</i> ,	<i>status</i>	[ , <i>itemnum1</i> ,	<i>item1</i> ,	<i>errornum1</i> ]
				[ , <i>itemnum2</i> ,	<i>item2</i> ,	<i>errornum2</i> ]
				[ , <i>itemnum3</i> ,	<i>item3</i> ,	<i>errornum3</i> ]
				[ , <i>itemnum4</i> ,	<i>item4</i> ,	<i>errornum4</i> ]
				[ , <i>itemnum5</i> ,	<i>item5</i> ,	<i>errornum5</i> ] );

## PARAMETERS

*jsind*

*integer by value (required)*

An integer indicating whether the *JS#nnn* is a session or job:

1= *JS#nnn* is a session.

2= *JS#nnn* is a job.

*JS#nnn*

*double (required)*

A double value, 32 bits, identifying the job or session for which information will be retrieved.

*status*

*logical array (required)*

A two word logical array reporting the overall success/failure of the call. Only the first word contains significant information.

0= Successful call. All *errornums* equal zero.

1= Semi-successful call. One or more *errornums* were returned with non-zero values.

- 2= Unsuccessful call. All *errornums* were returned with nonzero values.
- 3= Unsuccessful call. Syntax error in calling sequence.
- 4= Unsuccessful call. Unable to retrieve *JS#nnn/S#nnn*.
- 5= Process died during the start of retrieval.

*itemnum**integer by value (optional)*

Cardinal number of the item desired. This specifies which item value is to be returned (refer to "ITEM#" in Table 1).

*item**logical array (optional)*

Name of a reference parameter (whose data type corresponds to the data type for the desired information) to which the desired information is returned (refer to "ITEM" in Table 1).

*errornum**integer (optional)*

A returned integer specifying the success or failure of the retrieval of each item. The returned values are:

- 0= Successful information retrieval.
- 1= Invalid *itemnum* (item number).
- 2= Desired information not pertinent to the given *JS#nnn* (eg., user specifies a session number and wishes to know if a job had RESTART option).
- 3= User has insufficient capability to access this information.
- 4= The desired information is no longer available (eg., when spoolfiles disappear).

## SPECIAL CONSIDERATIONS

A user without System Manager (SM) or Account Manager (AM) capability can only retrieve information about the jobs/sessions logged on under the user name and account. A user with AM capability but not SM capability will be restricted to information concerning his account sessions and jobs; a user with SM capability will be able to retrieve information concerning all sessions and jobs. The exception to the above security is access to items which are normally available to a user, through MPE commands, who does not have any special capabilities.

## CONDITION CODES

There are no condition codes in the traditional sense, but the *status* parameter can be thought of as a condition code.

The *status* parameter returns a number representing the overall status of the call. The *errornum* parameter returns the status of the individual accesses *items* and *itemnums*. Combinations of successful and unsuccessful data retrievals could be returned from the same call. For example, a user who does not have System Manager or Account Manager capabilities writes a program with *JOBINFO*. The *JOBINFO* intrinsic retrieves the jobfence and the current job step of access user. Upon the return of *JOBINFO*, the parameter *status* will return a 2 (semi-successful call). The call is not successful since the *errornum* corresponding to the jobfence access will be 0 (successful retrieval) and the *errornum* corresponding to the current job step access will be a 3 (insufficient capability).

Table 1. Item Descriptions

ITEM#	ITEM (information returned)	DATA TYPE
1	[JSNAME,]user.account (See note 1)	LA
2	session/job name (See note 2)	LA
3	user name (See note 2)	LA
4	user logon group (See note 2)	LA
5	user account (See note 2)	LA
6	user home group (See note 2)	LA
7	session/job introduction time (See note 3)	LA
8	session/job introduction date (See note 4)	LA
9	input ldev/class name (See note 2)	LA
10	output ldev/class name (See note 2)	LA
11	current job step (See note 5)	LA
12	current number of jobs	I
13	current number of sessions	I
14	job input priority	I
15	job/session number	D
16	jobfence	I
17	job output priority	I
18	number of copies	I
19	job limit (system)	I
20	session limit (system)	I
21	job deferred (See note 6)	I
22	main PIN - CI PIN for job/session	L
23	original job-spooled (See note 6)	L
24	RESTART option (See note 6)	L
25	sequenced - job (See note 6)	L
26	term code (See note 7)	L
27	CPU limit	L
28	session/job state (See note 8)	L
29	user's local attributes	D
30	\$\$STDIN spoolfile number (See notes 9 & 10)	I
31	\$\$STDIN spoolfile status (See notes 9 & 11)	I
32	\$\$STDLIST spoolfile number (See notes 9 & 10)	I
33	\$\$STDLIST spoolfile status (See notes 9 & 11)	I
34	length of current job step of item number 11	I
35	:SET \$\$STDLIST=DELETE invoked (See note 12)	L
36	Job Information Table data segment number	L

Table 1 (continued). Item Number Notes

1.	A maximum of 26 characters are allowed for input. Input must be in the form [jsname,]user.account. The wild card character (@) is not allowed.
2.	A maximum of 8 characters are returned.
3.	Returns a 32-bit double word in a form to be used by the FMTCLOCK intrinsic.
4.	Returns a 16-bit logical word in a form to be used by the FMTCALNDAR intrinsic.
5.	Returns a maximum of 283 characters, and is the image of the command currently executing.
6.	Returns the values: 0 = No. 1 = Yes.
7.	Returns the values: 0 = Regular terminal. 1 = Regular terminal with special log on. 2 = APL terminal. 3 = APL terminal.
8.	Returns the values: 2 = Executing. 4 = Suspending. 32 = Wait. 48 = Initialization.
9.	Returns data for current jobs and sessions. \$STDIN/\$STDLIST files only.
10.	Returns the spoolfile number as an integer.
11.	Returns the values: 0 = Active. 1 = Ready. 2 = Open. 3 = Locked.
12.	Returns the values: 0 = \$STDLIST will be saved. 1 = :SET \$STDLIST=DELETE is invoked.

All *itemnum* and *item* parameters are output parameters with one exception. Item number 1 can be used for an input and output parameter. Item number 1 is an input parameter only if the user is retrieving data by the Parse Method of Retrieval (Refer to the MPE V Intrinsics Reference Manual (32033-90007)). Otherwise, it is an output parameter. The number of characters returned is twenty-six or less. The returned message will be left justified and padded with blanks:

[*jsname*, *user.account*

A useful program would be to poll a user's :STREAM job to determine what job step it is currently executing. The following piece of pseudo code could be used to accomplish this:

# KILL

INTRINSIC NUMBER 102

Deletes a process.

```
IV  
KILL(pin);
```

A process can delete one of its sons by using the KILL intrinsic.

## PARAMETERS

*pin*                                *integer by value (required)*  
A word containing the Process Identification Number (PIN) of the process to be deleted. The value of *pin* must be an integer ranging from 1 to 255.

## CONDITION CODES

CCE                                Request granted.  
CCG                                Request granted. The specified process was terminating.  
CCL                                Request denied because an illegal PIN was specified.

## SPECIAL CONSIDERATIONS

Process Handling Capability required.

## TEXT DISCUSSION

Page 7-8.

Dynamically loads a library procedure.

INTRINSIC NUMBER 80

```
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.



# LOCKGLORIN

INTRINSIC NUMBER 34

Locks a global RIN.

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

Any global Resource Identification Number (RIN) assigned to a group of users can be locked, by one process at a time, by using the LOCKGLORIN intrinsic. When this is done, any other processes 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

<i>rinum</i>	<i>integer by value (required)</i> 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 <i>MPE Commands Reference Manual</i> for a description of the :GETRIN command.
<i>lockcond</i>	<i>logical (required)</i> 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.
<i>rinpassword</i>	<i>byte array (required)</i> 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 <i>lockcond</i> . If the RIN was free, TRUE is returned to <i>lockcond</i> .
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 Request denied because of invalid RIN. *Rinum* is not a global RIN or the value is out of bounds for the RIN table.

## 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.

# LOCKLOCRRIN

INTRINSIC NUMBER 32

Locks a local RIN.

IV L  
LOCKLOCRRIN(*rinum*,*lockcond*);

Any local Resource Identification Number (RIN) assigned to a job can be locked, by one process at a time, by using the LOCKLOCRRIN 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

- rinum*                                    *integer by value (required)*  
An identifier specifying one of the previously-allocated local RIN's, designated by an integer from 1 to the value specified in the *rincount* parameter of the GETLOCRRIN intrinsic.
- lockcond*                                *logical (required)*  
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 *lockcond*. If the RIN was free, FALSE is returned to *lockcond*.
- CCG                                        Not returned.
- 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.

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 *lockcond*. If the RIN was free, FALSE is returned to *lockcond*.

# 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.

# LOGSTATUS

INTRINSIC NUMBER 214

Provides information about an opened user logging file.

```
      D   LA   I  
LOGSTATUS (index,loginfo,status)
```

The LOGSTATUS intrinsic is used to obtain information about the opened logging file. Its primary use is to determine the amount of space used and remaining in a disc logging file.

## PARAMETERS

*index*                                    *double (required)*  
The parameter returned from OPENLOG that identifies your access to the logging system.

*loginfo*                                 *logical (required)*  
A formatted array in which the following information is returned:

- words 0 and 1 — total records written in log file
- words 2 and 3 — the size of the logging file
- words 4 and 5 — the space remaining in the log file
- word 6            — the number of users using the log system

*status*                                   *integer (required)*  
An integer in which error information is returned to the caller. Zero indicates OK status.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

None.

# LOCRINOWNER

INTRINSIC NUMBER 36

Determines PIN of process that has locked a local RIN.

```
I                                     IV  
pin := LOCRINOWNER (rinum);
```

After local RIN's have been acquired by a process, they can be locked and unlocked by other processes in the process structure. LOCRINOWNER determines the PIN (Process Identification Number) of the process that has a particular RIN locked.

## FUNCTIONAL RETURN

If the particular RIN is locked by the father process of the process which called LOCRINOWNER, a 0 is returned. Otherwise, the PIN of the son or brother process which has the local RIN locked is returned.

## PARAMETERS

*rinum*

*integer by value (required)*

The number of the local RIN (from 1 to the value specified in the *rincount* parameter of the GETLOCRIN intrinsic) for which the PIN of the locking process is to be determined.

## CONDITION CODES

CCE	Request granted.
CCG	Request denied. The local RIN specified by <i>rinum</i> is not currently locked by any process.
CCL	Request denied. The <i>rinum</i> parameter was invalid (was $\leq 0$ , greater than the RIN table size, or was greater than the number of local RIN's currently allocated to this process structure).

## TEXT DISCUSSION

Page 6-9

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

<i>pin</i>	<i>integer by value (required)</i> 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.
<i>count</i>	<i>integer (required)</i> 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.



# MYCOMMAND

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. Leading and trailing blanks are ignored. Lower case is upshifted.

In the byte array named for the *comimage* parameter, the first character of the 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. If the parameter is empty or all blanks, points to the delimiter.

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.

# MYCOMMAND

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.

# OPENLOG

INTRINSIC NUMBER 210

Provides access to the logging facility.

```
      D   LA  LA  I   I  
OPENLOG (index, logid, pass, mode, status);
```

The OPENLOG intrinsic provides access to the user logging facility.

## PARAMETERS

<i>index</i>	<i>double (required)</i> A double word returned to you which identifies logging access. The index is used to check the validity of subsequent calls to WRITELOG and CLOSELOG.
<i>logid</i>	<i>logical array (required)</i> An array of up to eight characters which supplies your logging identification. The array contains alphanumeric characters. Arrays of less than eight characters must end in a space.
<i>pass</i>	<i>logical array (required)</i> An array in which you assign a password associated with the logging identifier by the GETLOG command.
<i>mode</i>	<i>integer (required)</i> An integer which you use to indicate whether or not your process should be suspended if your request for service cannot be completed immediately. Enter a zero if you want to wait for service; enter a one if you do not want to wait.
<i>status</i>	<i>integer (required)</i> An integer which indicates logging system errors to you. (See table 10-12, User Logging Error Messages.)

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 10-93

Suspends the calling process for a specified number of seconds.

INTRINSIC NUMBER 45

**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	Request denied because of insufficient system table (Timer Request List) space.
CCL	Request denied because a negative value was specified for <i>interval</i> or the value is too large.

## TEXT DISCUSSION

Page 4-19.

# PCHECK

Returns an integer code specifying the completion status of the most recently executed DS/3000 program-to-program intrinsic.

```
      I          IV  
      icode:=PCHECK(dsnum);
```

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

# PCLOSE

Terminates program-to-program communication with a remote slave program.

```
IV  
PCLOSE(dsnum);
```

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

# PCONTROL

Exchanges tag fields with a remote slave program.

IV IA  
PCONTROL(*dsnum,itag*);

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.



# POPEN

Initiates program-to-program communication with a remote slave program.

```
      I      BA      BA IA      BA  IV  
dsnum:=POPEN(dsdevice,progrname,itag,entryname,param,  
             LV      IV  IV      IV  IV  
             flags,stacksize,dsize,maxdata,bufsize);
```

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

# PREAD

Requests a remote slave program to send a block of data.

```
I          IV  IA  IV  IA  
lgth:=PREAD(dsnum,target,tcount,itag);
```

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

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 only on terminals.

*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 10-45

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> The value 0 or %320.

## 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. The character string is limited to 50 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. This parameter should never specify a message length of more than 50 bytes.
<i>control</i>	<i>integer by value (required)</i> This parameter must be specified but is not used by MPE.
<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.



Copies, to a disc file, input from paper tapes which do not contain X-OFF control characters.

```

IV      IV
PTAPE(filenum1,filenum2);

```

When using terminals with attached tape readers (such as the ASR-33), 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 *aoption* 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

<i>filenum1</i>	<i>integer by value (required)</i> 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.
<i>filenum2</i>	<i>integer by value (required)</i> 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

# PUTJCW

Puts value of a particular job control word (JCW) in JCW table.

```
BA L I  
PUTJCW(jcwname,jcwvalue,status);
```

The PUTJCW intrinsic puts the value of a job control word (JCW) in the JCW table. If an entry of the same name already exists in the table, only its value is entered. If no entry exists for this name, an entry is created and its value is entered.

## PARAMETERS

<i>jcwname</i>	<i>byte array (required)</i> A byte array containing the name of the JCW. May contain up to 255 characters, beginning with a letter and ending with a non-alphanumeric character such as a blank. "@" causes all executing JCW's to be set to jcwvalue.
<i>jcwvalue</i>	<i>logical (required)</i> A word identifier containing the value of the JCW.
<i>status</i>	<i>integer (required)</i> A word identifier used by the system to return a value denoting the execution status of the intrinsic, as follows:  0 - Successful execution. Value entered in JCW. 1 - Error, <i>jcwname</i> greater than 255 characters long. 2 - Error, <i>jcwname</i> does not start with a letter. 3 - Error, JCW table overflow. No JCW with this name exists in table and unable to create new entry.

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-47

# PWRITE

Sends a block of data to the remote slave program.

```
IV IA IV IA  
PWRITE(dnum, target, tcount, itag);
```

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

# 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.

Affects the system job control word.

## 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
```

If *num* =0, PARAM=*num* is not printed.

### CONDITION CODES

The condition code remains unchanged.

### SPECIAL CONSIDERATIONS

Split stack calls permitted.

Affects the system job control word.

### TEXT DISCUSSION

Page 4-22

# 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

*message*

*logical array (required)*

The array into which the ASCII characters are read.

*expectedl*

*integer by value (required)*

An integer specifying the maximum length of the array *message*. If *expectedl* is positive, this specifies the length in words; if *expectedl* is negative, this specifies the length in bytes. When the record is read, the first *expectedl* characters are input. If *expectedl* 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 in 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)



# RECEIVEMAIL

## CONDITION CODES

- CCE Request granted. The mail was collected (the value 2 is returned to RECEIVEMAIL) or the mail was not collected because the mailbox contained outgoing mail from the receiving process. The value 1 is returned to RECEIVEMAIL.
- CCG Request denied because of an illegal *pin* parameter. The value 3 is returned to RECEIVEMAIL.
- CCL Request denied because the bounds check revealed that the *location* 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.

# REJECT

Rejects a request received by the preceding GET intrinsic call and returns an optional tag field back to a remote master program..

**IA**  
**REJECT(*itag*);**

See the DS/3000 Reference Manual (32190-90001) for a discussion of this intrinsic.

# RESETCONTROL

Resets terminal to accept CONTROL-Y signal.

INTRINSIC NUMBER 55

```
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.

# RESETDUMP

INTRINSIC NUMBER 79

Disables the abort stack analysis facility.

**RESETDUMP;**

## PARAMETERS

None

## CONDITION CODES

CCE	Request granted.
CCG	Abort stack analysis facility was disabled prior to the RESETDUMP call and remains disabled.
CCL	Not returned by this intrinsic.

## TEXT DISCUSSION

*MPE DEBUG/Stack Dump Reference Manual.*

Searches an array for a specified entry or name.

INTRINSIC NUMBER 70

```
      I      BA   IV BA 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.

# SENDMAIL

<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 (the value 0 or 1 is returned to SENDMAIL) or the mail was not sent because the mailbox contained incoming mail to be collected by the sending process. The value 2 is returned to SENDMAIL.
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.

# SETDUMP

INTRINSIC NUMBER 78

Enables the stack analysis facility.

```
LV  
SETDUMP(flags);
```

## PARAMETERS

*flags*

*logical by value (required)*

A logical word whose bits specify the following:

Bit 15 = If on, specifies a DL to Q initial dump.

Bit 14 = If on, specifies a Q initial to S dump.

Bit 13 = If on, specifies a Q-63 to S dump. This bit is ignored if bit 14 is on.

Bit 12 = If on, causes an ASCII dump of the octal content along with the octal values.

A 0 value for *flags* results in a display of registers and stack marker trace only.

## CONDITION CODES

CCE

Request granted.

CCG

Abort stack analysis facility already enabled before SETDUMP call. The facility is now set up according to new specifications from this call.

CCL

Not returned by this intrinsic.

## TEXT DISCUSSION

*MPE DEBUG/Stack Dump Reference Manual.*



Sets bits in the system job control word (JCW)

INTRINSIC NUMBER 72

```
LV
SETJCW(word);
```

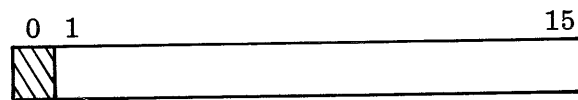
You can establish the bit contents of the system job control word (JCW) with the SETJCW intrinsic.

## 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:

PROGRAM TERMINATED IN ERROR STATE (CIERR 976)

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 (see the description on page 4-46).

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 4-46

# STACKDUMP

INTRINSIC NUMBER 77

Dumps selected parts of stack to a file.

```
          BA      I  L  DA  O-V  
STACKDUMP(filename, idnumber, flags, selec);
```

or (from SPL only)

```
          BA      I  L  DA  O-V  
STACKDUMP'(filename, idnumber, flags, selec);
```

## PARAMETERS

*filename*

*byte array (optional)*

Contains the file name of the file where the information is to be dumped. When *filename* contains the formal designator of the file, the file will be opened and closed by the STACKDUMP intrinsic. If the secondary entry point (STACKDUMP') is used to enter this intrinsic, MPE assumes that *filename*(0) contains the file number of a file which has been successfully opened prior to the call to STACKDUMP. In this case, the file is not closed before returning to the calling program. When a file number is passed via the STACKDUMP' secondary entry point, the record length must be between 32 and 256 words and write access must be allowed for the file.

*Default: Dump is sent to \$STDLIST.*

*idnumber*

*integer (optional)*

An integer which is displayed in the header of the dump to identify the printout.

*Default: Identifying integer not displayed.*

*flags*

*logical (optional)*

A logical value used to specify the following options:

Bit 15 = Suppress ASCII dump.

Bit 14 = Suppress trace back of stack markers.

*Default: If bits 14 and 15=00, a corresponding ASCII dump is provided for all values dumped in octal, and a trace back of stack markers is displayed.*

*selec*

*double array (optional)*

Specifies which stack areas are to be dumped. The format of the array is shown in the *MPE DEBUG/Stack Dump Reference Manual*. The array has no predetermined length; the first double word containing the values 0/-1 indicates the end of the array. An entry for which the count is 0 is interpreted as a "skip" (i.e., go to next double word element in list).

*Default: If missing, or if the first double word contains 0/-1 (indicating end of array), no dump occurs (except for the header), unless flags bit 14=0, in which case the trace back of stack markers is displayed.*

## CONDITION CODES

CCE	Request granted.
CCG	Request denied. Bounds violation occurred and the dump was not completed. Record size was not between 32 and 256 words.
CCL	Request denied. File system error occurred during opening, writing to, or closing the file. The file error number is returned in <i>idnumber</i> .

## TEXT DISCUSSION

*MPE DEBUG/Stack Dump Reference Manual.*

# SUSPEND

INTRINSIC NUMBER 103

Suspends a process.

```
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.

Switches DB register pointer.

INTRINSIC NUMBER 139

```
      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

INTRINSIC NUMBER 60

Terminates a process.

**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

Returns system timer.

INTRINSIC NUMBER 40

```
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 midnight preceding the last system cold load.

## CONDITION CODES

The condition code remains unchanged.

## SPECIAL CONSIDERATIONS

Split stack calls permitted.

## TEXT DISCUSSION

Page 4-42.





# UNLOCKGLORIN

Unlocks a global RIN.

INTRINSIC NUMBER 35

**IV**  
**UNLOCKGLORIN(*rinum*);**

The UNLOCKGLORIN intrinsic unlocks a global Resource Identification Number (RIN) that has been locked with the LOCKGLORIN intrinsic.

## PARAMETERS

*rinum*                                      *integer by value (required)*  
A word supplying the number of any RIN locked by the calling process. If *rinum* does not specify a RIN locked by the calling process, no action is taken.

## CONDITION CODES

CCE	Request granted.
CCG	Request denied because this RIN was not locked for this process.
CCL	Request denied because the specified RIN was not allocated.

## TEXT DISCUSSION

Page 6-3.

# UNLOCKLOCIN

INTRINSIC NUMBER 33

Unlocks a local RIN.

```
IV
UNLOCKLOCIN(rinum);
```

The UNLOCKLOCIN intrinsic unlocks a local Resource Identification Number (RIN) that has been locked by the LOCKLOCIN intrinsic.

## PARAMETERS

*rinum*                      *integer by value (required)*  
A word supplying the locked RIN, designated by an integer from 1 to the value specified in the *rincount* parameter of the GETLOCIN 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.

Returns user attributes.

INTRINSIC NUMBER 69

```

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).

Bit (13:1) = Communications System. (CS)

Bits (9:4) = Reserved for MPE. The WHO intrinsic sets these bits to zero.

Bit (8:1) = User Logging (LG)

Bit (7:1) = Volume set usage (UV).

Bit (6:1) = Volume set creation (CV).

# WHO

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.*

*lattr*

*double (optional)*

A double word to which is returned the local attributes of the user, as defined by a user with the Account Manager attribute.

*Default: The user's local attributes are not returned.*

*usern*

*byte array (optional)*

An 8-byte array to which the user's name is returned.

*Default: The user's name is not returned.*

*groupn*

*byte array (optional)*

An 8-byte array to which the name of the user's log-on group is returned.

*Default: The user's log-on group is not returned.*

*acctn*

*byte array (optional)*

An 8-byte array to which the name of the user's log-on account is returned.

*Default: The user's log-on account is not returned.*

*homen*

*byte array (optional)*

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.

*Default: This information is not returned.*

*termn*

*logical (optional)*

A word to which the logical device number of the job/session input device is returned.

*Default: The logical device number is not returned.*

**CONDITION CODES**

The condition code remains unchanged.

**TEXT DISCUSSION**

Page 4-10.

# WRITELOG

INTRINSIC NUMBER 211

Writes a record to a logging file.

```
      D L A I I I
WRITELOG (index, data, len, mode, status);
```

The WRITELOG intrinsic journalizes data base and subsystem file additions and modifications to logging tape or disc files.

## PARAMETERS

<i>index</i>	<i>double (required)</i> The parameter returned from OPENLOG that identifies your access to the logging file.
<i>data</i>	<i>logical array (required)</i> The array in which the information to be logged is passed. The maximum length of the data is 32K words. A log record contains 128 words of which 119 are available to you. If you specify a length greater than 119 words a continuation record is automatically posted for you. The most efficient use of the log file is a multiple of 119 words.
<i>len</i>	<i>integer (required)</i> The length of the data in <i>data</i> . A positive number indicates words, and a negative number indicates bytes. If the length is greater than 119 words, the information in <i>data</i> is divided into two or more physical log records.
<i>mode</i>	<i>integer (required)</i> An integer which you use to indicate whether or not your process should be suspended if your request for service cannot be completed immediately. Enter a zero if you want to wait for service, enter a one if you do not want to wait.
<i>status</i>	<i>integer (required)</i> An integer which indicates logging system errors to you. (See table 10-12, User Logging Error Messages.)

## CONDITION CODES

The condition code remains unchanged.

## TEXT DISCUSSION

Page 10-93

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 (required)*

A word in which the value of the previous *mask* is returned to the user's program.

# XARITRAP

*oldlabel*

*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

*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.

*oldplabel*                                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. Trap enabled.

CCG                                      Request granted. Trap disabled.

CCL                                      Request denied because of illegal *plabel*, 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



Enables or disables the system trap.

```
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  
actsize:=ZSIZE(size);
```

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

*size*                                      *integer by value (required)*  
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 *size* parameter, less than  $(S - DB) + 64$  words, was specified. This minimum value is assigned by default.

## TEXT DISCUSSION

Page 4-27.

# INTERPROCESS COMMUNICATION AND CIRCULAR FILES

SECTION

III

## INTRODUCTION

Interprocess communication (IPC) is a facility of the file system which permits multiple user processes to communicate with one another in an easy and efficient manner. To accomplish this, IPC uses message files as the interface between user processes. These message files act as first-in-first-out queues of records, with entries made by FWRITEs and deletions made by FREADs: one process may submit records to the file with the FWRITE intrinsic while another process takes records from the file using FREADs.

Occasionally a process may attempt to read a record from an empty message file, or write a record to a message file that is full. In such situations, the file system will usually cause the process to wait until its request can be serviced; that is, until another process either writes a record to the empty file or reads enough records to take a block from the full file.

There is a unidirectional flow of information between a given process and a message file: a process opening the file with read access, identified as a "reader", may only read from the file, and not write; a process opening the file with write access, identified as a "writer", may only write to the file, and not read. (If it is necessary for the same process to read and write, it may open the message file twice, once as a reader and once as a writer.) More than one message file may be associated with a process, and the process may be configured as a reader to some of the files and as a writer to others. A given message file typically has one reader, though more are allowed, and one or more writers.

Applications for IPC exist wherever it is necessary for processes to communicate with one another. In the case of a father process with several sons, message files may serve as interfaces between the processes: through one file, the father may direct the activities of the sons; through another, the sons may inform the father of their progress. Message files may also aid object managers during data base operations: several writers may send information to a file which serves as the single source from which the data base process actually receives the information.

## OPERATION

Message files are maintained and manipulated by several intrinsics. The FOPEN, FREAD, FWRITE, FCONTROL, and FCLOSE intrinsics operate upon the files to yield a unidirectional, first-in-first-out message queue.

### FOPEN

Establishes a connection to a message file. With FOPEN, a user process identifies itself as either a reader or writer; readers access the head of the message file and writers access the tail. Incompatible parameters that are specified with FOPEN are adjusted. For example, since messages are read or written to the file one record at a time, a multirecord parameter is corrected. If FOPEN is used to access a new file, a new message file is created.

## NOTE

There are different Access Type (bits (12:4) of the Access Options) specifications for slightly different types of writer processes. In one case, if a writer is the first accessor to a message file, the file's contents are purged; in another case, the writer simply appends records to the tail of the file. These AOPTIONS are discussed in a later section.

### FREAD

Reads one record from the head of a message file. The record is copied to the reader's TARGET area and is logically deleted from the message queue; the next record is now at the head of the file. If a process tries to read from an empty message file which writers are accessing, the file system causes it to wait until a writer process enters a record to the file; if there are no writers associated with the message file, an end-of-file indication, CCG, is returned.

## NOTE

If the message file is empty and there are no writers, the process will wait if there is an FCONTROL 45 in effect, or if this is the first FREAD after the reader's FOPEN.

### FWRITE

Appends one record to the tail of a message file. If a process tries to write to a full message file which readers are accessing, the file system causes it to wait until a reader process has read a block of records from the file; if there are no readers associated with the message file, an end-of-file indication, CCG, is returned.

## NOTE

If the message file is full and there are no readers, the process will wait if there is an FCONTROL 45 in effect, or if this is the first FWRITE after the writer's FOPEN.

### FCONTROL

Supplies various control functions during a process that is using a message file. These control functions permit a process to take advantage of the additional features of IPC, which are discussed in detail in the next section.

### FCLOSE

Breaks a process' connection with a message file. If the process reopens the same file later, it may do so as either a reader or a writer, regardless of what it was previously.

## ADDITIONAL FEATURES

Besides the regular attributes of IPC and message files, several features are available for use with these facilities. Writer ID's and nondestructive reads are specifically intended for use with IPC; copy access, the global multiaccess option and the ability to append to variable-length files are general enhancements to the file system. The time-out feature has been expanded to apply to IPC.

**Writer ID's.** When a writer process opens a message file, the file system assigns a unique 16-bit ID number to the writer. Each record the process writes to the message file is prefixed with this number by FWRITE. When the writer closes the file, the ID number is no longer associated with the process and may be reused. Whenever a writer opens or closes a message file, records are written to the file indicating these actions. Record prefixes and open/close records are usually transparent to the readers of the message file, but by issuing an FCONTROL 46, the reader process may see them. The interested reader may use the writer ID's to determine the source of the records it is receiving.

**Time-outs.** A reader or a writer process may limit the length of time it will wait to be serviced. By issuing an FCONTROL 4, a reader may specify the maximum number of seconds it will permit the file system to keep it waiting for a record to be written to an empty message file; a writer may also use FCONTROL 4 to specify the maximum number of seconds it will wait for a block of records to be read from a full file.

**Copy access.** When records are read from a message file, FREAD logically deletes them as it reads. In order to copy a message file without destroying it, the file must be opened with the file copy option specified in the AOPTIONS of the FOPEN, or the COPY keyword may be specified in a FILE equation. When this option is selected, the message file is treated as a standard sequential file rather than as a message queue, and may be copied safely. The file may then be read by logical record or by block, and information may be written to it by block.

#### NOTE

In order to access a message file in copy mode, a process must have exclusive access to the file.

**Nondestructive read.** By issuing an FCONTROL 47, a reader may avoid deleting the next record it reads; the record will remain at the head of the message queue. This feature differs from the copy access feature in that it is a temporary condition: the second FREAD following the FCONTROL 47 will reread the record and delete it in the usual manner.

**Global multiaccess.** When the global multiaccess option is requested, processes located in different jobs or sessions may open the same file. The global multiaccess option may be requested in the AOPTIONS of the FOPEN to the file, or by using the GMULTI keyword in a FILE command to create the file.

#### NOTE

Global multiaccess is unavailable to message files when they have been opened with exclusive access in copy mode.

**Appending to variable-length files.** Variable-length files may be opened with append access. It is not necessary to have fixed-length records of the maximum possible size, so space is conserved.

## USING IPC

Message files can be created in several ways. When a user process opens a new file and indicates in the FOPTIONS that it will be a message file, the FOPEN intrinsic creates the new message file. In order to create a message file with the BUILD command, use the MSG keyword; for example, to build a message file named SARA, enter:

```
:BUILD SARA; MSG
```

A new message file may also be specified with a FILE command. Use the MSG keyword for a new file:

```
:FILE LISBETH, NEW; MSG
```

A message file named LISBETH is indicated.

When you perform a LISTF,2 command, message files will be identified by an "M" in the third column of the TYP field; SARA is identified here:

FILENAME	CODE	-----LOGICAL RECORD-----			-----SPACE-----		
		SIZE	TYP	EOF	LIMIT	R/B	SECTORS #X MX
SARA		128W	VBM	0	1031	1	258 1 8

Other types of files are similarly indicated by a token in the TYP field:

- K — identifies a KSAM file
- R — identifies a Relative I/O file
- O — identifies a Circular file

A blank in the third column indicates a standard MPE file. Circular files are discussed in a later section.

Occasionally, you might create a message file and specify a certain number of records for the file to contain, only to discover that the file system has allocated more records for the file than you requested. The reason for this is that the file system is maintaining the necessary internal structure for the message file. The file system has four basic rules for establishing this structure when the message file is created:

- 1 Since records are written to the message file every time a writer process opens or closes the file, the file system adds two records to the requested number to allow for a minimum of one open and one close operation.
- 2 The requested number of records is rounded up to fill an even number of blocks.
- 3 The file system adds an extra block to the message file for the file label to occupy. (This block is transparent to you.)
- 4 Each extent is the same size; that is, the file system assigns the same number of blocks to each extent.

For example, suppose you want to create a message file named ODDSIZE:

```
:BUILD ODDSIZE; MSG; REC=,3; DISC=51,8
```

You have specified a message file with fifty-one records, three records per block, that occupies eight extents. The file system will adjust the number of records to conform to the rules for message file structure:

The file system adds two records to allow for one open and one close indication; the number of records goes from 51 to 53.

The number of records is rounded up to 54 to provide an even number of blocks. With three records per block, 54 records will fill 18 blocks.



An additional block is added to the file to accommodate the file label. Now the file contains 19 blocks.

The eight extents must all be the same size, so the number of blocks is increased from 19 to 24. Each extent now contains three blocks.

Of the 24 blocks in ODDSIZE, 23 are data blocks and one contains the file label, which is invisible to you. With three records per block, 23 blocks contain a total of 69 data records!

## FEATURES OF INTRINSICS FOR MESSAGE FILES

There are a few features of several intrinsics which apply specifically to message files. Most of these features are found in FOPEN and FCONTROL, but several other intrinsics are also affected.

Certain intrinsics are not allowed for message files.\* These intrinsics are listed in Table 3-1:

Table 3-1. Intrinsics that are not permitted with message files.

FPOINT	FREADDIR
FREADSEEK	FSPACE
FUPDATE	FWRITEDIR
FDELETE	

\*The FSETMODE intrinsic is permitted, but ignored.

Parameters that are omitted in the following descriptions retain their normal range of values and their normal default values.

### FOPEN

FOPTIONS: (2:3) — File type. Determines the type of file to create for a new file. If the file is old, this field is ignored.

- 000 — Ordinary file
- 001 — KSAM file
- 010 — Relative I/O file
- 100 — Circular file; discussed in the next section
- 110 — Message file

#### NOTE

The Default Designator FOPTION, bits 10 through 12, offers several choices for default file designators. Any value used other than 0 for “filename” will override the File Type field.

(8:2) — Record format. Message files are always internally formatted as variable-length records. However, a message file can appear as a fixed file to an opener. There is no difference for a writer, but a reader will have the portion of his target area which exceeds the file filled with blanks (for an ASCII file) or zeroes (for a binary file.)

- 00 — Fixed
- 01 — Variable
- 10 — Undefined, changed to variable

AOPTIONS:

- (3:1) — File copy. This feature permits a message file to be treated as a standard sequential file, so it can be copied by logical record or physical block to another file.
  - 0 — The file will be accessed in its native mode; that is, a message file will be treated as a message file.
  - 1 — The file is to be treated as a standard, sequential file with variable-length records. This allows nondestructive reading of an old message file at either the logical record or physical block level. Only block level access is permitted if the file is opened with write access. These blocks are checked for proper message file format to prevent incorrectly formatted data from being written to the message file while it is unprotected.

NOTE

In order to access a message file in copy mode, a process must have exclusive access to the file.

Setting this bit on causes all the remaining file parameters to have their normal defaults.

- (5:2) — Multiaccess mode. This feature permits processes located in different jobs or sessions to open the same file.
  - 00 — No multiaccess. The file system changes this value to 2 to allow global multiaccess.
  - 01 — Only intra-job multi-access allowed; this is the same as specifying the MULTI option in a FILE command.
  - 10 — Inter-job multi-access allowed; this is the same as specifying the GMULTI option in a FILE command.
  - 11 — Undefined. If this is specified, the FOPEN will be rejected with an error code of 40: ACCESS VIOLATION.
- (7:1) — Inhibit buffering. For message files, the file system sets this bit off.

NOTE

Readers may open a message file with NOBUF if they are in copy mode; this determines whether they will be accessing the file record by record or block by block:

- 0 — read by logical record
- 1 — read by physical block

Writers must open message files with NOBUF if they are in copy mode; they will access the file block by block.

- (8:2) — Exclusive. The values for this field are the same as for any disc file, but they have different meanings for the readers and writers of a message file:

USER VALUE	Means
EXCLUSIVE	One reader, one writer
SEMI	One reader, multiple writers
SHARE	Multiple readers and writers
Default	One reader, one writer

- (11:1) — Multirecord. For message files, the file system sets this bit to 0.

- (12:4) — Access type. These bits specify whether the user will be a reader or a writer process.

0000 —READ access only. The FWRITE intrinsic cannot reference this file. This access type requires both read and write access capability to the file. A process that has opened a file with this access type is a “reader”.

0001 —WRITE access only. If this is the first accessor to the file and the process has write access capability, then the file’s contents are purged. If this is not the first accessor to the file, the file system sets this access type to APPEND. The FREAD intrinsic cannot reference this file. A process that has opened a file with this access type is a “writer”.

0010 —WRITE SAVE access. The file system sets this to APPEND access.

0011 —APPEND access only. The FREAD intrinsic cannot reference this file. This access type requires append capability to the file. A process that has opened a file with this access type is a “writer”.

DEVICE: This field is relevant only if this is a new file. The DEVICE field must either be omitted or specify a disc; specification of any device other than a disc opens the device. When this happens, the file is no longer a message file.

NUMBUFFERS: (0:11) — Ignored.

(11:5) — Value between 2 and 31; default is 2. This parameter must not exceed the physical record capacity of the file.

FILESIZE: The number of records is rounded up to completely fill the last block and to make the last extent the same size as the other extents. Two additional records are included for the open and close records.

## FCONTROL

A few controlcodes deal specifically with IPC. Those not mentioned here are invalid when IPC is being used.

CONTROL-CODE	PARAM	DESCRIPTION
2	—	Complete all I/O; ignored in the case of message files.
3	—	Read hardware status word.
4	integer	Set time-out interval. This applies to both FREADs and FWRITES. The time-out will be armed at the beginning of the I/O request and cleared when the I/O completes. PARAM specifies the length of the time-out in seconds. A value of zero disables time-outs on the file.
6	—	Write end-of-file. Used only to verify the state of the file by writing out the file label and buffer area to disc; this ensures that the message file can survive system crashes. No EOF is written.
43	—	A CCG condition code is returned if an outstanding I/O operation has completed. An IOWAIT must be issued to finish the request.
45	TRUE	Enable extended wait. Permits a reader to wait on an empty file that is not currently opened by any writer, or a writer to wait on a full file that has no reader. This FCONTROL will remain in effect until FCONTROL 45 is issued with a PARAM value of FALSE.
	FALSE	Disable extended wait. Specifies that when an FREAD encounters an empty file that has no writer, or an FWRITE encounters a full file that has no reader, it will return an end-of-file condition. (Default)
46	TRUE	Enable reading the writer's ID. Each record read will have a two-word header. The first word will indicate the type of record: 0 - data record 1 - open record 2 - close record  The second word will contain the writer's ID number. If the record is a data record, the data will follow the header; open and close records contain no more information.
	FALSE	Disable reading the writer's ID. Only data is read to the reader's TARGET area. The open and close records are skipped and deleted by the file system when they come to the head of the message queue, and the two-word header is transparent to the reader. (Default)
47	TRUE	Nondestructive read. The next FREAD by this reader will not delete the record. Subsequent FREADs will be unaffected.
	FALSE	The next FREAD by this reader will delete the record. (Default)

## FCHECK

There is one error message that is returned only when using IPC:

```
151    CURRENT RECORD WAS LAST RECORD WRITTEN BEFORE SYSTEM CRASHED.
```

This message is returned when this record is read following system startup.

## FGETINFO

The value returned in RECSIZE will indicate the user's data record size, and the value returned in EOF will indicate the number of data records, unless an FCONTROL 46 is in effect. When an FCONTROL 46 is in effect, the value returned in RECSIZE will be the size of the user's data records, including the two-word header; the number of records returned in EOF will include open, close and data records.

The value returned in BLKSIZE reflects the actual blocksize of the file. When the file is created, the blocksize is computed by the following algorithm:

$$\text{BLOCKSIZE} := (( \text{RECORDSIZE} + 3 ) * \text{BLOCKING FACTOR} ) + 2$$

where RECORDSIZE and BLOCKSIZE are in words. For example, with a recordsize of 100 words and a blocking factor of 10, the blocksize would be 1032 words.

## FFILEINFO

Two values for ITEMVALUE are specifically for use with IPC:

ITEM #	TYPE	DESCRIPTION
34	integer	The current number of writers.
35	integer	The current number of readers.

## CIRCULAR FILES

Circular files are wrap-around structures which behave as standard sequential files until they are full. As records are written to a circular file, they are appended to the tail of the file; when the file is filled, the next record added causes the block at the head of the file to be deleted and all other blocks to be logically shifted toward the head of the file. Circular files may not be simultaneously accessed by both readers and writers. When the file has been closed by all writers, it may be read; a reader takes records from the circular file one at a time, starting at the head of the file.

Circular files are particularly useful as history files, when a user is interested in the information recently written to the file and is less concerned about earlier material that has been deleted. These history files are frequently used as debugging tools: diagnostic information may be written to the file, and the most recent and relevant material can be saved and studied.

Creating a circular file is similar to creating a message file. When a user process opens a new file and indicates in the AOPTIONS that it will be a circular file, the FOPEN intrinsic creates the new circular file. In order to create a circular file with the BUILD command, use the CIR keyword; for example, to build a circular file named CIRCLE, enter:

```
:BUILD CIRCLE; CIR
```

A new circular file may also be specified with a FILE command. Use the CIR keyword for a new file:

```
FILE ROUND, NEW; CIR
```

A circular file named ROUND is indicated.

When you perform a LISTF,2 command, circular files will be identified by an "0" in the TYP field; CIRCLE is identified here:

FILENAME	CODE	-----LOGICAL RECORD-----			-----SPACE-----		
		SIZE	TYP	EOF	LIMIT	R/B	SECTORS #X MX
CIRCLE		128W	FBO	0	1023	1	128 1 8

## FEATURES OF INTRINSICS FOR CIRCULAR FILES

Most intrinsics treat circular files the same way they treat regular disc files, but some have special features which apply specifically to circular files. Most of these features are found in FOPEN, but a few other intrinsics are also affected.

Certain intrinsics are not allowed when circular files are used. These intrinsics are listed in Table 3-2:

Table 3-2. Intrinsics that are not permitted with circular files.

Not permitted for READ access	Not permitted for WRITE access
FUPDATE	FUPDATE
FDELETE	FDELETE
FWRITEDIR	FWRITEDIR
FWRITE	FREAD
	FREADDIR
	FREADSEEK
	FPOINT
	FSPACE

Parameters that are omitted in the following descriptions retain their normal range of values and their normal default values.

### FOPEN

- FOPTIONS: (2:3) — File type. Determines the type of file to create. If the file is old, this field is ignored.
- 000 — Ordinary file
  - 001 — KSAM file
  - 010 — Relative I/O file
  - 100 — Circular file
  - 110 — Message file

- AOPTIONS: (5:2) — Multiaccess mode. This feature permits processes located in different jobs or sessions to open the same file.
- 00 — No multiaccess. For a writer, the file system changes this value to a 2 for global multiaccess.
  - 01 — Only intra-job multiaccess allowed; this is the same as specifying the MULTI option in a FILE command.
  - 10 — Inter-job multiaccess allowed; this is the same as specifying the GMULTI option in a FILE command.

11 — Undefined. If this is specified, the FOPEN will be rejected with an error code of 40: ACCESS VIOLATION.

(7:1) — Inhibit buffering. Reader processes may open circular file with either the BUF or NOBUF option; for write access to circular files, the file system sets this bit off.

NOTE

Readers may open a circular file with NOBUF if they are in copy mode; this determines whether they will be reading the file record by record or block by block:

- 0 — read by logical record
- 1 — read by physical block

(8:2) — Exclusive. The values for this field are the same as for any standard disc file, but they have different meanings for the readers and writers of a circular file:

USER VALUE	Changed to:	
	READER	WRITER
EXCLUSIVE	EXCLUSIVE	EXCLUSIVE
SEMI	SHARE	EXCLUSIVE
SHARE	SHARE	SHARE
Default	SHARE	EXCLUSIVE

For readers, SHARE means “allow other readers”;  
for writers, SHARE means “allow other writers”.

(11:1) — Multirecord. When a reader specifies this option, the file will be accessed NOBUF; for writers, this bit is set to zero.

(12:4) — Access type. These bits specify whether the user will be a reader or a writer process.

0000 — READ access only.

0001 — WRITE access only. If this is the first accessor to the file, then the file’s contents are purged. If this is not the first accessor to the file, the access type is set to APPEND.

0010 — WRITE SAVE access. Set to APPEND access.

0011 — APPEND access only.

NOTE

Circular files allow variable length records with append access.

Any other access types are invalid.

**FILESIZE:** The number of records is rounded up to completely fill the last block.

## **FWRITE**

This intrinsic logically appends the user's record to the end of the file. If the file is full, the first block is deleted, the remaining blocks are logically shifted to the file's head, and the new record is appended to the end of the file.

## **FCLOSE**

For circular files, deletion of disc space beyond the end-of-file is not allowed.

## **EXAMPLES**

The following programs illustrate the use of IPC via message files. Intrinsic called within the programs manipulate the message files to produce a unidirectional flow of information.

In these two programs, the first is sending information to the second through a message file: the first program, PROC1, reads data from a data file and writes it to MSGFILE2; the second program, PROC2, can then read this data from MSGFILE2 and print it. When PROC2 finishes reading and printing the data, it writes a message to MSGFILE1 indicating this and terminates. PROC1 reads this message from MSGFILE1 and also terminates. The messages travel among processes and message files as illustrated in Figure 3-1:

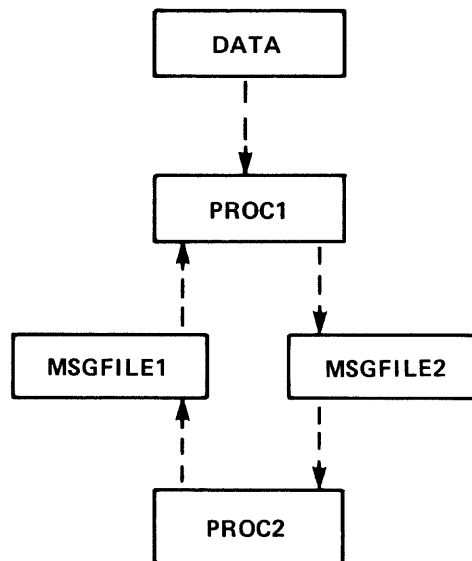


Figure 3-1. Data paths among processes and message files.



\$CONTROL USLINIT

<< Purpose: >>

<< Read data from a data file and send to another process. >>

BEGIN

```
LOGICAL EOF := FALSE;
INTEGER DATA'FILE, LEN, PIN, IN'FILE, OUT'FILE;
BYTE ARRAY IN'FILE'NAME (0:8) := "MSGFILE1 ";
BYTE ARRAY OUT'FILE'NAME (0:8) := "MSGFILE2 ";
BYTE ARRAY DATA'FILE'NAME (0:8) := "DATA ";
BYTE ARRAY PRINTPROC (0:8) := "PRNTPROC ";
ARRAY MESSAGE (0:39);
```

```
INTRINSIC CREATEPROCESS, FCLOSE, FOPEN, FREAD, FWRITE,
QUITPROG, PRINT, READ;
```

<< Create entries for the message files in the directory: >>

<< Note that IN'FILE'NAME ("MSGFILE1") is opened with FOPTIONs >>  
<< %30004: this indicates a new ASCII message file. >>

```
IN'FILE := FOPEN (IN'FILE'NAME, %30004);
IF < THEN QUITPROG (1);
FCLOSE (IN'FILE, 2, 0);      << Save file as session temporary. >>
IF < THEN QUITPROG (2);
```

<< Note that OUT'FILE'NAME ("MSGFILE2") is opened with FOPTIONs >>  
<< %30004: this indicates a new ASCII message file. >>

```
OUT'FILE := FOPEN (OUT'FILE'NAME, %30004);
IF < THEN QUITPROG (3);
FCLOSE (OUT'FILE, 2, 0);    << Save file as session temporary. >>
IF < THEN QUITPROG (4);
```

<< Create and activate the print process: >>

```
CREATEPROCESS (, PIN, PRINT'PROC)
IF < THEN QUITPROG (5);
```

<< Open message file for traffic from print process: >>

```
<< Note that IN'FILE'NAME ("MSGFILE1") is opened with FOPTIONs >>
<< %106 and AOPTIONS %1100: %106 indicates an old temporary >>
<< ASCII file and %1100 indicates a reader process with >>
<< exclusive access and multiaccess capability. MSGFILE1 >>
<< has already been designated as a message file. Since >>
<< only one reader and one writer process will be accessing >>
<< the message file, exclusive access mode is specified. >>
```

```
IN'FILE := FOPEN (IN'FILE'NAME, %106, %1100);
IF < THEN QUITPROG (7);
```

<< Open message file for traffic to print process: >>

```

<< Note that OUT'FILE'NAME ("MSGFILE2") is opened with FOPTIONs >>
<< %106 and AOPTIONS %1101: %106 indicates an old temporary >>
<< ASCII file and %1101 indicates a writer process with >>
<< exclusive access and multiaccess capability. MSGFILE2 has >>
<< already been designated as a message file. Since only >>
<< one reader and one writer process will be accessing the >>
<< message file, exclusive access mode is specified. >>

OUT'FILE := FOPEN (OUT'FILE'NAME, %106, %1101);
IF < THEN QUITPROG (8);

<< Open data input file: >>

<< Note that DATA'FILE'NAME ("DATA") is opened with FOPTIONs %3 >>
<< and AOPTIONS 0: %3 indicates an old permanent or temporary >>
<< file and 0 indicates read only access. The file system >>
<< will change the FOPTIONs to specify an ASCII file. >>

DATA'FILE := FOPEN (DATA'FILE'NAME, %3, 0);
IF <> THEN QUITPROG (9);

WHILE NOT EOF DO BEGIN
  LEN := FREAD (DATA'FILE, MESSAGE, -80);
  IF < THEN QUITPROG (10);
  IF > THEN EOF := TRUE
  ELSE BEGIN
    FWRITE (OUT'FILE, MESSAGE, -LEN, 0);
    IF <> THEN QUITPROG (11);
  END;
END << WHILE >>;

FCLOSE (OUT'FILE, 4, 0); << No more data to send: EOF >>
IF < THEN QUITPROG (12);

FREAD (IN'FILE, MESSAGE, 1); << Wait for printing process >>
IF <> THEN QUITPROG (13); << to finish. >>

FCLOSE (IN'FILE, 4, 0);
IF < THEN QUITPROG (14);
END.

```

\$CONTROL USLINIT

<< Purpose: >>

<< Receive data from other process and print it. >>

BEGIN

LOGICAL EOF := FALSE;  
INTEGER LEN, IN'FILE, OUT'FILE;

BYTE ARRAY IN'FILE'NAME (0:8) := "MSGFILE2 " ;  
BYTE ARRAY OUT'FILE'NAME (0:8) := "MSGFILE1 " ;  
ARRAY MESSAGE (0:39);

INTRINSIC FCLOSE, FOPEN, FREAD, FWRITE, QUITPROG, PRINT;

<< Open message file for traffic from other process: >>

<< Note that IN'FILE'NAME ("MSGFILE2") is opened with FOPTIONs >>  
<< %106 and AOPTIONS %1100: %106 indicates an old temporary >>  
<< ASCII file and %1100 indicates a reader process with >>  
<< exclusive access and multiaccess capability. MSGFILE2 >>  
<< has already been designated as a message file. Since >>  
<< only one reader and one writer process will be accessing >>  
<< the message file, exclusive access mode is specified. >>

IN'FILE := FOPEN (IN'FILE'NAME, %106, %1100);  
IF < THEN QUITPROG (13);

<< Open message file for traffic to other process: >>

<< Note that OUT'FILE'NAME ("MSGFILE1") is opened with FOPTIONs >>  
<< %106 and AOPTIONS %1101: %106 indicates an old temporary >>  
<< ASCII file and %1101 indicates a writer process with >>  
<< exclusive access and multiaccess capability. MSGFILE1 >>  
<< has already been designated as a message file. Since only >>  
<< one reader and one writer process will be accessing the >>  
<< message file, exclusive access mode is specified. >>

OUT'FILE := FOPEN (OUT'FILE'NAME, %106, %1101);  
IF < THEN QUITPROG (14);

WHILE NOT EOF DO BEGIN  
LEN := FREAD (IN'FILE, MESSAGE, -80);  
IF < THEN QUITPROG (15);  
IF > THEN EOF := TRUE  
ELSE PRINT (MESSAGE, -LEN, 0);  
END << WHILE >>;

<< Now signal other process; we are done. >>

FCLOSE (OUT'FILE, 4, 0);  
IF < THEN QUITPROG (16);

FCLOSE (IN'FILE, 4, 0);  
IF < THEN QUITPROG (17);

END.

These two COBOL programs perform the same tasks as the preceding SPL programs: the first program, FATHERPROC, reads data from a data file and writes it to MSGFILE2; the second program, SONPROC, can then read this data from MSGFILE2 and print it. When SONPROC finishes reading and printing the data, it writes a message to MSGFILE1 indicating this and terminates. FATHERPROC reads this message from MSGFILE1 and also terminates. The messages travel among processes and message files as illustrated in Figure 3-2:

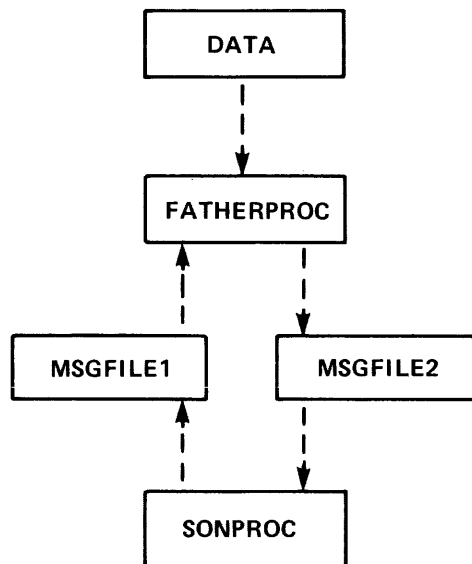


Figure 3-2. Data paths among processes and message files.

```

$CONTROL USLINIT
IDENTIFICATION DIVISION.
PROGRAM-ID. FATHERPROC.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP3000.
OBJECT-COMPUTER. HP3000.
SPECIAL-NAMES.
CONDITION-CODE IS CC.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATA-FILE      PIC S9(4) COMP.
01 LEN            PIC S9(4) COMP.
01 PIN           PIC S9(4) COMP.
01 IN-FILE       PIC S9(4) COMP.
01 OUT-FILE      PIC S9(4) COMP.
01 IN-FILE-NAME  PIC X(9) VALUE "MSGFILE1 ".
01 OUT-FILE-NAME PIC X(9) VALUE "MSGFILE2 ".
01 DATA-FILE-NAME PIC X(5) VALUE "DATA ".
01 PRINTPROC     PIC X(9) VALUE "PRNTPROC ".
01 MESSAGE-BUF   PIC X(80).
01 EOF-VAR       PIC X.
08 EOF           VALUE "E".
  
```

\* ERROR VARIABLES

```
01 ERROR-BUFFER.  
    05 FILLER          PIC X OCCURS 1 TO 80 TIMES  
                        DEPENDING ON LEN.  
01 ERR-NUM            PIC S9(4) COMP.  
01 FILE-NUM          PIC S9(4) COMP.  
01 QUIT-PARM         PIC S9(4) COMP.  
PROCEDURE DIVISION.  
MAIN PROCESSING SECTION.
```

```
$DEFINE %QUITPROG=                                QUITPROG  
    MOVE !1 TO QUIT-PARM                          QUITPROG  
    MOVE !2 TO FILE-NUM                          QUITPROG  
    PERFORM PRINT-ERROR#                          QUITPROG  
DRIVER-PARA.  
    PERFORM INIT-PARA.  
    MOVE "F" TO EOF-VAR.  
    PERFORM LOAD-PARA UNTIL EOF.  
    PERFORM CLOSE-PARA.  
    STOP RUN.
```

\*  
\* Create entries for the message files in the directory.  
\*  
\* Note that IN-FILE-NAME ("MSGFILE1") is opened with FOPTIONs  
\* %30004: this indicates a new ASCII message file.  
\*

```
INIT-PARA.  
    CALL INTRINSIC "FOPEN"  
        USING IN-FILE-NAME %30004  
        GIVING IN-FILE.  
    IF CC NOT = 0  
        %QUITPROG(1#,IN-FILE#).  
    CALL INTRINSIC "FCLOSE" USING IN-FILE %2 %0.  
    IF CC NOT = 0  
        %QUITPROG(2#,IN-FILE#).
```

\*  
\* Note that OUT-FILE-NAME ("MSGFILE2") is opened with FOPTIONs  
\* %30004: this indicates a new ASCII message file.  
\*

```
    CALL INTRINSIC "FOPEN"  
        USING OUT-FILE-NAME %30004  
        GIVING OUT-FILE.  
    IF CC NOT = 0  
        %QUITPROG(3#,OUT-FILE#).  
    CALL INTRINSIC "FCLOSE" USING OUT-FILE %2 %0.  
    IF CC NOT = 0  
        %QUITPROG(4#,OUT-FILE#).
```

\*  
\* Create and activate the print process.  
\*  
 CALL INTRINSIC "CREATEPROCESS" USING \ PIN PRINTPROC.  
 IF CC NOT = 0  
 %QUITPROG(5#,-1#).  
\*

\* Open message file for traffic from print process.  
 \*  
 \* Note that IN-FILE-NAME ("MSGFILE1") is opened with FOPTIONs  
 \* %106 and AOPTIONs %1100: %106 indicates an old temporary  
 \* ASCII file and %1100 indicates a reader process with exclu-  
 \* sive access and multiaccess capability. MSGFILE1 has already  
 \* been designated as a message file. Since only one reader and  
 \* one writer process will be accessing the message file,  
 \* exclusive access mode is specified.  
 \*

```
CALL INTRINSIC "FOPEN"
      USING IN-FILE-NAME %106 %1100
      GIVING IN-FILE.
IF CC NOT = 0
      %QUITPROG (7#,IN-FILE#).
```

\*  
 \* Open message file for traffic to print process.  
 \*  
 \* Note that OUT-FILE-NAME ("MSGFILE2") is opened with FOPTIONs  
 \* %106 and AOPTIONs %1101: %106 indicates an old temporary  
 \* ASCII file and %1101 indicates a writer process with exclu-  
 \* sive access and multiaccess capability. MSGFILE2 has already  
 \* been designated as a message file. Since only one reader and  
 \* one writer process will be accessing the message file,  
 \* exclusive access mode is specified.  
 \*

```
CALL INTRINSIC "FOPEN"
      USING OUT-FILE-NAME %106 %1101
      GIVING OUT-FILE.
IF CC NOT = 0
      %QUITPROG(8#,OUT-FILE#).
```

\*  
 \* Open data input file.  
 \*  
 \* Note that DATA-FILE-NAME ("DATA") is opened with FOPTIONs %3  
 \* and AOPTIONs 0: %3 indicates an old permanent or temporary  
 \* file and 0 indicates read only access. The file system will  
 \* change the FOPTIONs to specify an ASCII file.  
 \*

```
CALL INTRINSIC "FOPEN"
      USING DATA-FILE-NAME %3 %0
      GIVING DATA-FILE.
IF CC NOT = 0
      %QUITPROG(9#,DATA-FILE#).
```

\*  
 \* Load input to message file.  
 \*

```

LOAD-PARA.
  CALL INTRINSIC "FREAD"
    USING DATA-FILE MESSAGE-BUF -80
    GIVING LEN.
  IF CC NOT = 0
    IF CC LESS THAN 0 THEN
      %QUITPROG(10#,DATA-FILE#)
    ELSE
      MOVE "E" TO EOF-VAR
  ELSE
    COMPUTE LEN = - LEN
    CALL INTRINSIC "FWRITE"
      USING OUT-FILE MESSAGE-BUF LEN %0
    IF CC NOT = 0
      %QUITPROG(11#,OUT-FILE#).

CLOSE-PARA.
  CALL INTRINSIC "FCLOSE" USING OUT-FILE %4 %0.
  IF CC NOT = 0
    %QUITPROG(12#,OUT-FILE#).

*
* Wait for print to finish.
*
  CALL INTRINSIC "FREAD" USING IN-FILE MESSAGE-BUF %1.
  IF CC < 0
    %QUITPROG(13#,IN-FILE#).
  CALL INTRINSIC "FCLOSE" USING IN-FILE %4 %0.
  IF CC NOT = 0
    %QUITPROG(14#,IN-FILE#).

*
* General error routine.
*
PRINT-ERROR SECTION.
WHAT-TYPE.

  IF FILE-NUM IS NOT NEGATIVE THEN
    CALL INTRINSIC "FCHECK" USING FILE-NUM ERR-NUM
    MOVE 80 TO LEN
    CALL INTRINSIC "FERRMSG" USING ERR-NUM ERROR-BUFFER LEN
    DISPLAY ERROR-BUFFER.
  IF QUIT-PARM IS NOT NEGATIVE THEN
    CALL INTRINSIC "QUITPROG" USING QUIT-PARM.

$CONTROL USLINIT
IDENTIFICATION DIVISION.
PROGRAM-ID. SONPROC.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. HP3000.
OBJECT-COMPUTER. HP3000.
SPECIAL-NAMES.
CONDITION-CODE IS CC.
DATA DIVISION.

```

```

WORKING-STORAGE SECTION.
01 LEN          PIC S9(4) COMP.
01 IN-FILE      PIC S9(4) COMP.
01 OUT-FILE     PIC S9(4) COMP.
01 IN-FILE-NAME PIC X(9) VALUE "MSGFILE2 ".
01 OUT-FILE-NAME PIC X(9) VALUE "MSGFILE1 ".
01 MESSAGE-BUF  PIC X(80).
01 EOF-VAR      PIC X.
      88 EOF          VALUE "E".

* Error variables.
01 ERROR-BUFFER.
      05 FILLER      PIC X OCCURS 1 TO 80 TIMES
                          DEPENDING ON LEN.

01 ERR-NUM      PIC S9(4) COMP.
01 FILE-NUM     PIC S9(4) COMP.
01 QUIT-PARM    PIC S9(4) COMP.
PROCEDURE DIVISION.
MAIN-PROCESSING SECTION.
$DEFINE %QUITPROG=
      MOVE !1 TO QUIT-PARM          QUITPROG
      MOVE !2 TO FILE-NUM          QUITPROG
      PERFORM PRINT-ERROR#        QUITPROG
DRIVER-PARA.
      PERFORM OPEN-PARA.
      MOVE "F" TO EOF-VAR.
      PERFORM READ-PARA UNTIL EOF.
      PERFORM CLOSE-PARA.
      STOP RUN.

*
* Open message file for traffic from other process.
*
* Note that IN-FILE-NAME ("MSGFILE2") is opened with FOPTIONs
* %106 and AOPTIONs %1100: %106 indicates an old temporary
* ASCII file and %1100 indicates a reader process with
* exclusive access and multiaccess capability. MSGFILE2 has
* already been designated as a message file. Since only one
* reader and one writer process will be accessing the message
* file, exclusive access mode is specified.
*
OPEN-PARA.
      CALL INTRINSIC "FOPEN"
          USING IN-FILE-NAME %106 %1100
          GIVING IN-FILE.
      IF CC NOT = 0
          %QUITPROG(15#,IN-FILE#).

*
* Open message file for traffic to other process.
*
* Note that OUT-FILE-NAME ("MSGFILE1") is opened with FOPTIONs
* %106 and AOPTIONs %1101: %106 indicates an old temporary
* ASCII file and %1101 indicates a writer process with exclu-
* sive access and multiaccess capability. MSGFILE1 has already

```



```

* been designated as a message file. Since only one reader and
* one writer process will be accessing the message file,
* exclusive access mode is specified.
*

```

```

    CALL INTRINSIC "FOPEN"
        USING OUT-FILE-NAME %106 %1101
        GIVING OUT-FILE.
    IF CC NOT = 0
        %QUITPROG(16#,OUT-FILE#).

```

```

*
* Read messages from message file.
*

```

```

READ-PARA.
    CALL INTRINSIC "FREAD"
        USING IN-FILE MESSAGE-BUF -80
        GIVING LEN.
    IF CC NOT = 0
        IF CC LESS THAN 0 THEN
            %QUITPROG(17#,IN-FILE#)
        ELSE
            MOVE "E" TO EOF-VAR

```

```

*
* Print message out.
*

```

```

    ELSE
        COMPUTE LEN = - LEN
        CALL INTRINSIC "PRINT"
            USING MESSAGE-BUF LEN %0
        IF CC NOT = 0
            %QUITPROG(18#,2#).

```

```

*
* Now signal the other process; we are done.
*

```

```

CLOSE-PARA.
    CALL INTRINSIC "FCLOSE" USING OUT-FILE %4 %0.
    IF CC NOT = 0
        %QUITPROG(19#,OUT-FILE#).
    CALL INTRINSIC "FCLOSE" USING IN-FILE %4 %0.
    IF CC NOT = 0
        %QUITPROG(20#,IN-FILE#).

```

```

*
* General error routine.
*

```

```

PRINT-ERROR SECTION.
WHAT-TYPE.

```

```

    IF FILE-NUM IS NOT NEGATIVE THEN
        CALL INTRINSIC "FCHECK" USING FILE-NUM ERR-NUM
        MOVE 80 TO LEN
        CALL INTRINSIC "FERRMSG" USING ERR-NUM ERROR-BUFFER LEN
        DISPLAY ERROR-BUFFER.
    IF QUIT-PARM IS NOT NEGATIVE THEN
        CALL INTRINSIC "QUITPROG" USING QUIT-PARM.

```

# 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, from ASCII to EBCDIC, from EBCDIC to JIK (Katakana), and from JIK 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 4-18.
- 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.
- Format the calendar date with `FMTCALENDAR`. See page 4-45.
- Format the time of day with `FMTCLOCK`. See page 4-45.
- Format the calendar date and time of day with `FMTDATE`. See page 4-45.
- 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 4-34.
- 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 words with `SETJCW` (see page 4-46), `GETJCW` (see page 4-46), `PUTJCW` (see page 4-47), and `FINDJCW` (see page 4-47).
- Access a message catalog in the MPE message system, and insert parameters in a message, with the `GENMESSAGE` intrinsic. See page 4-50.
- Control function of 2680 A page printer.

## 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

<i>procname</i>	Contained in the byte array PNAME. The contents of PNAME are "PROC1". Note that the last character is a blank.
<i>lib</i>	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 <i>lib</i> parameter would cause the search to be conducted in this order: Account Public Library System Library
<i>plabel</i>	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.

Example 2:

```
BYTE ARRAY SHORTCOMMANDS (0:29):=
```

Entry	Definition
6,1,"I",	"IN^ ",
7,1,"O",	"OUT^ ",
8,1,"S",	"SKIP^ ",
8,1,"E",	"EXIT^ ",
0;	

NOTE: This would enable the program to allow abbreviations and replace them with the full command.

Example 3:

```
BYTE ARRAY RESPONSETABLE (0:9):=
```

Entry
5,3,"YES",
4,2,"NO",
0;

NOTE: In this example, the definition portion was not deemed necessary by the main program.

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 in Example 1 for the string "IN", the following intrinsic call could be used.

```
BYTE ARRAY COMMAND (0:3);MOVE COMMAND;="IN";  
ENUM:=SEARCH (COMMAND,2,COMMANDTABLE,DEFADDR);
```

The length of the string in COMMAND 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 "IN" will be returned to the word ENUM.

## 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.
- A zero, as the length of the last entry, indicating the end of the dictionary.

The entry number of the first entry in such a dictionary is *one*. If the entry is not found, it is indicated by a *zero*.

In the following examples, consider a byte array wherein the relationship of each entry to its definition is:

Example 1:

```
BYTE ARRAY COMMANDTABLE(0:25):=
```

Entry	Definition
5,2,“IN”,	1,
6,3,“OUT”,	1,
7,4,“SKIP”,	2,
7,4,“EXIT”,	0,
0;	

NOTE: The main program presumably will use the definitions to mean:

0 = No parameter follows  
1 = Filename parameter follows  
2 = Numeric parameter follows

## 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, REG:="? ",
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(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 4-1. 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.

:ABORTIO	:COMMENT	:GIVE
:ABORTJOB	:CONSOLE	:HEADOFF
:ACCEPT	:DEALLOCATE	:HEADON
:ALLOW	:DELETESPOOLFILE	:HELP
:ALTACCT	:DISALLOW	:IMLCONTROL
:ALTGROUP	:DISASSOCIATE	:JOBFENCE
:ALTJOB	:DOWN	:JOBPRI
:ALTLOG	:DOWNLOAD	:JOBSECURITY
:ALTSEC	:DSCONTROL	:LDISMOUNT
:ALTSPOOLFILE	:DSLIN	:LIMIT
:ALTUSER	:DSTAT	:LISTACCT
:ALTVSET	:FILE	:LISTF
:ASSOCIATE	:FOREIGN	:LISTGROUP
:BREAKJOB	:GETLOG	:LISTLOG
:BUILD	:GETRIN	:LISTUSER

:LISTVS	:RENAME	:SHOWME
:LMOUNT	:REPLY	:SHOWOUT
:LOG	:REPORT	:SHOWQ
:MPLINE	:RESET	:SHOWTIME
:MRJECONTROL	:RESETACCT	:SPEED
:NEWACCT	:RESETDUMP	:STARTSPOOL
:NEWGROUP	:RESTORE	:STOPSPPOOL
:NEWUSER	:RESUMEJOB	:STORE
:NEWVSET	:RESUMELOG	:STREAM
:OUTFENCE	:RESUMESPOOL	:STREAMS
:PTAPE	:SAVE	:SUSPENDSPOOL
:PURGE	:SECURE	:SWITCHLOG
:PURGEACCT	:SETDUMP	:TAKE
:PURGEGROUP	:SETJCW	:TELL
:PURGEUSER	:SETMSG	:TELLOP
:PURGEVSET	:SHOWALLOW	:TUNE
:RECALL	:SHOWCOM	:UP
:REFUSE	:SHOWDEV	:VMOUNT
:RELEASE	:SHOWIN	:WARN
:RELLOG	:SHOWJCW	:WELCOME
:REMOTE	:SHOWJOB	
:REMOTE HELLO	:SHOWLOG	

See the *MPE Commands Reference Manual* and the *Console Operator's Guide* 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

Conditions which result in warning messages are not returned to the user.

## 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);
```

```

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, REG:="?",
00014000 00016 1  LGTH, INDX, PARMCNT, TYPE;
00015000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00016000 00016 1
00017000 00016 1  <<END OF DECLARATIONS>>
00019000 00016 1
00020000 00016 1  PRINT(HEADING,9,0); <<PROGRAM ID>>
00021000 00016 1  WHO(INTERACTIVE); <<LIVE USER?>>
00022000 00004 1
00023000 00010 1  LOOP: <<PROMPT USER?>>
00024000 00010 1  IF INTERACTIVE THEN PRINT(REG,1,%320); <<GET COMMAND>>
00025000 00016 1  LGTH:=READ(INPUT,-72); <<CHECK FOR ERROR>>
00026000 00023 1  IF <> THEN QUIT(1); <<DONE - EXIT>>
00027000 00026 1  IF COMMAND="END" THEN GO EXIT; <<CARRIAGE RETURN>>
00028000 00040 1  COMMAND(LGTH):=%35;
00028100 00043 1  TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART COMMAND>>
00029000 00043 1  PARMINFO,TABLE);
00030000 00050 1  IF < THEN GO ERROR; <<NO COMMAND MATCH>>
00031000 00056 1  IF PARMCNT<>1 THEN GO ERROR; <<NO PARAMETERS>>
00032000 00057 1  INDX:=PARMINFO=@COMMAND; <<SUBSCRIPT OF PARM>>
00033000 00062 1  OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARAMETER>>
00034000 00065 1  PARMINFO(1).(0:8));
00035000 00070 1  IF <> THEN GO ERROR; <<CHECK FOR ERROR>>
00036000 00075 1
00036100 00076 1  CASE (TYPE-1) OF <<SELECT OPERATION>>
00037000 00076 1  BEGIN
00038000 00100 1  ACCUM:=ACCUM+OPERAND; <<ADD COMMAND>>
00039000 00106 2  ACCUM:=ACCUM-OPERAND; <<SUB COMMAND>>
00040000 00116 2  ACCUM:=ACCUM*OPERAND; <<MUL COMMAND>>
00041000 00122 2  ACCUM:=ACCUM/OPERAND; <<DIV COMMAND>>
00042000 00126 2  ACCUM:=OPERAND; <<SET COMMAND>>
00043000 00133 2  END;
00044000 00136 2  RESULT:
00045000 00143 1  MOVE ANSWER(8):=" "; <<RESET OLD ANSWER>>
00046000 00143 1  ASCII(ACCUM,10,ANSWER(8)); <<CONVERT ACCUM>>
00047000 00155 1  PRINT(OUTPUT,7,0); <<OUTPUT NEW ANSWER>>
00048000 00163 1  GO LOOP; <<CONTINUE CALCULATION>>
00049000 00170 1  ERROR:
00050000 00171 1  PRINT(ERRMSG,7,0); <<ERROR MESSAGE>>
00051000 00171 1  IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVE USER-QUIT>>
00052000 00175 1  GO LOOP; <<CONTINUE CALCULATION>>
00053000 00201 1  EXIT: END.
00054000 00202 1  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

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 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.

```

00001000 00000 0 $CONTROL 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, REG:="? ",
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(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 4-3. Using the ASCII Intrinsic

```

00001000 00000 0  $CONTROL USLIMIT
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 BMSG(*)=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,BMSG(20)); <<CONVERT COUNTER>>
00016000 00007 2  PPRINT(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(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 4-4. Using the DASCII Intrinsic

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 WITH THE CTRANSLATE INTRINSIC

The CTRANSLATE intrinsic is used for character code translating, whether between the standard computer character codes or with a user-defined code. It permits you to obtain character code conversions within programs of your own design. In the *code* parameter of CTRANSLATE, the following values specify the translation table to be used:

- 0 - The user supplied table specified in the parameter, table.
- 1 - EBCDIC to ASCII. (EBCDIC characters with no ASCII equivalent are translated to a byte of zero.)
- 2 - ASCII to EBCDIC. (The ASCII parity bit is ignored.)
- 3 - Reserved for future use.



```

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,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(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

- 4 - Reserved for future use.
- 5 - EBCDIK to JIS (Katakana data).
- 6 - JIS to EBCDIK.

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:

<i>code</i>	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.
<i>instring</i>	<code>ESTRING</code> , a byte array containing the string to be converted.
<i>outstring</i>	<code>ASTRING</code> , a byte array which will contain the ASCII characters for "JOB 2" when the intrinsic is executed.
<i>stringlength</i>	5, which specifies the length, in bytes, of the string "JOB 2".
<i>table</i>	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);
```

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,10,0);
```

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.

```

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 4-6. Using the PRINT and READ Intrinsics

## 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,Ø,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.
<i>control</i>	Ø (MPE ignores this parameter).
<i>reply</i>	The operator's reply will be returned to the array REPLY.
<i>expectedl</i>	-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	:IF	:RESUME	:SPEED
:ALTSEC	:LISTF	:SAVE	:STORE
:ALTVSET	:LISTVS	:SECURE	:STREAM
:BUILD	:MOUNT	:SETCATALOG	:TELL
:BYE	:NEWVSET	:SETDUMP	:TELLOP
:COMMENT	:PTAPE	:SETJCW	:VSUER
:DEBUG	:PURGE	:SETMSG	
:DISMOUNT	:REDO	:SHOWCATALOG	
:DSLNE	:RELEASE	:SHOWDEV	
:DSTAT	:REMOTE HELLO	:SHOWIN	
:ELSE	:RENAME	:SHOWJCW	
:ENDIF	:REPORT	:SHOWJOB	
:FILE	:RESET	:SHOWME	
:GETRIN	:RESETDUMP	:SHOWOUT	
:HELP	:RESTORE	:SHOWTIME	

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
```

```

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, REG:="? ",
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(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 4-7. Using the QUIT Intrinsic

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). The system job control word (JCW) is set to the value FATAL, with the QUIT parameter as a modifier. In this example, JCW would be set to FATAL1, or %100001.

## 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.

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. In addition, QUITPROG sets the system job control word (JCW) to the value FATAL, with the QUITPROG parameter as a modifier. Thus, in this example, JCW would be set to FATAL1, or %100001.

## 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 *segsiz* 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.

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.

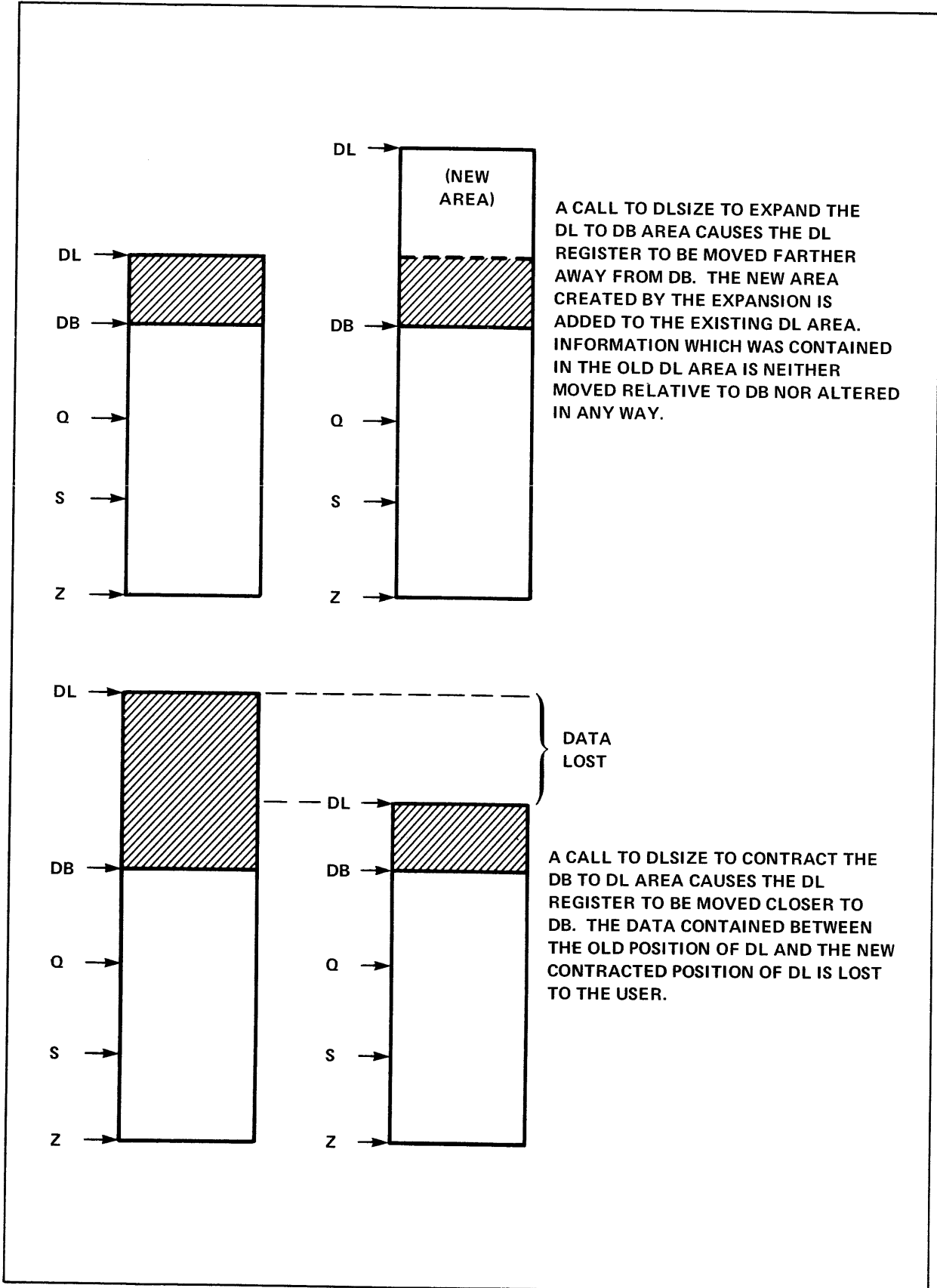
Some possible applications for the DL area are:

- 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.



A CALL TO DLSIZE TO EXPAND THE DL TO DB AREA CAUSES THE DL REGISTER TO BE MOVED FARTHER AWAY FROM DB. THE NEW AREA CREATED BY THE EXPANSION IS ADDED TO THE EXISTING DL AREA. INFORMATION WHICH WAS CONTAINED IN THE OLD DL AREA IS NEITHER MOVED RELATIVE TO DB NOR ALTERED IN ANY WAY.

DATA LOST

A CALL TO DLSIZE TO CONTRACT THE DB TO DL AREA CAUSES THE DL REGISTER TO BE MOVED CLOSER TO DB. THE DATA CONTAINED BETWEEN THE OLD POSITION OF DL AND THE NEW CONTRACTED POSITION OF DL IS LOST TO THE USER.

Figure 4-8. Expanding and Contracting the DL to DB Area

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

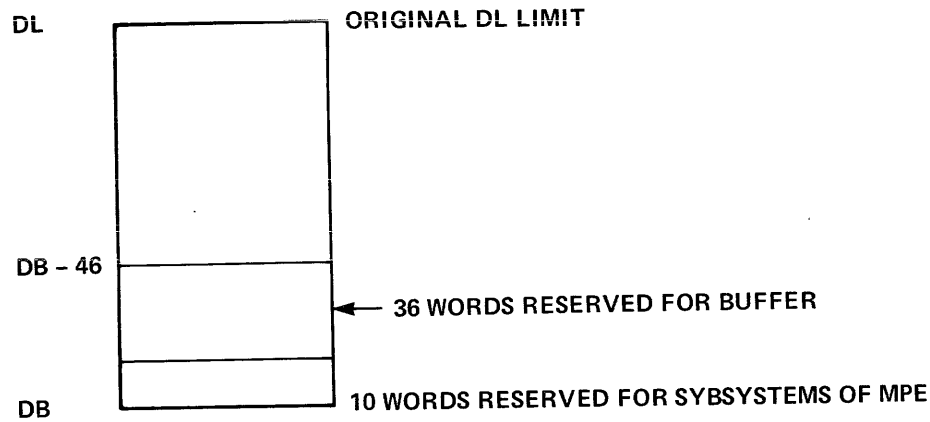
The program was PREPped with a MAXDATA = 2000 parameter.

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.



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.

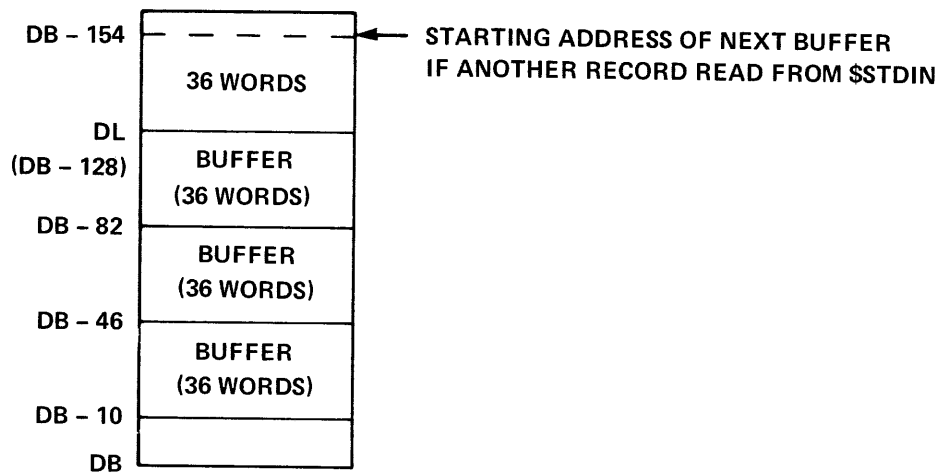
The statement

```
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.

```

00001000 00000 0 $CONTROL USLINIT
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 TOS>@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

```

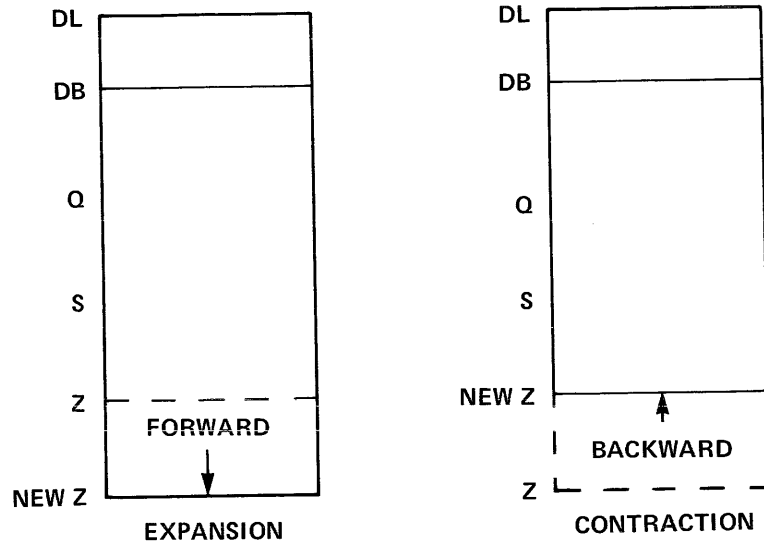
Figure 4-9. Using the DLSIZE Intrinsic



## 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.

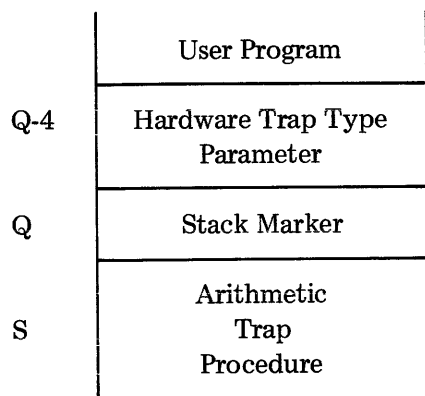


When a software arithmetic trap procedure is executed, the Index register contains the word of code being executed when the trap occurred. This information, plus, if necessary, the right stackop bit in the stacked status word, can be used to identify the offending instruction. A one-word parameter is 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

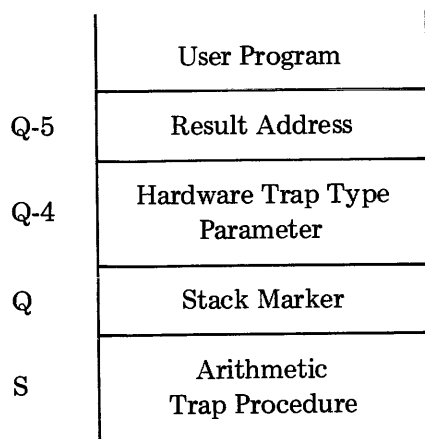
A return from the trap procedure (through an (EXIT1) instruction) will resume execution in the user code domain at the instruction following that which activated the trap procedure. The condition of the stack when the trap procedure is invoked is



### EXTENDED PRECISION FLOATING-POINT TRAPS

- Bit 10 = Extended Precision Overflow
- 9 = Extended Precision Underflow
- 8 = Extended Precision Divide by 0

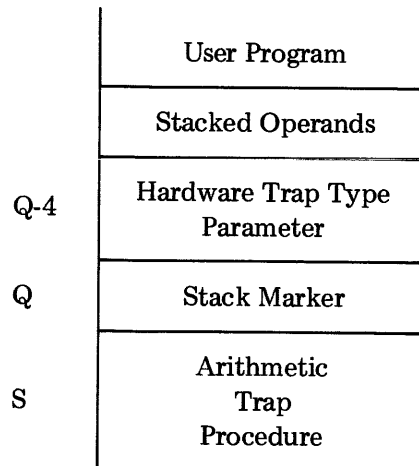
The address of the result operand is left on the stack in Q-5. An (EXIT 2) return will resume execution in the user code domain at the instruction following the one which caused the trap. The condition of the stack when the trap procedure is invoked is



## 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

```
XARITRAP(%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:=%3777777777D << - LARGEST VALUE>>
00018000 00011 2 ELSE QUOTIENT:=%1777777777D; << + 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-6	RESULT
Q-4	Hardware Trap Type Parameter (%1)
Q	Stack Marker
S	Arithmetic Trap Procedure

The value of RESULT has been left at Q-6 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.%0.%26  
PROGRAM ERROR #3: FLOATING POINT OVERFLOW
```

```
PROGRAM TERMINATED IN AN ERROR STATE. (CIERR 976)
```

```
:
```

## LIBRARY TRAP

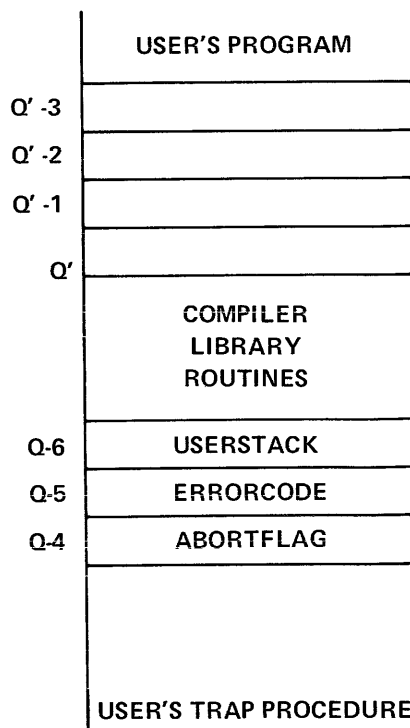
The software library trap reacts to major errors that occur during execution of procedures from the System Library. When a user program begins execution, this trap is disabled automatically. You can enable (or disable) it with the XLIBTRAP intrinsic. When enabled, the library trap passes control to a 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. OLDPLABEL 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 reference parameter indicating the type of compiler library error, described in the *HP 3000 Compiler Library Manual*.

**ABORTFLAG**

A reference parameter set before the user exits 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.

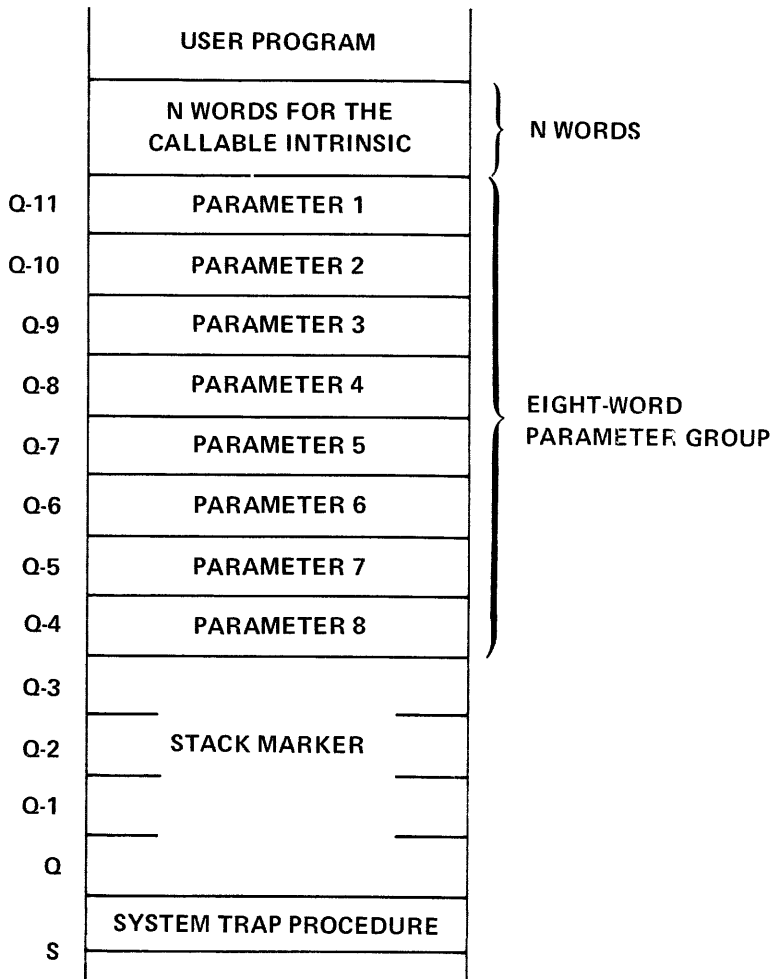
When a user program begins execution, the system trap is disabled automatically. When enabled by the XSYSTRAP intrinsic call and subsequently activated by an error, the trap transfers control to a trap procedure.

The system trap is enabled or disabled by a XSYSTRAP intrinsic call, as follows:

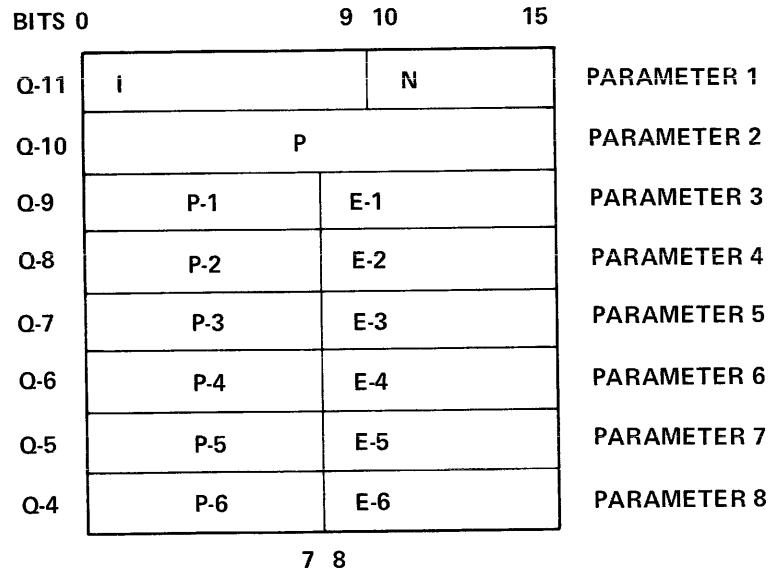
```
XSYSTRAP(PLABEL,OLDPLABEL);
```

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:

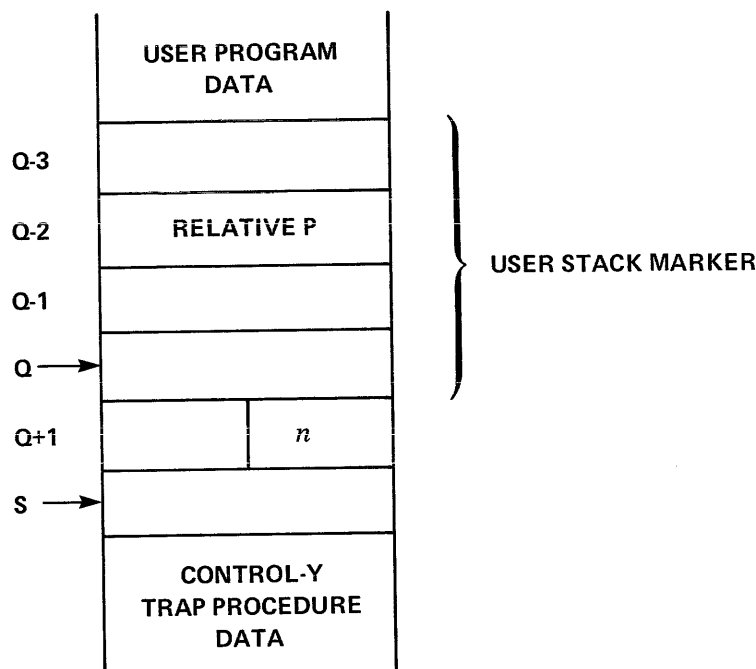
1. The input/output transactions pending between the process and the terminal are halted and flagged as though all were completed successfully.
2. Control is transferred to the trap procedure defined by you, with which you can now interact. The trap procedure executes in the same mode (privileged or 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 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, MPE searches back to the last user stack marker and sets bit 0 of relative P in that marker. No interrupt will occur until an EXIT instruction is executed through the above marker. The trap is recognized, a marker is built, and control is transferred to the trap procedure. 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 this word contains the number of words in the stack, plus the stack marker. This value will be deleted from the stack upon exit from the procedure. This value is non-zero when parameters to a system procedure (which was executing when the CONTROL-Y occurred) have been left on the stack. On return, the trap procedure must know the value contained in Q+1 and pass it to 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 BMSG(*)=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,BMSG(20)); <<CONVERT COUNTER>>
00016000 00007 2  PPRINT(MSG,16,0); <<OUTPUT COUNTER>>
00017000 00013 2  RESETCONTROL; <<REARM 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 SDEC (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
CONTROL 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.

This quantity is reset to zero on 24 day intervals at 12 o'clock midnight. Detection and correction of this case between two calls to `TIMER` (less than 24 days apart) for computing an elapsed time interval can be done as follows:

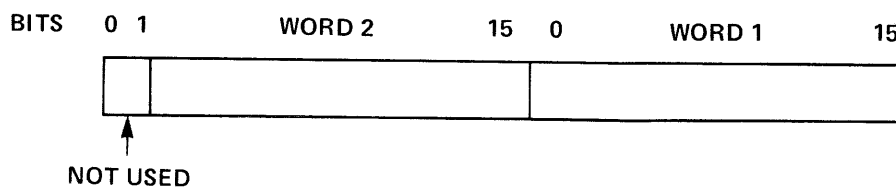
If, when subtracting a current `TIMER` count from a previous count, the result is negative, add 2,073,600,000 (the number of milliseconds in 24 days) to the result.

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:



```

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,CBUF,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 <<WRONG CHARACTER MESSAGE>>
00041000 00126 2 FWRITE(OUT,MSG(8),8,0); <<CHECK FOR ERROR>>
00042000 00134 2 IF < THEN QUIT(8) ELSE GO NEXT;
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
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:10

```

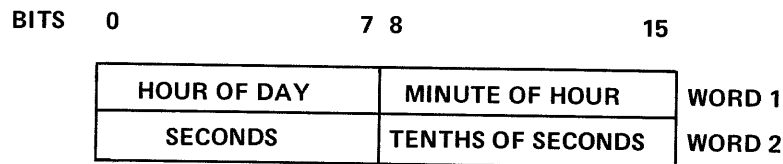
Figure 4-13. Using the TIMER Intrinsic

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).

#### 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:

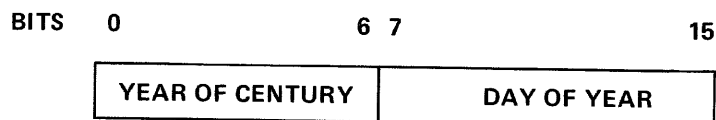


In the following intrinsic call, the above information would be returned to the double-word identifier TIME.

```
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;
```

## FORMATTING CALENDAR DATE AND TIME INFORMATION

You can format the calendar date with the `FMTCALENDAR` intrinsic, the time of day with the `FMTCLOCK` intrinsic, and the calendar date and time of day with the `FMTDATE` intrinsic. These intrinsics use the information returned by the `CALENDAR` and `CLOCK` intrinsics.

The simple program shown in Figure 4-14 illustrates the use of these intrinsics.

```
$CONTROL USLINIT
BEGIN

    BYTE ARRAY FMTDATE(0:16);
    BYTE ARRAY FMETIME(0:7);
    BYTE ARRAY DATETIME(0:26);

    ARRAY FMT'DATE(*)=FMTDATE;
    ARRAY FMT'TIME(*)=FMETIME;
    ARRAY DATE'TIME(*)=DATETIME;

    LOGICAL DATE;
    DOUBLE TIME;

    INTRINSIC CALENDAR,CLOCK,FMTCALENDAR,FMTCLOCK,FMTDATE,PRINT;

    DATE:=CALENDAR;
    TIME:=CLOCK;

    FMTCALENDAR(DATE,FMTDATE);
    PRINT(FMT'DATE,-17,%60);

    FMTCLOCK(TIME,FMETIME);
    PRINT(FMT'TIME,-8,%60);

    FMTDATE(DATE,TIME,DATETIME);
    PRINT(DATE'TIME,-27,0);

END.
```

Figure 4-14. `FMTCALENDAR`, `FMTCLOCK`, and `FMTDATE` Intrinsics Example

## INTERPROCESS COMMUNICATION

You can arrange for two processes belonging to the same job/session to communicate with each other through a job control word (JCW). 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 (JCW). A SETJCW intrinsic call could be

```
SETJCW(WORD);
```

where WORD is a 16-bit logical word whose bits are set by you. If you set bit 0 to 1, the system displays the following message when your program terminates, either normally or due to an error:

```
PROGRAM TERMINATED IN AN ERROR STATE (CIERR 976)
```

Bits 1 through 15 may be set to any pattern.

### NOTE

In batch mode, the job is terminated unless the :CONTINUE command is used. If you have a JCW of *exactly* %14000, (bits 0 and 1 only), the CIERR 976 message is replaced by CIERR 989, PROGRAM ABORTED PER USER REQUEST. See the *MPE Commands Reference Manual* for a discussion of :CONTINUE.

The job control word (JCW) can be read 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. 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;
```



## USER-DEFINED JOB CONTROL WORDS

MPE allows you to establish and manipulate job control words including the system-defined job control word "JCW." This capability overcomes a disadvantage of using the system-defined JCW, which is that because MPE uses the JCW for status information, you cannot be sure that MPE will not modify it, thus destroying whatever information you may wish to pass.

A user-defined JCW is a 16-bit logical word which resides in an MPE managed table. This table, which also holds the system-defined JCW, is shared by all processes in a job, thus any process of a job can access any JCW in the table.

The name of a user-defined JCW must start with a letter and be between 1 and 255 characters long.

A user-defined JCW is established with the PUTJCW intrinsic. This intrinsic scans the JCW table for a given JCW. If found, the JCW's value is updated to the value passed by the PUTJCW intrinsic call. If a JCW of that time is not found, the name is added to the table and assigned the value passed with the name. For example, the intrinsic call

```
PUTJCW(JCWNAME,JCWVALUE,STATUS);
```

would search the JCW table for a name which matches the name contained in JCWNAME (a byte array). If the name exists, its value is updated to the value contained in JCWVALUE. If a name matching that contained in JCWNAME is not found, the name is added to the JCW table and assigned the value contained in JCWVALUE.

The STATUS parameter indicates the status of the intrinsic call and returns an integer value to indicate this status as follows:

- 0 - Successful execution.
- 1 - Error. JCWNAME is longer than 255 characters.
- 2 - Error. JCWNAME does not start with a letter.
- 3 - Error. The JCW table is out of space.

The FINDJCW intrinsic is used to scan the JCW table for a given JCW name and return its value.

Thus, the intrinsic call

```
FINDJCW(JCWNAME,JCWVALUE,STATUS);
```

would search the JCW table for a JCW of the same name as that contained in JCWNAME. If a JCW of the same name is found, its current value is returned in JCWVALUE. If a JCW of the same name is not found, an error is returned in STATUS.

The STATUS parameter indicates the status of the intrinsic call and returns an integer value indicating this status as follows:

- 0 - Successful execution.
- 1 - Error. JCWNAME is longer than 255 characters.
- 2 - Error. JCWNAME does not start with a letter.
- 3 - Error. The JCW named in JCWNAME does not exist.

# MPE MESSAGE SYSTEM

The MPE message system consists of a message catalog (CATALOG.PUB.SYS), the Help subsystem catalog (CICAT, containing descriptions of all MPE commands), any number of user message catalogs, a program (MAKECAT) for building message catalogs, and an intrinsic (GENMESSAGE) used to insert parameters in messages in a catalog.

## MESSAGE CATALOG

A message catalog *must be a standard editor-type file* containing sets of messages. That is, a numbered file which contains 80 byte records in a fixed record format. The sets serve to break a catalog into manageable portions. After a message file is created, the MAKECAT program is used to build a catalog that is readable by the message system. This catalog file can still be texted into the editor, but it now contains a directory (written as a user label by MAKECAT).

Messages in the catalog can be of any length and can contain up to five parameters (parameters are indicated in a message by the symbol !). Continuation of a message is indicated by “%” or “&” at the end of a line. The “%” symbol indicates that the message is continued and that a carriage return, line feed will be issued to the terminal. The “&” symbol indicates that the message is continued on the same line with no carriage return, line feed. The GENMESSAGE intrinsic ignores all blanks between the last non-blank character of a message and the continuation character. This allows free-formatting of the continuation character.

Message sets are indicated by “\$SET n” starting in column 1 (the rest of the line is treated as a comment). Maximum value for n is 62. Comments can be inserted in the catalog by placing “\$” in column 1 of a line. Message numbers are positive integers, need not be contiguous, but must be in ascending order. After processing by the program MAKECAT, the catalog file contains records of 80 bytes, blocked 16, in 32 extents. (The system message catalog is only one extent, however.)

The format of the message catalog is as follows:

```
$SET 1      SYSTEM MESSAGES
1 LDEV #! IN USE BY FILE SYSTEM
2 LDEV #! IN USE BY DIAGNOSTICS
3 LDEV IN USE, DOWN PENDING
5 IS “!” ON LDEV#! (Y/N)?
.
.
.
$ MESSAGE 35 IS TWO LINES LONG, A PARAMETER STARTS THE
$ FIRST LINE AND THE SECOND LINE IS “HP32002”
35 !%
HP32002B.00.!
.
.
.
276 LDEV # FOR “!” ON ! (NUM)!
$
$SET 2 CIERROR MESSAGES
82 STREAM FACILITY NOT ENABLED: SEE OPERATOR. (CIERR 82)
200 MORE THAN 30 PARAMETERS TO BUILD COMMAND. (CIERR 200)
```

.  
. .  
204 FILE COMMAND REQUIRES AT LEAST TWO PARAMETERS, INCLUDING THE %  
FORMAL NAME OF THE FILE. (CIERR 204)  
. . .

## MAKECAT PROGRAM

The program MAKECAT.PUB.SYS is used to build message catalogs (and Help catalogs). The program's input file has the formal designator INPUT. The program has the following entry points:

No entry point	Reads from input file and builds a temporary file (formal designator CATALOG). Also renames any old temporary CATALOG, CATnn, using an archival numbering scheme (i.e., CAT1, CAT2, etc.).
BUILD	(Must log on under MANAGER.SYS to use this entry point.) Reads from input file, builds the system message catalog (formal designator CATALOG), and installs the message system. Existing catalog is renamed CATnn according to the same scheme as for no entry point (above). Installation of the message system means moving the directory contained in the user label of the catalog into a data segment. The Data Segment Table (DST) number and the disc address of CATALOG are placed in the system global area. The message system may be installed while the system is running.
DIR	(Must log on under MANAGER.SYS to use this entry point.) Installs the system message catalog (does not build a new one). Opens input file, moves the directory in the CATALOG into a data segment, and places the DST number and disc address of CATALOG in the system global area. This entry point may be used when the message system seems to be malfunctioning, but the catalog is intact. (For example, MPE is issuing "MISSING MSG SET=mm. MSG=nn" at terminals and the system console.) This may be done while the system is running.
HELP	(Must have System Manager of System Supervisor capability to use this entry point.) Used to build the Help catalog. Reads input file and builds a Help catalog (formal designator HELPCAT).

EXAMPLES. To use MAKECAT to build your own message catalog, enter:

```
:FILE INPUT=CAT15  
:RUN MAKECAT.PUB.SYS  
** VALID MESSAGE CATALOG (printed if no errors in catalog CAT15)  
:SAVE CATALOG
```

To use MAKECAT to modify the system message catalog:

1. Text CATALOG.PUB.SYS into the Editor.
2. Make the desired changes.
3. Keep the file under a new name and exit the Editor.
4. Log on as MANAGER.SYS, and enter:  
:FILE INPUT=catname.group.account  
:RUN MAKECAT,BUILD  
\*\* NEW CATALOG INSTALLED

To reinstall the message catalog if MPE is printing "MISSING MSG. SET=mm. MSG=nn," enter:

```
:HELLO MANAGER.SYS
:FILE INPUT=CATALOG
:RUN MAKECAT,DIR
** NEW CATALOG INSTALLED
```

To build a Help catalog for the command interpreter, enter:

```
::HELLO MANAGER.SYS
:PURGE CICAT
:FILE INPUT=catalog.group.account
:RUN MAKECAT,HELP
  END OF PROGRAM
:RENAME HELPCAT, CICAT
```

## USING THE GENMESSAGE INTRINSIC TO INSERT PARAMETERS IN MESSAGES

The GENMESSAGE intrinsic can be used to access the MPE message system. GENMESSAGE is called with a message number from a catalog as a parameter. The message system fetches the message from a message catalog, inserts parameters, then routes the message to a file or returns the message in a buffer to the calling program.

In order to use the message catalog, the program must first open the message catalog, then call GENMESSAGE with the file number, message set number, and message number. The file must be opened with the aoptions nobuf and multi-record access.

### NOTE

The file must be opened with foptions old, permanent, ASCII (foptions 5), and aoptions nobuf and multi-record access (aoptions %420).

Parameters may be inserted into the message from the catalog. The parameters are passed to the message with the parm1, parm2, parm3, parm4, and parm5 parameters in the GENMESSAGE intrinsic call and are inserted in the message wherever a "!" is found. Parameters are inserted in the following order: parm1 substitutes for the leftmost "!" in the message, parm2 for the next leftmost, and so forth. If parm(n) is present, parm(n-1) must be present (for example, you cannot specify parm3 unless parm1 and parm2 are specified).

Figure 4-15 contains a simple program that inserts the value 95 into message number 11 in message set 1 in the message catalog CATALOG.PUB.SYS. The complete message then is displayed on the terminal. Note that the file CATALOG.PUB.SYS is equated to CATALOG with a :FILE command, then the name CATALOG is used in the FOPEN call (passed to FOPEN in byte array BUFF). Note also that the file is opened with aoptions nobuf and multi-record access (aoptions %420). The message set (1) and message number (11) are included as parameters in the GENMESSAGE call. The parameter parmask is set to %10000 and parm1 (NUMBER) has the value 95. The complete message is returned in BUFF, which is then printed on the terminal with the PRINT intrinsic.

```

:SPLPREP TEST,MSGTFST

PAGE 0001  HPJ2100A,06.4J [4W] (C) HEWLETT-PACKARD COMPANY 1976

00001000 00000 0  $CONTROL USLIMIT
00002000 00000 0  BEGIN
00003000 00000 1
00004000 00000 1  BYTE ARRAY BUFF(0:255);
00005000 00000 1  ARRAY OUTBUFF(*)=BUFF;
00006000 00000 1
00007000 00000 1  INTEGER FILENUM,MSGLEN,NUMBER:=95;
00008000 00000 1
00009000 00000 1  INTRINSIC FOPEN,PRINTF,GENMESSAGE,PRINT;
00010000 00000 1
00011000 00000 1  MOVE BUFF:="CATALOG ";
00012000 00016 1  FILENUM:=FOPEN(BUFF,5,%420);
00013000 00026 1  IF <> THEN PRINTFILEINFO(FILENUM);
00014000 00031 1
00015000 00031 1  MSGLEN:=GENMESSAGE(FILENUM,1,11,BUFF,,%10000,NUMBER);
00016000 00045 1
00017000 00045 1  PRINT(OUTBUFF,-MSGLEN,0);
00018000 00051 1
00019000 00051 1  END.
PRIMARY DB STORAGE=%005;  SECONDARY DB STORAGE=%00200
NO. ERRORS=0000;          NO. WARNINGS=0000
PROCESSOR TIME=0:00:00;  ELAPSED TIME=0:00:29

END OF COMPILE
END OF PREPARE
:SAVE MSGTEST
:FILE CATALOG=CATALOG.PUB.SYS
:RUN MSGTEST

LDEV#95 NOT READY

END OF PROGRAM

```

Figure 4-15. GENMESSAGE Intrinsic Example

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 2-71).

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 2-49).

**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. There is no defined end-of-record character in binary mode, 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 not 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 (postspace). The prespace operation is simulated by performing a print operation and then filling the line printer buffer. Note that a carriage control code of %320 will append data to the current contents of the line printer buffer, whether prespace or postspace is selected.

Table 5-1 describes the differences between the 5 subtypes of line printers.

The HP 2608/2610/2614 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 2607/2613/2617/2617J/2618/2619 printers have only a 4-bit space count that imposes a maximum vertical spacing of 15 lines at a time. MPE will simulate vertical spacing of more than 15 lines, when carriage control codes of %200 to %277 are used, by concatenating as many 15-line spacings as necessary. The final spacing may consist of less than 15 lines.

Table 5-1. Line Printer Differences

SUBTYPE	HP PRODUCT NO.	SUPPRESS SPACE	VFC CHANNELS AVAILABLE**
0	2610	YES	1-8
0	2614	YES	1-8
1	2607	NO*	1-8
2	2613	YES	1-12
2	2617	YES	1-12
2	2618	YES	1-12
2	2619	YES	1-12
3	2617J	YES	1-8
4	2608	YES***	1-16

\* A suppress space request (carriage control code = %53 or %200) will result in a single space without automatic page eject.

\*\* Carriage control codes %300 to %317 specify VFC channels 1-16 respectively. A request to skip to a channel which is undefined for that subtype printer will result in a single space (with or without automatic page eject, the same as for carriage control code %10).

\*\*\* Data may be lost if spacing is suppressed on two or more consecutive requests.

To change the mode control settings (pre/post spacing and auto/no auto page eject) FWRITE is used with carriage controls %100 — %103 and %400 — %403. If FWRITE is called with one of these carriage controls and count=0 (count =1 if imbedded control), then no physical I/O will occur; the only effect is changing the mode.

## 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. Carriage-Control Directives

OCTAL CODE	ASCII SYMBOL	CARRIAGE ACTION
%40	" "	Single space (with or without automatic page eject).
%53	"+"	No space, return (next printing at column 1). Not valid on 2607 (results in single space without automatic page eject).
%55	"_"	Triple space (with or without automatic page eject).*
%60	"0"	Double space (with or without automatic page eject).*
%61	"1"	Page eject (form feed). Selects VFC Channel 1. Ignored if: Post-space mode: The current request has a transfer count of 0 and the previous request was an FOPEN or FCLOSE or an FWRITE which specified a carriage-control directive of %61. Pre-space mode: Both the current request and the previous request have transfer counts of 0, and the current request and previous request are any combination of FOPEN, FLCLOSE or an FWRITE specifying a carriage-control directive of %61.
%2nn (nn is any octal number from 0 through 77)		Space nn lines (no automatic page eject). %200 not valid for 2607 (results in single space without automatic page eject).
%300-%307		Select VFC Channel 1-8 (2607)
%300-%313		Select VFC Channel 1-12 (2613, 2617, 2618, 2619)
%300-%317		Select VFC Channel 1-16 (2608)
		NOTE: Channel assignments shown below are the HP standard defaults.
%300		Skip to top of form (page eject).
%301		Skip to bottom of form.
%302		Single spacing with automatic page eject.
%303		Skip to next odd line with automatic page eject.
%304		Skip to next third line with automatic page eject.
%305		Skip to next 1/2 page.
%306		Skip to next 1/4 page.
%307		Skip to next 1/6 page.
%310		Skip to bottom of form.
%311		User option (2613/17/18/19), skip to one line before bottom of form (2608)

**\*Note:**

*Series 30/33/44:* If these codes are selected with automatic page eject in effect (by default or following an Octal Code of %102 or %402), the resulting skip is to a location absolute to the page. A code of %60 is replaced by %303 and %61 is replaced by %304. Thus the resulting skip may be less than two or three lines, respectively.

If automatic page eject is not in effect, a true double or triple space results, but the perforation between pages is not automatically skipped.

*Series 11/111:* If these codes are selected with automatic page eject in effect, %60 and %61 are replaced by two or three %302 codes, respectively. This results in true double or triple spacing, and also skips the perforation.

If automatic page eject is not in effect, the behavior is the same as for Series 30/33/44.

Figure 2-3. Carriage-Control Directives

OCTAL CODE	ASCII SYMBOL	CARRIAGE ACTION
%312		User option (2613/17/18/19), skip to one line before top of form (2608)
%313		User option (2613/17/18/19), skip to top of form (2608)
%314		Skip to next seventh line with automatic page eject.
%315		Skip to next sixth line with automatic page eject.
%316		Skip to next fifth line with automatic page eject.
%317		Skip to next fourth line with automatic page eject.
%320		No space, no return (next printing physically follows this).
%2-%37		
%41-%52		
%54		
%56-%57		
%62-%77		Same as %40
%104-%177		
%310-%317 (2607)		
%314-%317 (2613/17/18/19)		
%321-%377		
%400 or %100		Sets post-space movement option; this first prints, then spaces. If previous option was pre-space movement, the driver outputs a line with a skip to VFC Channel 3 to clear the buffer.
%401 or %101		Sets pre-space movement option; this first spaces, then prints.
%402 or %102		Sets single-space option, with automatic page eject (60 lines per page).
%403 or %103		Sets single-space option, without automatic page eject (66 lines per page).

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 terminal controller (each controller controls up to 16 terminals). The terminal controller supports 103A and 202A modems, and hardwired terminals.

**TERMINAL TYPES.** The terminals shown in Table 5-3 are supported by MPE. 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 linefeed. (The superscript c denotes a control character. Thus, "X <sup>c</sup> " means "CONTROL-X.")
H <sup>c</sup>	Deletes the previous character. (To delete n characters, enter n 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 linefeed 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 X <sup>c</sup> is received and tape mode is in effect, no exclamation points (!!!) are sent to the terminal. If used after S <sup>c</sup> , Q <sup>c</sup> also resumes write operation during output (cancels S <sup>c</sup> ).
R <sup>c</sup>	Indicates the beginning of a block mode read and starts a special block mode timer. If the read doesn't complete successfully within the timer period, (approximately twice the expected read time determined from line speed and number of characters to read plus thirty seconds), the read is returned with an FSERR 27. Normal block mode transfers proceed as follows: The computer sends DC1 to the terminal to initiate a read. If the user has pressed ENTER for a block mode read, the terminal then sends DC2 (R <sup>c</sup> ) to the computer to indicate a block mode read; the computer sends another DC1 to the terminal to initiate the transfer; the terminal then sends the data to the computer. NOTE: R <sup>c</sup> has special significance only for termtypes which support block mode.
S <sup>c</sup>	Suspends the write operation during output.
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.
BREAK	Requests a system break.
ESC:	Places the terminal in the echo-on mode so that characters input are echoed on the terminal by MPE.
ESC;	Places the terminal in echo-off mode so that characters input are not echoed on the terminal by MPE.

**Key****Meaning**

**LINEFEED** For any terminal with a linefeed entry, the log-on user may strike this key and a carriage return will be echoed. The linefeed character is not transmitted to the input buffer. This mechanism permits multiple lines to be entered in response to a single read request (for example, the length of a read request from a terminal is not constrained by the carriage or line width).

The defined control characters Xc, Hc, Qc, Sc, and Yc are recognized even when following an ESC key entry. However, entry of ESC followed by any other 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.

Table 5-3. Terminals Supported by MPE

TERMINAL TYPE	DESCRIPTION
0	HP 2749B (ASR-33 EIA compatible) Terminal (10 characters per second (cps)).
1	ASR-37 Teleprinter Terminal with Paper Tape Reader/Punch (10 cps).
2	ASR-35 EIA-compatible Terminal (10 cps).
3	Execuport 300 Data Communications Transceiver Terminal (10/15/30 cps).
4	HP 2600A or Datapoint 3300 Keyboard Display Terminal (10/15/30/60/120/240 cps).
5	Memorex 1240 Communications Terminal (10/15/30/60 cps).
6	HP 2762A/B (General Electric Terminet 300 or 1200), or Data Communications Terminal, Model B (10/15/30/120 cps) with Paper Tape Reader/Punch, Option 2.
9	HP 2615A Terminal (Beehive MiniBee) (10/15/30/60/120/240 cps).
10	HP 2640A/B, HP 2641A, HP 2644A, or HP 2645A Character Mode or full program control of block mode transmission (10-240 cps).
11	HP 2640A/B, HP 2641A, HP 2644A, or HP 2645A. Allows user to use block mode without program control of block mode transmission. Requires user to position cursor before pressing ENTER. Recommended for speeds exceeding 30 cps when you expect to switch between character mode and block/line mode. May not be used in block/page mode. (10-240 cps.)
12	HP 2645K Katakana/Roman Data Terminal.
13	Message switching network or other computer.
14	Multi point Terminal.
15	HP 2635A Printing Terminal. 8-bit protocol (for second character set).
16	HP 2635A Printing Terminal. 7-bit protocol (standard character set).



## 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 FSETMODE, 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(filenum,controlcode,speed);
```

The parameters are

*filenum*                    *integer by value (required)*  
A word identifier supplying the file number of the terminal for which the speed is to be changed.

<i>controlcode</i>	<i>integer by value (required)</i> The decimal integer 10 to change the input speed or 11 to change the output speed.
<i>speed</i>	<i>logical (required)</i> A word identifier that specifies the new speed desired: 10, 14, 15, 30, 60, 120, 240, 480, or 960, cps. When the FCONTROL intrinsic is executed, the previous input or output speed is returned to the calling process through the <i>speed</i> 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.

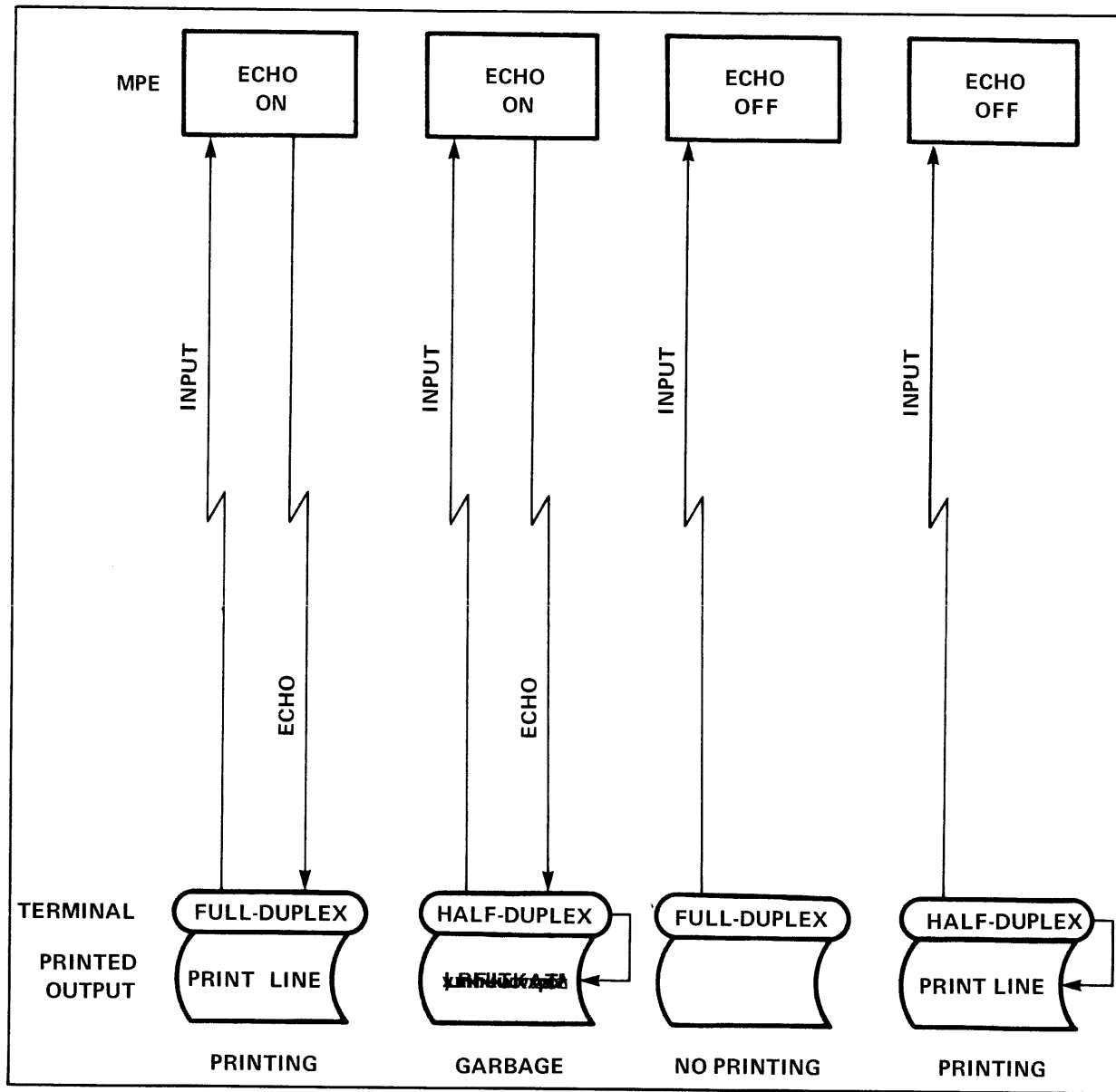


Figure 5-1. Echo Facility vs Duplex Mode

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. These control code options are not valid for terminals configured as termtype 12 (HP 2645K).

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 24 to enable parity checking, or 23 to disable parity checking. (Not valid for termtype 12 (HP 2645K).)
- anyinfo*                            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. To inhibit carriage return, linefeed after READ or FREAD, use FSETMODE (see page 2-84).

The format for this application of the FCONTROL intrinsic is

IV            IV            L  
FCONTROL(*filenum*,*controlcode*,*anyinfo*);

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 19 to enable tape mode, or 18 to disable tape mode.
<i>anyinfo</i>	<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 programmatically 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

<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 21 to enable the time, or 20 to disable the timer.
<i>anyinfo</i>	<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 set a time out on FREAD. 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 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY INNAME(0:5):="INPUT ";
00004000 00004 1 BYTE ARRAY OUTNAME(0:6):="OUTPUT ";
00005000 00005 1 INTEGER IN,OUT,LGTH,DUMMY,TIME,TIMEOUT:=10;
00006000 00005 1 ARRAY BUFR(0:3):="TYPE X",0;
00007000 00004 1 BYTE ARRAY CBUF(*)=BUFR;
00008000 00004 1 ARRAY INSTRUCTIONS(0:34):="REACTION TIMER: ",%6412,
00009000 00011 1 "TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN. ";
00010000 00043 1 ARRAY MSG(0:24):="TRY AGAIN? (Y/N)","WRONG CHARACTER.",
00011000 00020 1 %6412,"YOU'RE TOO SLOW!";
00012000 00031 1 ARRAY RESPONSE(0:16):="REACTION TIME: MILLISECONDS";
00013000 00021 1 BYTE ARRAY CHRESP(*)=RESPONSE;
00014000 00021 1
00015000 00021 1 INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,ASCII,TIMER,QUIT;
00016000 00021 1
00017000 00021 1 <<END OF DECLARATIONS>>
00018000 00021 1
00019000 00021 1 IN:=FOPEN(INNAME,%45); <<$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 CHUF(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 CHUF(5)<>CHUF(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,CHRESP(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 CHUF(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
PROCESSOR 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(filename,controlcode,inputtime);
```

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 22.

*inputtime*                      *logical (required)*  
A word to which is returned the input time (in seconds/100).

The condition codes are

CCE                      Request granted.

CCG                      Request granted, but the result overflowed 16 bits (*inputtime* was greater than 655.35 seconds).

CCL                      Request denied because the file number specified did not belong to this process or the device is not a terminal.

Refer to figure 5-2. The statement

```
FCONTROL(IN,22,TIME);
```

reads the result from the terminal input timer. This result is returned to the word TIME.

The statement

```
ASCII(TIME*10,10,CRESP(15));
```

multiplies the value of TIME by 10 and converts this result to an ASCII string so that the user's reaction time, in milliseconds, can be displayed. The resulting ASCII string is stored in the byte array CRESP, starting at the 16th position (CRESP(15)). The statement

```
FWRITE(OUT,RESPONSE,17,0);
```

displays the reaction time. (Arrays CRESP and RESPONSE have been equivalenced, see statements 12 and 13.)

**Defining Line-Termination Characters for Terminal Input.** Normally, when using a terminal, you indicate the end of a line by entering a carriage return (with the RETURN key on most terminals). With the FCONTROL intrinsic, however, you can specify that an additional character, such as an equal sign, a period, or an exclamation point, be recognized as the standard line terminator. On subsequent read operations during your process, the input line is terminated by the specified character. That character is returned to your buffer. No carriage return or line feed is generated.

The format for this application of the FCONTROL intrinsic is

```
IV            IV            L  
FCONTROL(filenum,controlcode,character);
```

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 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 because the file number specified did not belong to this process or the device is not a terminal.

The following characters are not recognized as line-terminating characters during normal reads:

ASCII Character		Octal Code
Backspace	(H <sup>c</sup> )	10
Line Feed	(J <sup>c</sup> )	12
Carriage Return	(M <sup>c</sup> )	15
X-ON	(Q <sup>c</sup> )	21
DC2	(R <sup>c</sup> )	22
X-OFF	(S <sup>c</sup> )	23
Line Delete	(X <sup>c</sup> )	30
Control-Y	(Y <sup>c</sup> )	31
Escape	(I <sup>c</sup> )	33
Del	--	177

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 %56 (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 (to or from block mode terminals such as the HP 2644/2645) 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(filename,controlcode,anyinfo);
```

The parameters are

<b><i>filename</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. This control code option is not valid for terminals configured as `termtyp 12` (HP 2645K).

The format for this application of the FCONTROL intrinsic is

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

The parameters are

<b><i>filename</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. (Not valid for <code>termtyp 12</code> (HP 2645K).)

***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(filenum,controlcode,param);
  
```

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 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(filename,controlcode,param);
```

The parameters are

<b><i>filename</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(filename,controlcode,param);
```

The parameters are

<b><i>filename</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 Table 5-3) 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 those that you can specify for 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.

To inhibit carriage return, linefeed after READ or FREAD, use the FSETMODE intrinsic (see page 2-84).

## 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 USLIMIT
BEGIN
INTEGER ARRAY RUFADR(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;

REGYN
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,%401);

END;

PROCEDURE FCARD(RECODE,FILENUM,RUFADR,COUNT,STATUS);
INTEGER ARRAY RUFADR;
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,RUFADR,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:=(%15,%12);
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 RETRANS;
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:=":EQJ, :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,FILENUM,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 RIN associated with this resource. 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 global RIN at a time. But processes run by users having the *Multiple RIN Optional Capability* can lock more than one global 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 unrelated process level are called *global* RIN's. Global RIN's are used to exclude simultaneous access of a resource by two or more processes. 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 process at a time with the LOCKGLORIN intrinsic. Once a RIN is locked, any other processes 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.

UNLOCKGLORIN does not check whether the *rinnum* parameter specifies the most recently locked global RIN. When global RIN's are locked and unlocked in any order by concurrent processes, deadlocks can occur. An effective way to avoid deadlocks is to assign an ordering of RIN's which is used by all processes locking them.

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 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  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); <<SSTDIN>>
00024000 00012 1  OUT:=FOPEN(OUTPUT,%414); CCL(2); <<SSTDLIST>>
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);
```

The RIN's acquired are identified by RIN numbers 1 through 6. Note that Multiple RIN capability is not required; it is the user's responsibility to avoid deadlocks.

## 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, . . .).	.

<pre> FWRITE (LP, . . .); . . . UNLOCKLOCRIN (1); . . . </pre>		<pre> FWRITE (LP, . . .); . . . UNLOCKLOCRIN (1); . . . </pre>
--	--	--

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.

#### IDENTIFYING LOCAL RIN OWNERS

LOCRINOWNER allows you to identify at any given time the PIN of the process that has a particular local RIN locked. This information can be useful, for example, in situations where father and son processes are being synchronized through calls to the ACTIVATE and SUSPEND intrinsics. (See descriptions of ACTIVATE and SUSPEND in the Process Handling Capability Section)

Consider the following example in which a father process acts as a monitor for several son processes. The father waits SUSPENDED at the top of its loop to be ACTIVATED by any son. When ACTIVATED, the father locks a RIN to synchronize its communication with the son. LOCRINOWNER is used to determine the PIN of the son that ACTIVATED the father, since that son will have the "whichson" RIN locked. The father can then do whatever processing it needs to do. The father finally ACTIVATES the son that ACTIVATED the father process and SUSPENDS, releasing the synchronization RIN and waiting for the next son to ACTIVATE it again.



<<Example of Process Synchronization with LOCRINOWNER>>

```
equate whichsonrin = 1,  
        synchrin   = 2,  
        waitforfather = 1,  
        waitforson   = 2;
```

<<father process>>

```
  .  
  .  
soncount := 0;  
while soncount <= maxsons do  
begin  
  SUSPEND (waitforson, synchrin);  
  LOCKLOCRIN (synchrin);  
  owner := LOCRINOWNER (whichsonrin);  
  .  
  .  
  soncount := soncount + 1;  
  ACTIVATE (owner);  
end;  
  .  
  .  
  .
```

<<son process>>

```
  .  
  .  
LOCKLOCRIN (whichsonrin);  
LOCKLOCRIN (synchrin);  
ACTIVATE (father);  
SUSPEND (waitforfather, synchrin);  
UNLOCKLOCRIN (whichsonrin);  
  .  
  .  
  .
```

### **FREEING LOCAL RIN'S**

To free all local RIN's currently reserved for your job, the FREELOCRIN intrinsic is called, as follows:

```
FREELOCRIN;
```

# 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 instructions 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  SCONTROL 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,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 = SEG DVR.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

```
SEG DVR.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 SEG DVR.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.

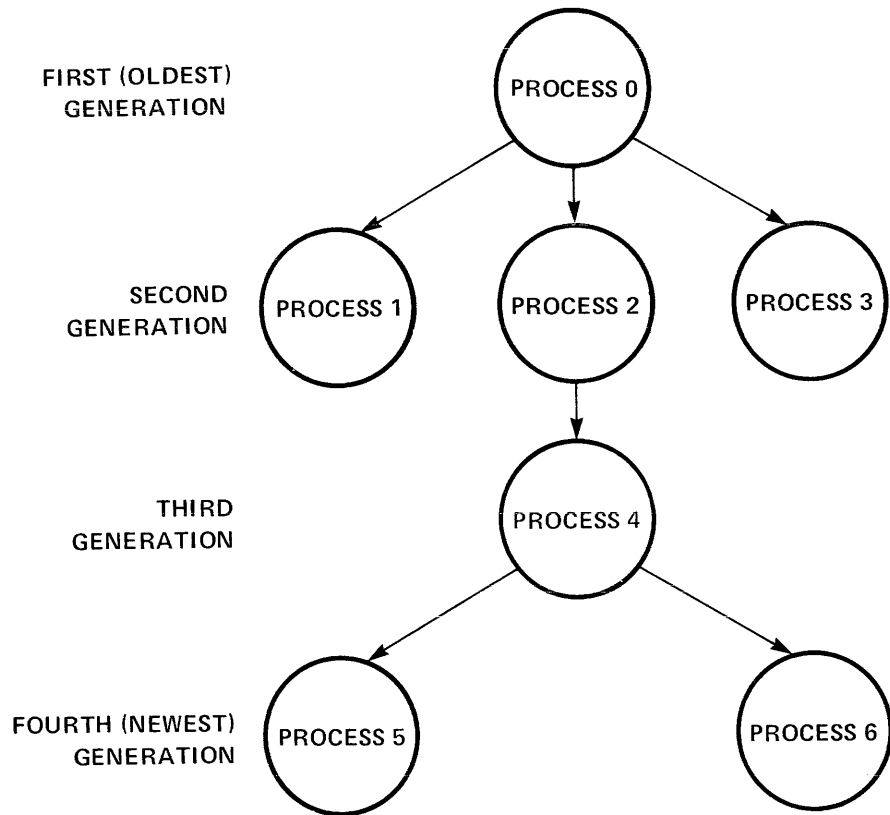
The form of the TERMINATE intrinsic call is

```
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.



(WHEN PROCESS 2 IS DELETED, THE FOLLOWING STRUCTURE RESULTS.)

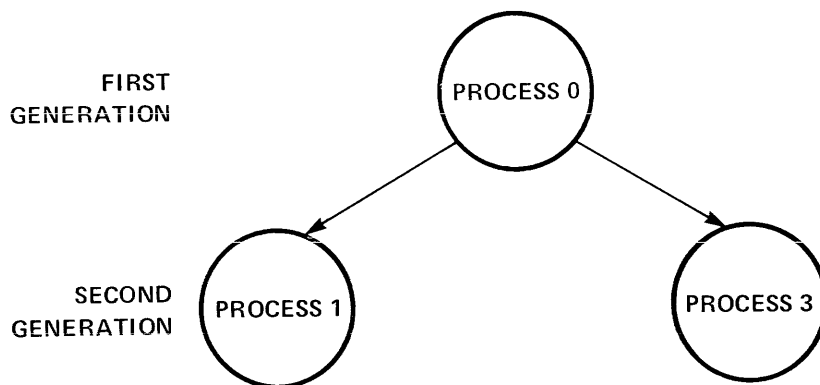


Figure 7-2. Process Deletion

## 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);
```

SONPIN contains the Process Identification Number (PIN) of the son process. 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. (Zero for father 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 = TRUE (bit 15 = 1), the intrinsic will wait until the incoming mail is ready for collection. If WAITSTAT = FALSE (bit 15 = 0), 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 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 = DS or ES priority class. 010 = CS priority class. 100 = Linearly scheduled (AS or BS Master queue).
	(0:4)	Reserved for MPE. These bits are set to zero by the system.

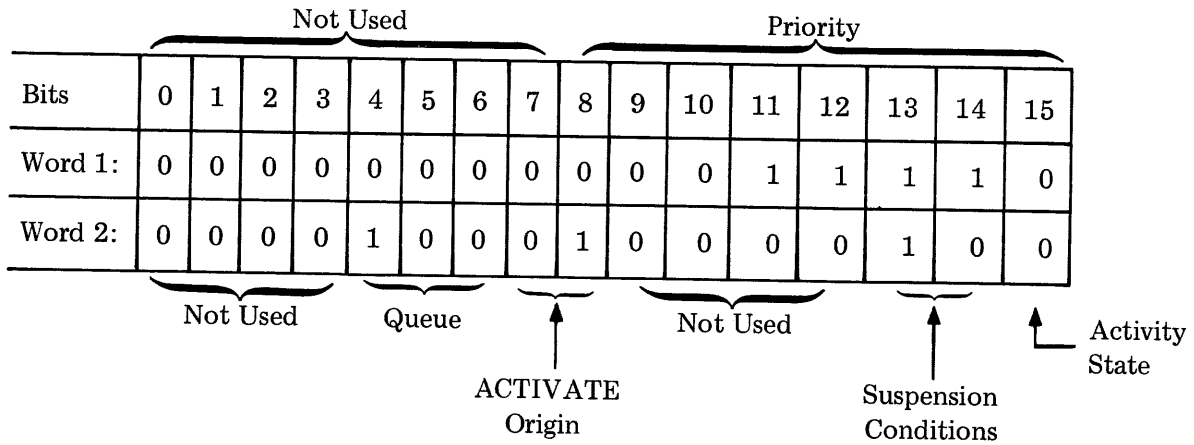
For example, to request information about its father, the following intrinsic call could be used:

```
INFO:=GETPROCINFO(0);
```

The 0 parameter specifies that the process is the father. If the process were 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 30 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,DSLGLTH:=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 GUIT#,
00015000 00001 1  CCNE= IF <> THEN GUIT#;
00016000 00001 1  INTRINSIC FOPEN,FREAD,FWRITE,GETDSEG,DMOVOUT,BINARY,QUIT;
00017000 00001 1
00018000 00001 1  <<END OF DECLARATIONS>>
00019000 00001 1
00020000 00001 1  IN:=FOPEN(INPUT,%45); CCL(1); <<SSTDIN>>
00021000 00001 1  OUT:=FOPEN(OUTPUT,%414,1); CCL(2); <<SSTDLIST>>
00022000 00012 1
00023000 00025 1  FWRITE(OUT,HEAD,15,0); CCNE(3); <<PROGRAM ID>>
00024000 00025 1
00025000 00035 1  GETDSEG(DSINDX,DSLGLTH,"JD"); CCL(4); <<SHARED EXTRA DS>>
00026000 00035 1
00027000 00044 1  FWRITE(OUT,REGST,6,%320); CCNE(5); <<REQUEST CALENDAR YEAR>>
00028000 00044 1  LGTH:=FREAD(IN,BUFR,-4); CCNE(6); <<INPUT YEAR>>
00029000 00054 1  YEAR:=BINARY(HBUF,LGTH); CCNE(7); <<CONVERT YEAR>>
00030000 00065 1
00031000 00075 1  IF YEAR MOD 4 = 0 THEN MAXDAY(1):=29; <<FIX FEB FOR LEAP YEAR>>
00032000 00075 1
00033000 00105 1  FOR MONTH:=0 UNTIL 11 DO <<INDEX 12 MONTHS>>
00034000 00105 1  FOR DAY:=0 UNTIL MAXDAY(MONTH)-1 DO <<INDEX DAYS IN EA MONTH>>
00035000 00112 1  BEGIN
00036000 00127 1  CALENDAR(MONTH*31+DAY):=DATE; <<SET IN JULIAN DATE>>
00037000 00132 2  DATE:=DATE+1; <<INCR JULIAN DATE>>
00038000 00140 2  END;
00039000 00141 2
00040000 00143 1  DMOVOUT(DSINDX,0,372,CALENDAR); <<JULIAN CALENDAR TO DS>>
00041000 00143 1  CCNE(8); <<CHECK FOR ERROR>>
00042000 00150 1
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 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,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); <<SSTDIN>>
00019000 00012 1 OUT:=FOPEN(OUTPUT,%414,1); CCL(2); <<SSTDLIST>>
00020000 00025 1
00021000 00025 1 CREATE(PFILE,,PIN1," 1"); CCNE(3); <<SON 1 USES TERMID1 FILE>>
00022000 00037 1 CREATE(PFILE,,PIN2," 2"); CCNE(4); <<SON 2 USES TERMID2 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 DSBOS)

```

00001000 00000 0 SCONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY NAME(0:7):="TERMIO# ";
00004000 00005 1 BYTE ARRAY DEV(0:4):="TERM ";
00005000 00004 1 INTEGER FNO, LGTH, MONTH, DAY, DSLGTH, JDATE, CURRENT:=-1;
00006000 00004 1 LOGICAL DSINDEX, 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 HBUFR(*)=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(DSINDEX, DSLGTH, "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(DSINDEX, (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 DSBOSS (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
```

DSBOSS (figure 8-2)

```
:PREP $OLDPASS,DSBOSS;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.



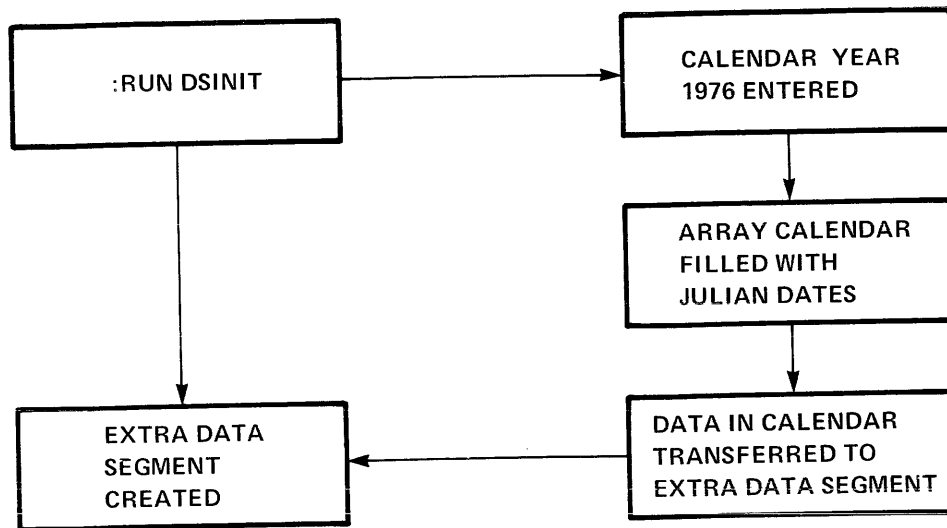
		DAYS																															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	
MONTHS	JAN	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	FEB	1	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	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 (DSBOSS) 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.

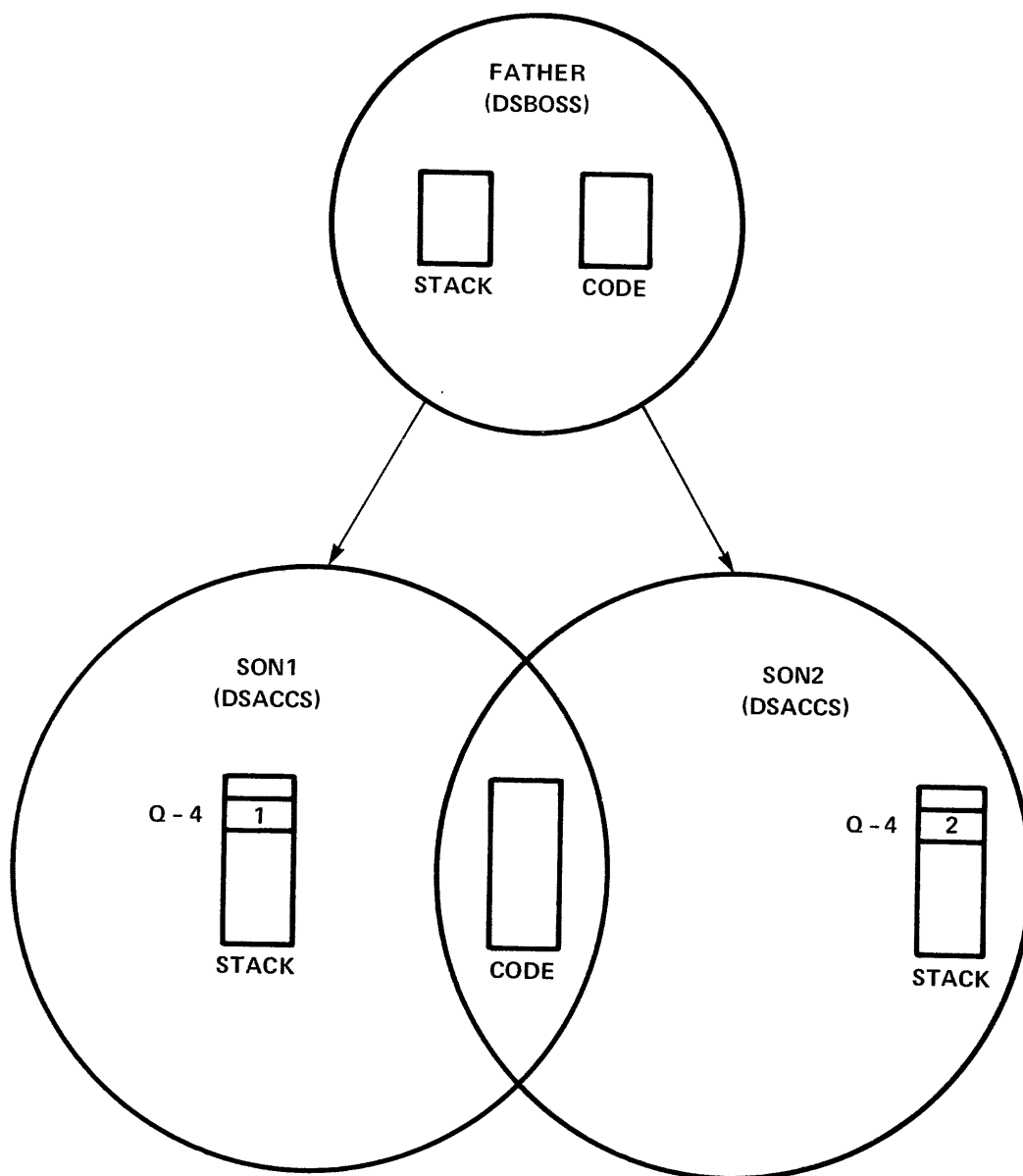
All other parameters are omitted from the CREATE intrinsic call.

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

*formaldesignator*

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.

- foptions*                    %405, which specifies the following:
- a. Old file.
  - b. ASCII.
  - c. Actual file designator is the same as the formal file designator.
  - d. Fixed-length records.
  - e. Carriage-control character expected.
- aoptions*                    4, which specifies input/output access.
- reclsize*                    36, specifying 36 words.
- device*                     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 GO 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, disc storage for a new segment is obtained by calling the GETDSEG intrinsic. Sufficient virtual space is allocated by the system to accommodate the original length of the data segment. This virtual space usually is allocated in increments of 512 words (depending on how the system is configured). For example, creation of a data segment with a length of 600 words would result in two increments of 512 words being allocated for the data segment space, thus resulting in 1024 words.

Once disc storage is obtained, you can use the ALTDSEG intrinsic to reduce the storage required by the segment when it is moved into main memory, then later expand it as needed for more efficient use of memory. In no case, however, can ALTDSEG be used to increase segment size beyond the virtual space originally allocated 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 occurred.

# 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. (\$OPTION PRIVILEGED is used.)
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 and System Reference Manual (Chapter 4) for principle of operation.

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 *options* 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 *options* 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 USLIMIT
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.

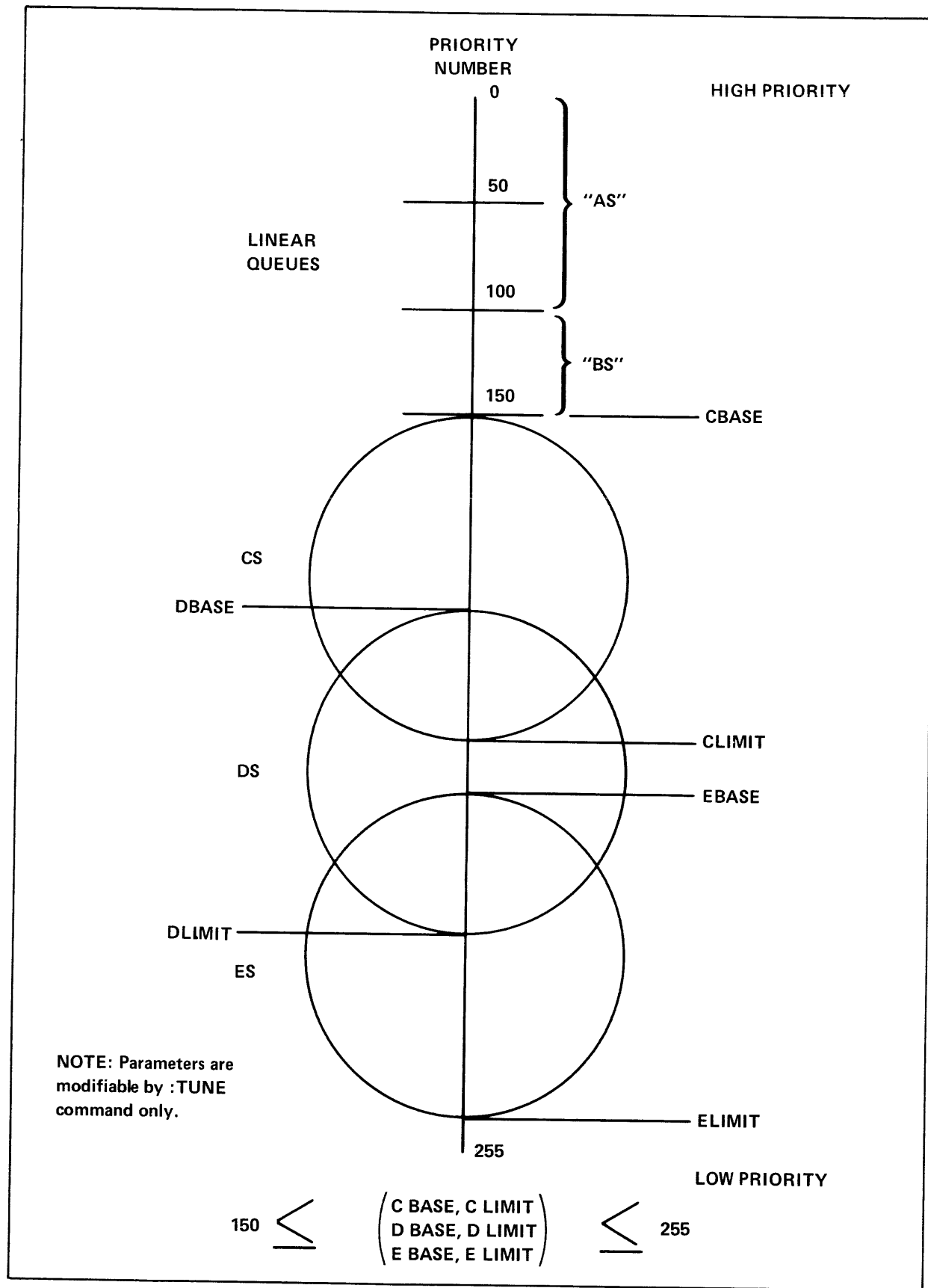


Figure 9-2. Queue Structure

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 GETPRIORITY intrinsics, the process may be set in the master queue at min (100 + rank, 149).
- CS                    Is a circular subqueue generally devoted to interactive sessions. A CS process whose CPU time between priority changes exceeds the "average short transaction" will be lowered in priority, but not below the C Subqueue Priority Limit, called CLIMIT, which may be set by the :TUNE command. (See Section II of the System Manager/System Supervisor Reference Manual, 30000-90014.)
- Scheduling Type:     Circular  
                      Priority Range:       150-CLIMIT
- DS                    Is a circular subqueue generally devoted to batch jobs. A DS process whose CPU time between priority changes exceeds the backgroundquantum will be lowered in priority, but not below the D Subqueue Priority Limit, or DLIMIT, which may be set by the :TUNE command. (See Section II of the System Manager/System Supervisor Reference Manual, 30000-90014.)
- Scheduling Type:     Circular  
                      Priority Range:       150-DLIMIT
- ES                    Is a circular subqueue generally used for so-called "idle" processes. When an ES process's CPU time between priority changes exceeds the background-quantum, its priority is reduced, but not below ELIMIT. Such a process will have a minimal impact on the performance of processes in other subqueues. (See Section II of the System Manager/System Supervisor Reference Manual, 30000-90014.)
- Scheduling Type:     Circular  
                      Priority Range:       150-ELIMIT

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 values of the starting priority (BASE) and priority limits (LIMIT) for each queue, as well as the average short transaction limit and backgroundquantum via the :TUNE command.



A CS process is given a priority of CBASE when it begins. (See Figure 9-2). When a process stops (for disc I/O, terminal I/O, preemption, etc.), its new priority is determined so that it may be re-queued for the CPU. If the process has completed a transaction, (a transaction is defined as the time between terminal reads), the priority becomes CBASE. The value of an “average short transaction” is then recalculated. If the CS process has not completed a transaction, and if the process has exceeded the average short transaction filter value since its priority was last reduced, the priority is decreased, but will not be worse than CLIMIT.

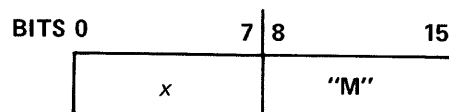
DS and ES processes begin at DBASE and EBASE respectively, and are rescheduled according to the same criteria as used for CS processes, with the exception that a fixed value (the value of max which has been specified for the subqueue) is used in place of the average short transaction value, which is used for CS processes only.

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.

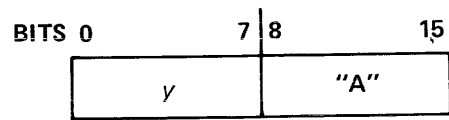
Priorities are assigned to processes through the *priorityclass* parameter of the CREATE and/or GETPRIORITY intrinsic. Because users with the Privileged Mode Capability can schedule processes within the master queue, the *priorityclass* parameter can take on the following values:

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.

# ACCESSING AND ALTERING FILES

SECTION

X

By using MPE file system intrinsics, you can perform the following operations on files:

- Open files with FOPEN. See page 10-28.
- Request access and status information about a file with FGETINFO. See page 10-70.
- Request file error information with FCHECK. See page 10-74.
- Read records (or a portion of a record) from a file with FREAD (see page 10-48) and FREADDIR (see page 10-52).
- Write a record (or a portion of a record) to a file with FWRITE (see page 10-51) and FWRITEDIR (see page 10-55).
- Move a specific record from a file into a buffer preparatory to reading the record to the stack with FREADSEEK. See page 10-55.
- Initiate completion operations for an I/O operation with the IOWAIT intrinsic. See page 10-63.
- Write a file label on a magnetic tape file. See page 10-93.
- Read a user-defined label from a disc file with FREADLABEL. See page 10-70.
- Write a user-defined label onto a disc file with FWRITELABEL. See page 10-66.
- Update a record on a disc file with FUPDATE. See page 10-60.
- Space forward or backward on a disc or tape file with FSPACE. See page 10-85.
- Reset the logical record pointer to any logical record in a fixed-length record disc file with FPOINT. See page 10-96.
- Perform control operations on a file (or the device on which the file resides) with FCONTROL. See page 10-95.
- Activate and deactivate access mode options with FSETMODE. See page 10-96.
- Rename an open disc file with FRENAME. See page 10-45.
- Determine if an input file and a list file are *interactive* or *duplicative* with FRELATE. See page 10-97.
- Coordinate access to shared files with FLOCK and FUNLOCK intrinsics (see pages 10-58 and 10-60).
- Close a file with FCLOSE. See page 10-40.

## 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 from that device. Similarly, output from an executing process is transmitted 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. In most cases, access to 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 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 *aoption* 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 *aoption* is specified, the remaining information is continued on the next line, also left-justified.

Files of fixed or undefined length permit sequential or direct access. Variable length files permit sequential access only in buffered mode, and direct access only in NOBUF.

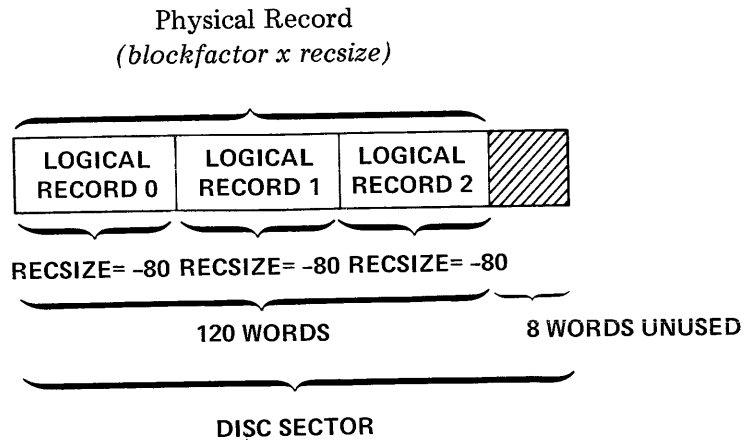
MPE manages each file on disc as a set of *extents*. Each extent is an integral number of contiguously-located disc sectors. All extents (except possibly the last) are of equal size. When a file is opened, the first extent (containing at least one sector for file label information) is allocated immediately. 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.

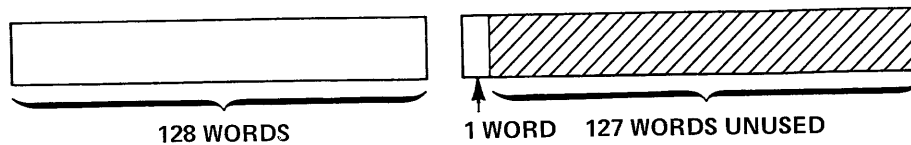
## RECORD FORMATS

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, *resize* 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 *resize* 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 (*resize* x *blockfactor*) 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 and so the number of logical records per physical record may vary. Therefore, when the file is created, the block size is determined by multiplying the blockfactor by the *resize* parameter specified in FOPEN plus one, and adding one word (reserved for file-system use).

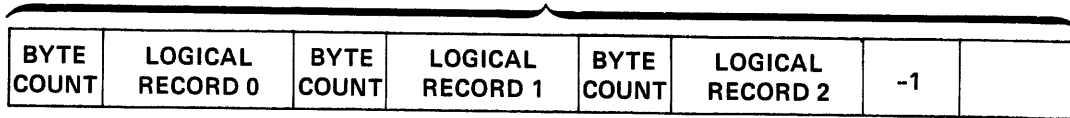
Because a fixed block factor has no meaning for a file of variable length records, once the calculation is done the block factor is reset to 1. Maximum logical *resize* is then set to actual blocksize minus two words.

$$\text{Actual Blocksize} = (\text{blockfactor} \times (\text{resize} + 1)) + 1$$

(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. (Logical records are not split across block boundaries.) The block format is:

$$\text{Physical Record} \\ [(\text{blockfactor} \times (\text{recsize} + 1) + 1)]$$

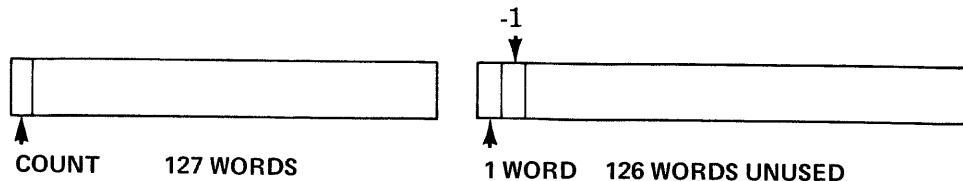


Unless block size is computed such that

$$(\text{actual block size}) \bmod 128 = 0,$$

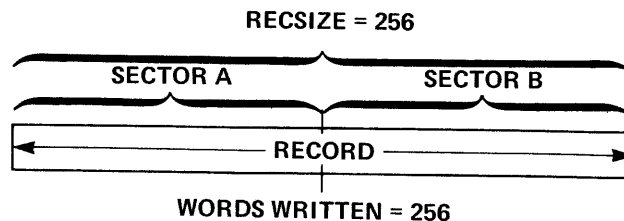
some disc space will be wasted.

For example, if *recsize* = 128 words and *blockfactor* = 1, 126 words of disc space will be wasted as follows:

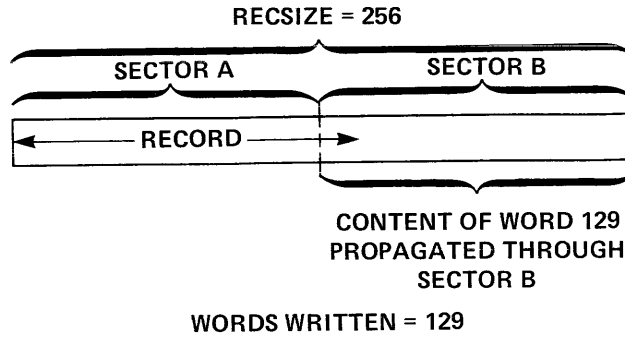


For *undefined-length records*, *physical records* and *logical records* are *synonymous* — that is, Physical record A is the same as logical record A. For records of this type, the *recsize* parameter specified by the user denotes the size of the longest record to be transferred. The format of undefined records written to disc, with respect to the disc sectors occupied, can be illustrated by three cases in which the user-specified *recsize* is 256.

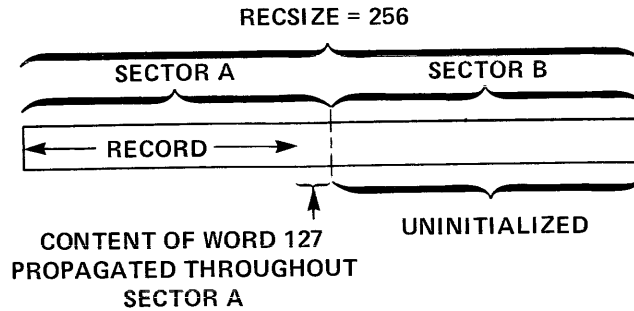
*Case 1:* You write a record 256 words long. The full record completely fills two disc sectors.



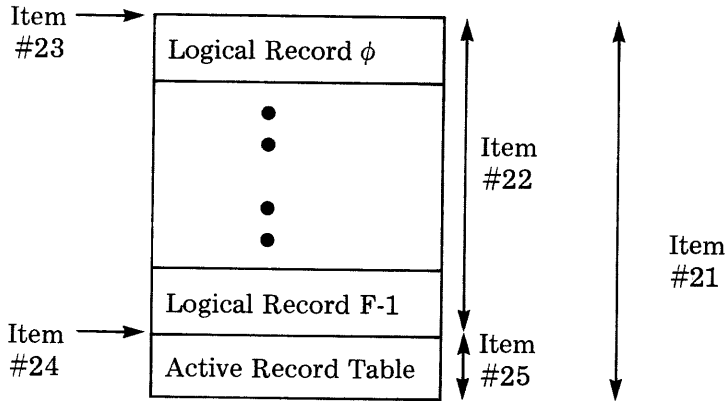
*Case 2:* You write a record 129 words long. The record written occupies all of sector A and the first word of sector B; the last word written is propagated throughout the remainder of sector B. (The rule is: if (*reclength*) modulo 128 is not zero, then the last word written is propagated through the 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).)



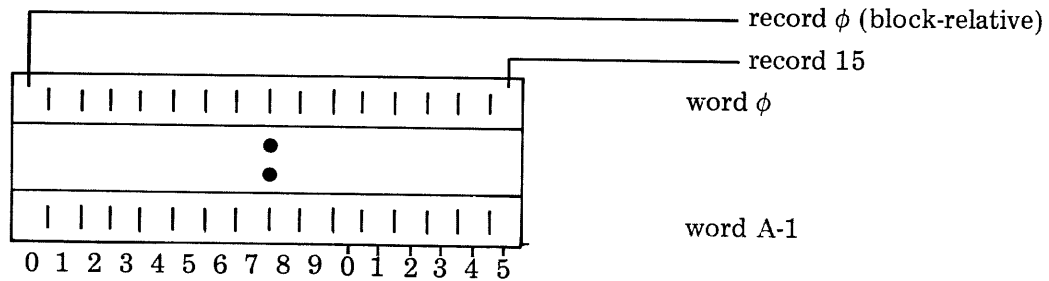
# RELATIVE I/O BLOCK FORMAT



FFILEINFO Item Numbers (see Figure 2-1A)

- Item 21 — Physical Block Size
- 22 — Data Block Size
- 23 — Offset to Data in Blocks
- 24 — Offset to Active Record Table within the block
- 25 — Size of Active Record Table

Active Record Table



$$A = \text{active-record table size in words} = \left\lceil \frac{F}{16} \right\rceil$$

F = blocking factor (number of records per block)

R = index of desired record, modulo F

W = index of word for desired record = R/16

P = index of bit for desired record = R mod 16

bit = 0: inactive record

= 1: active record



## FILE DEVICE RELATIONSHIPS

Devices required by files are allocated 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. In this case, no console operator intervention is required.

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. This is the name supplied by the program to the FOPEN intrinsic.) At the time a file is FOPENed, this formal file designator is used to determine the *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*.

## RELATIVE I/O

In addition to the conventional direct and serial access, MPE offers Relative I/O access. RIO is intended for use primarily by COBOL II programs; however you can access these files by programs written in any language.

RIO is a direct access method that permits individual file records to be deactivated. These inactive records retain their space and position within the file, i.e., their relative position.

RIO files may be accessed in two ways; RIO access and non-RIO access. RIO access ignores the inactive records when the file is read serially using the FREAD intrinsic; however such records can be read by direct access using FREADDIR. They may be overwritten both serially and directly using FWRITE, FWRITEDIR or FUPDATE. With RIO access the internal structure of RIO blocks is transparent.

Non-RIO access is provided to facilitate the replication of RIO files. This method requires knowledge of the internal structure of the file and is intended primarily for system use. Non-RIO access is enabled by specifying NOBUF I/O when the file is opened.

Most of the existing MPE file intrinsics access RIO files in the same manner as they would access other MPE files. There are exceptions, however. Some of these are:

FREAD inputs an active record, not necessarily the next physical record.

FREADDIR inputs the specified record regardless of its activity state; a warning will result if the record is inactive

FDELETE deactivates the next record, not the last-accessed record as defined by COBOL II.

The actual size of a block will be slightly larger than the size specified when the file was built. The value returned by FGETINFO for *blksize* will depend on how the RIO file is being accessed.

The amount of space available for user labels also may be slightly larger; this depends on the size of an extent which itself depends on the block size.

## 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 `$STDIN` file should not contain a colon in column 1, since this indicates the end-of-data. (When data is to be delimited, this should be done through the `:EOD` command, which performs no other function.) Once `:` or `:EOD` is detected on `$STDIN`, that end-of-file condition is considered permanent for the life of the process. No further reading may be done from `$STDIN/$STDINX` by that process. [If `:EOF`: (hardware EOF command) is encountered, the end of file condition applies to the entire session which will then have to terminate.]

**Actual File Designator****Device/File Referenced**

\$STDINX	Equivalent to \$STDIN, except that records 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, :EOD and :EOF: are exceptions that always indicate the end-of-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 disc file that is always assumed to be new, and is always closed in such a way that it can be automatically passed to any succeeding MPE command/job step within the same job/session, which will reference it by the file name \$OLDPASS. Only one \$OLDPASS file can exist in the job/session at any one time. (When \$NEWPASS is closed, it is automatically changed to \$OLDPASS, and any previous file named \$OLDPASS in the job/session is deleted.)
-----------	---

<i>filereference</i>	Requests that a new file be created with this name, residing on disc, or some other device. Unless other action is taken, a new disc file will be deleted on termination of the creating program. If closed as a job/session temporary file, as shown later in this section, such a file 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 resulting from the last close of a \$NEWPASS file.
<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 records containing : in column 1 allowed, (exclusive of :EOD, :EOJ, :DATA, etc.)
\$OLDPASS	The last \$NEWPASS file closed, (considered a "passed file").
\$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 (via :FILE command).
<i>filereference</i>	A file name, and perhaps account and group names and a lockword. When file name, group name and account name are supplied, the name is said to be a <i>fully qualified</i> file name.

### Output Set

\$STDLIST	The job/session listing device.
\$OLDPASS	The last \$NEWPASS file closed, (considered a "passed file").
\$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 10-1.

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			
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			
FOPEN for Input/Output	SHR	Requested Access Granted	Input Only Granted	Requested Access Granted	Input Only Granted	Requested Access Granted	Input Only Granted	Input Granted	Input Granted	Error 91
	EAR	Requested Access Granted	Input Only Granted	Error 90	Error 90	Error 90	Error 90			
:RUN,CREATE		Requested Access Granted		Error Message		Error Message		Requested Access Granted	Only if Loaded	Error Message
:STORE		Requested Access Granted		Error Message		Error Message		Requested Access Granted	Error Message	Error Message
:RESTORE		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 10-1. Actions Resulting from Multiple Access of Files

Figure 10-1. Actions Resulting from Multiple Access of Files

When a new file is opened but not yet closed, it is always local to the process. At this time, the file name assigned by you need not be unique. But if the program tries to save the file permanent or job temporary (via FCLOSE), MPE determines whether another file with the same designator name exists in the domain in which you are trying to save that file (permanent or job temporary). If a name conflict occurs, a CCL condition code is returned to the user process from FCLOSE 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. The fact that a duplicate file name is detected at FCLOSE (not FOPEN) time is important for many applications.

#### NOTE

All intrinsics discussed in this section, with the exception of FOPEN, FGETINFO, FFILEINFO and FRENAME, can be called 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. Privileged mode is required to enter split-stack mode; once in split-stack mode, you need not remain in privileged mode to call File System intrinsics.

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 10-28). 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. Whenever a process is run, MPE calls FOPEN twice to open \$STDIN and \$STDLIST for that process before any of the user code is executed. Thus there are 253 file numbers available to the user process. However, no assumption should *ever* be made concerning the allocation order of these file numbers.

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 access where the multi-access option was not requested or not permitted at FOPEN time.

In opening a file, FOPEN establishes a communication link between the file and your program by

- Determining the computer system on which the file resides.
- 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 five 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. FOPEN overrides of incompatible options.
  3. 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.
  4. The parameter list of this FOPEN intrinsic call.
  5. System default values provided by MPE (when values are not obtainable from the above three sources).

When information in one of these five 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/FFILEINFO 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 four 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/FFILEINFO intrinsic should be used to determine the logical device number assigned to an opened file. The subsequent FOPEN which supplies this logical device number should insure that no existing file equation overriding the device number is accidentally picked up.

When a job opens a new file on a magnetic tape unit, operator intervention is always required; the operator must make the tape available.

### **SPECIAL CONSIDERATIONS FOR SHARED FILES**

When a file is being shared among two or more processes, or within the same process, and is being written to by one or more of them, care must be taken to ensure that the processes are appropriately interlocked. For example, if Process A is trying to read a particular record of the file, and at that time Process B should execute and try to write that record, the results are not predictable. Process A may see the old record, the new record, or hash consisting of parts of both. If buffering is being done, please bear in mind that an output request (FWRITE) will not cause physical I/O to occur until a block is filled, which typically will contain several records. A process trying to read such a file could, for example, read past the last record of the file which has been written on the disc because the end-of-file pointer is not kept in the file but is kept in core where it can be updated quickly as writes occur. This interlocking is provided by the intrinsics FLOCK and FUNLOCK, which use a Resource Identification Number (RIN) as a flag to interlock multiple accessors.

For processes within a job/session sharing a file in multi-access mode, the use of a local RIN is recommended instead of FLOCK and FUNLOCK. That recommendation does not apply if the file is simultaneously being accessed from another job or session.

In the simple case of a file shared between a writer process and a reader process, where the writer is merely adding records to the file, the writer calls FLOCK prior to writing each record and FUNLOCK after writing. The reader calls FLOCK prior to reading record, and FUNLOCK after reading. If the writing process should execute while the reader is in the middle of a read, the writer will be impeded on its FLOCK call until the reader signifies that it is done by calling FUNLOCK. Similarly, if the reader should execute while the writer is performing a write, the reader will be impeded on its FLOCK call until the writer calls FUNLOCK. FUNLOCK ensures that all buffers are posted on the disc so that reading processes can see all the data.

Protection offered by FLOCK and FUNLOCK *depends on cooperation among all processes accessing the file*. If one process does not use FLOCK, or uses it improperly, problems will arise.

More complicated cases arise when a file has two or more writing processes, or when the write consists of writing more than one record at a time. If, for example, it should be necessary to write pairs of records, with read prohibited until both records of the pair are written, the writing process can call FLOCK before writing the first record of the pair, and FUNLOCK after writing the second. This procedure also can be used if the records are to be written in different files; one of the files is used as a "sentinel" file and the processes FLOCK and FUNLOCK this file as required.

## PRIVATE VOLUMES SUBSYSTEM

Users with the Volume Set Usage (UV) capability can maintain files on private disc volumes. These private volumes consist of removable disc packs, which, when mounted on a disc drive, can be accessed by MPE through the MPE Private Volumes Subsystem.

Individual removable volumes can be combined to form logical units in the form of volume sets or volume classes.

See the *MPE Commands Reference Manual* for a further discussion of the Private Volumes Subsystem.

Whether disc files created by you will be assigned to a private disc volume set or the system volume set depends on how the file group for your account was established by the System Manager. The System Manager can establish your file group so that disc files created by you will be stored on a private home volume set or on the system volume set. In either case, you do not need to be concerned with, or even aware of, where disc files created and opened by you will be stored.

If your file group has been structured to use the Private Volumes Subsystem, then when you create a new disc file with the :BUILD command (see the *MPE Commands Reference Manual*) or the FOPEN intrinsic, MPE checks to determine if your home volume set is mounted. If your home volume set is not mounted, MPE asks the Console Operator to mount it. The only indication to you of this action is that your program will suspend, if your home volume set is not mounted, until it is mounted by the Console Operator. Similarly, when you close and save a disc file with the FCLOSE intrinsic, it is automatically stored on your home volume set or the system volume set, depending on how your file group was established, again with no action necessary from you.

## 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 10-2 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 10-2, 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.

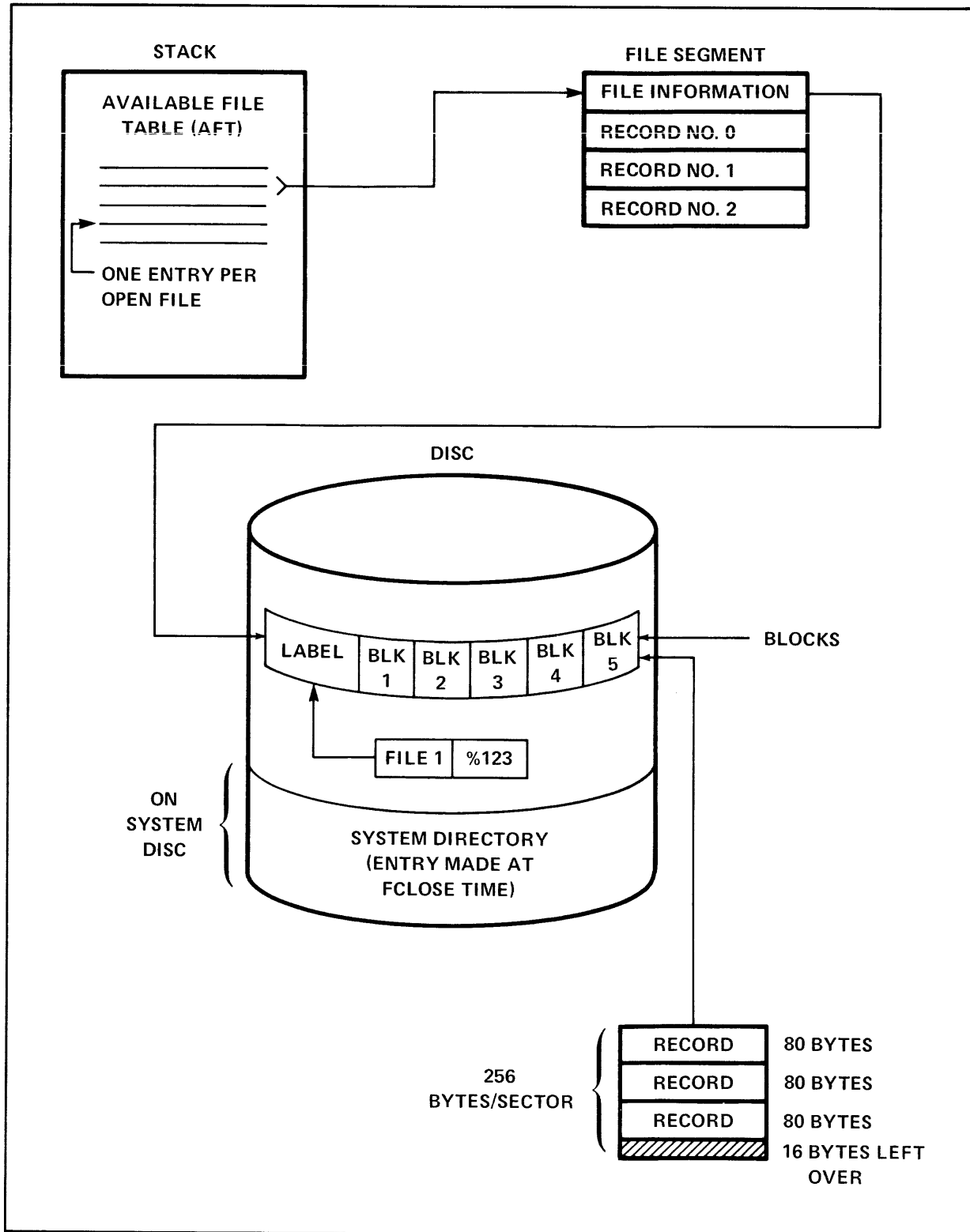


Figure 10-2. File Access Interface for New Disc Files

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 allocates initial space for this file, (the number of sectors allocated depends on the parameters specified in the FOPEN call), by deallocating free space from the table and writing a file label in the first sector of the newly allocated space. 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 10-2, 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 10-2, 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 10-3 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 10-4 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

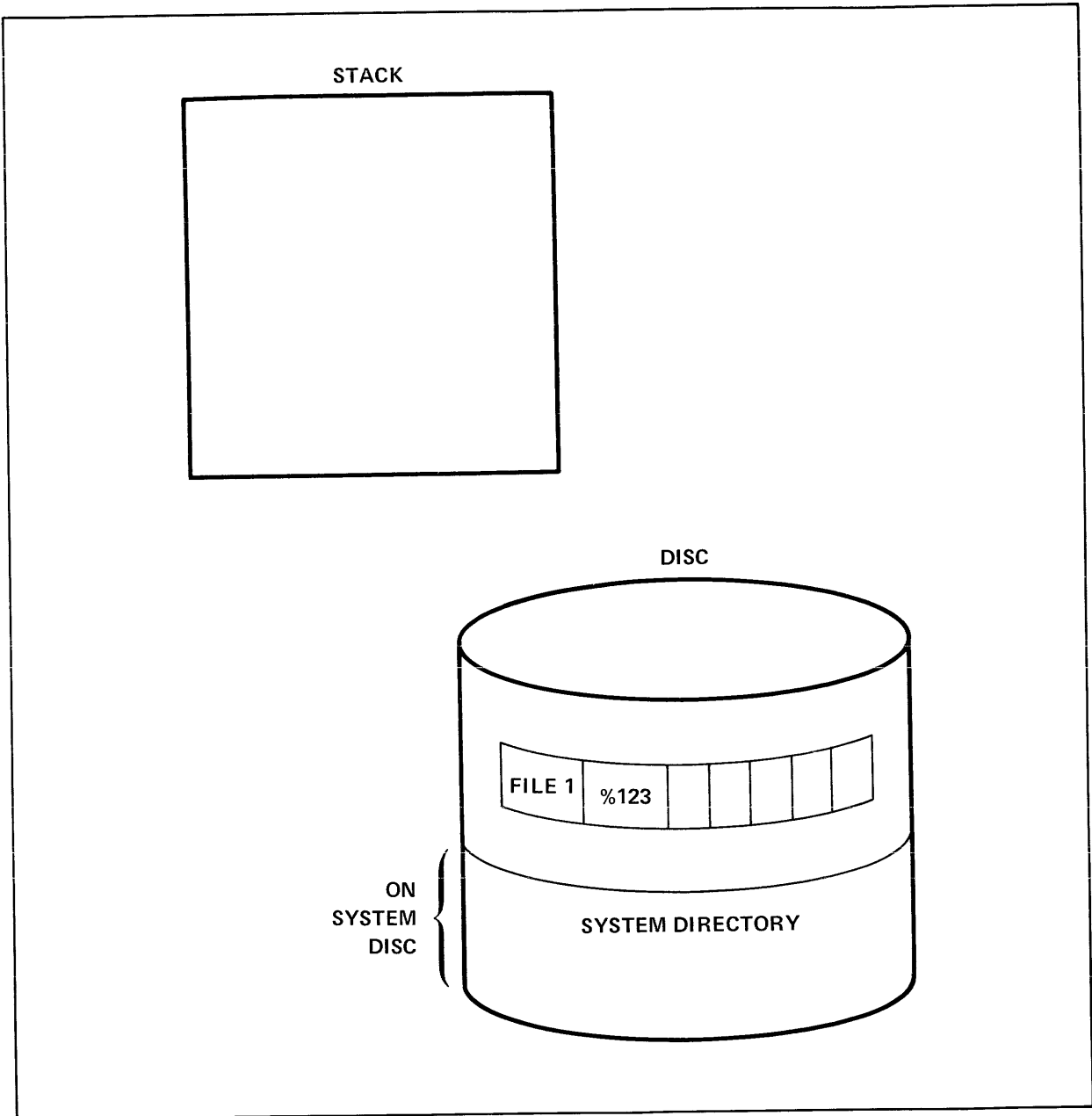


Figure 10-3 File Name and Sector Address Storage

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)

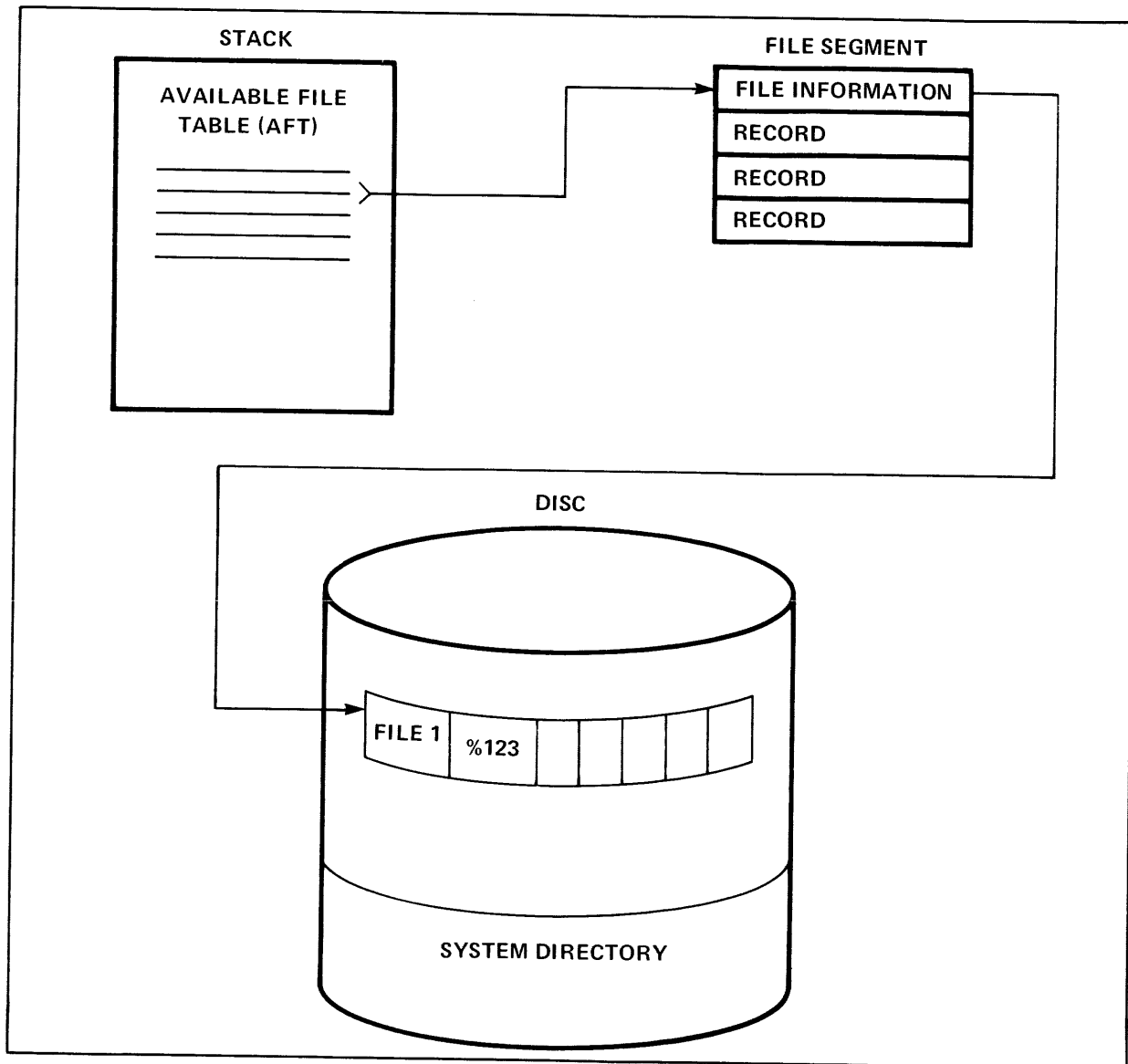


Figure 10-4. File Access Interface for Old Disc Files

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 10-1.

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, most differences between a spooled and a non-spooled device file are insignificant to the program.

One exception, however, may affect applications which write very large reports. Regardless of the spooled device, there is a limit as to how much disc space an individual spoolfile can have. In some cases, it is possible to exhaust this maximum space allowed. As an example, an application would not normally expect to encounter an end-of-file condition when writing to a line printer. Yet this may actually happen if the line printer is spooled and the report being written is large enough to have reached the spoolfile's limit. Therefore, applications should check for end-of-file after every write. When an end-of-file condition is detected, the output file should be closed and a new one opened.

#### NOTE

The maximum size of a spoolfile in sectors is 32\* configured extend size for spoolfiles. (The maximum number of extents is 32.) A configured extent size of 384 sectors means the spoolfile has room for approximately 25,000 lines.

Table 10-1. 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

Serial Disc Drive

No restriction

OUTPUT ONLY (SERIAL)

Line Printer/Card Punch/Paper Tape Punch/Plotter

If Line Printer, 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 10-2. 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
Hardwired Serial Interface	23	NEW or OLD
Synchronous Single-Line Controller	26	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

An alternative to generating one large output spoolfile is to periodically close the output file and open a new one. A large report program might start a new output file every 200 pages. While this technique requires gathering several files for the complete report, it has the advantage of allowing the first portion of the report to print while the program is still running.

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 10-2 shows the manner in which devices are classified. Included in table 10-2 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 10-5 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.

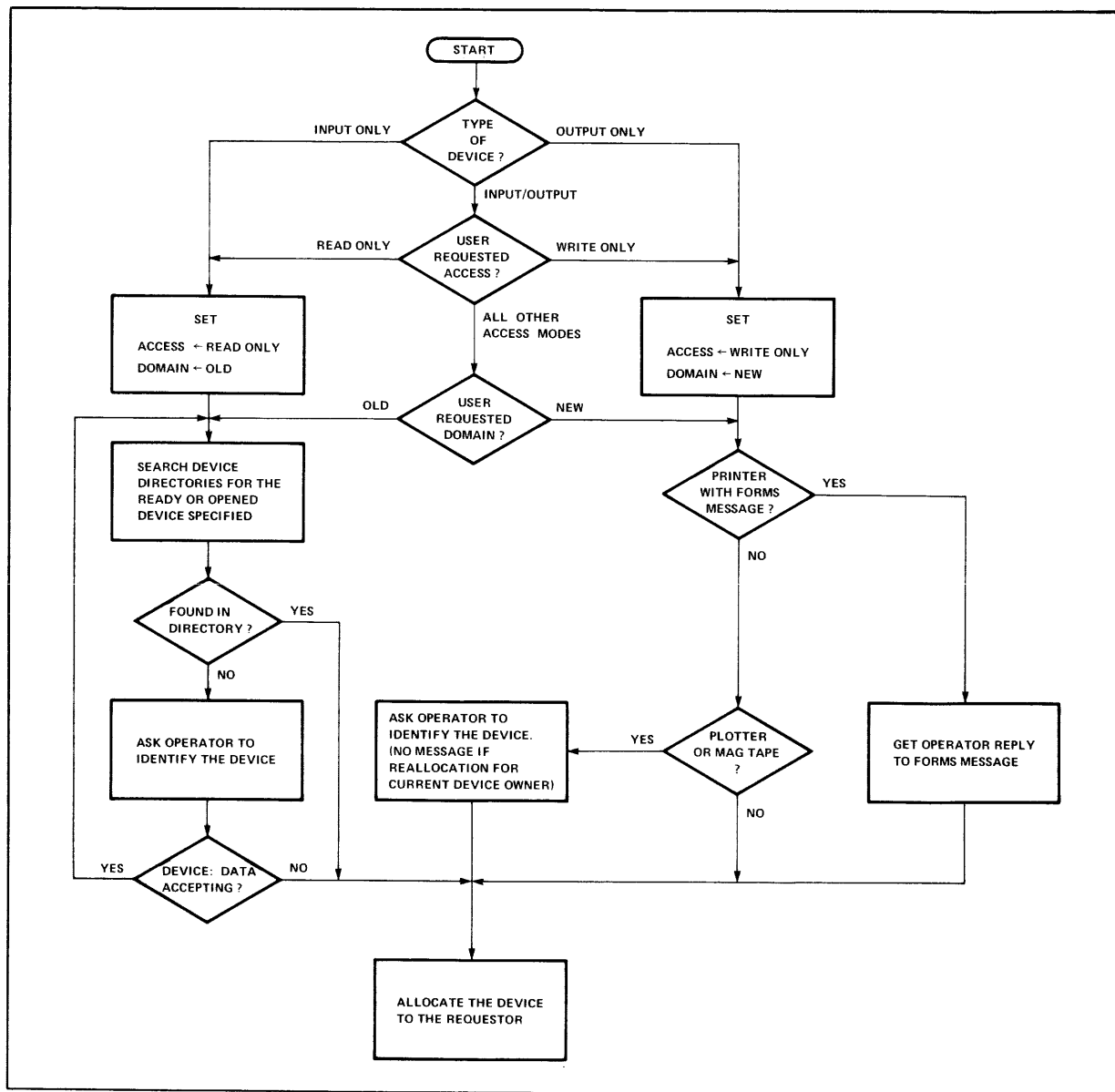


Figure 10-5. Device Allocation Flowchart

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 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 in the same process tree as the process which has the file open. This is accomplished by referencing a logical device number, not a device classname. 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. Attempts to use this technique with a non-spooled printer can result in inter-mixing of output data.

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 10-6 contains an SPL program which opens two files: a card reader file and a new disc file.

The second FOPEN call in figure 10-6.

```
OUT:=FOPEN(OUTPUT,%4,%101,128);
```

opens the new disc file. The parameters specified are

*formal designator* 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 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
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 10-6. 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.

*aoptions*

%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.

*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, the system file directory should be searched. Bits (14:2) = 01.

ASCII/Binary: ASCII. Bit (13:1) = 1.

*options* %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 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA1(0:7):="DATAONE "
00004000 00005 1 ARRAY BUFFER(0:127)
00005000 00005 1 INTEGER DFILE1, LGTH, DUMMY, IN, LIST
00006000 00005 1
00007000 00005 1 INTRINSIC FOPEN, FREAD, FUPDATE, FLOCK, FUNLOCK, FCLOSE,
00008000 00005 1 PRINT'FILE'INFO, QUIT, FWRITE, FREAD
00009000 00005 1
00010000 00005 1 PROCEDURE FILERERR(FILENO, QUITNO)
00011000 00000 1 VALUE QUITNO
00012000 00000 1 INTEGER FILENO, QUITNO
00013000 00000 1 BEGIN
00014000 00000 2 PRINT'FILE'INFO(FILENO)
00015000 00002 2 QUIT(QUITNO)
00016000 00004 2 END
00017000 00000 1
00018000 00000 1 <<END OF DECLARATIONS>>
00019000 00000 1
00020000 00000 1 DFILE1:=FOPEN(DATA1,%5,%345,128) <<OLD DISC FILE>>
00021000 00011 1 IF < THEN FILERERR(DFILE1,1) <<CHECK FOR ERROR>>
00022000 00015 1
00023000 00015 1 IN:=FOPEN(,%244) <<$STDIN>>
00024000 00024 1 IF < THEN FILERERR(IN,2) <<CHECK FOR ERROR>>
00025000 00030 1
00026000 00030 1 LIST:=FOPEN(,%614,%1) <<$STDLIST>>
00027000 00040 1 IF < THEN FILERERR(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 FILERERR(DFILE1,4) <<CHECK FOR ERROR>>
00032000 00053 1
00033000 00053 1 LGTH:=FREAD(DFILE1,BUFFER,128) <<GET EMPLOYEE RECD>>
00034000 00061 1 IF < THEN FILERERR(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 FILERERR(LIST,6) <<CHECK FOR ERROR>>
00039000 00101 1
00040000 00101 1 DUMMY:=FREAD(IN,BUFFER(30),5) <<EMPLOYEE NUMBER>>
00041000 00110 1 IF < THEN FILERERR(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 FILERERR(DFILE1,8) <<CHECK FOR ERROR>>
00046000 00125 1
00047000 00125 1 FUNLOCK(DFILE1) <<ALLOW OTHER ACCESS>>
00048000 00127 1 IF <> THEN FILERERR(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 FILERERR(DFILE1,10) <<CHECK FOR ERROR>>
00055000 00146 1
00056000 00146 1 FCLOSE(DFILE1,0,0) <<DISP-NO CHANGE>>
00057000 00151 1 IF < THEN FILERERR(DFILE1,11) <<CHECK FOR ERROR>>
00058000 00155 1 END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00204
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:17

```

Figure 10-7. Opening an Old Disc File

All other parameters are omitted in the FOPEN intrinsic call. Note that for existing files FOPEN will return a lockword violation (FSERR92 via FCHECK) if the lockword is not included in the *formaldesignator* parameter.

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 procedure 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. The system Job Control Word "JCW" is set to FATAL1 in this example.

## FOREIGN DISC FACILITY

The Foreign Disc Facility (FDF) allows you to use the file system to access and alter disc packs and flexible diskettes that do not have standard HP 3000 file system disc label formats. When mounted, a disc volume with an unrecognizable disc label is assumed to be a foreign disc.

Discs and diskettes must be physically compatible with HP hardware. The IBM 3741 format diskettes (64 words per sector), for example, are compatible.

When using the FOPEN intrinsic to open a foreign disc file, the reconfig is forced to 128 words (IBM diskettes are forced to 64 words). The file system will treat disc sectors as file records, thereby allowing you to manipulate the foreign file as though it was an MPE created file.

In addition to FOPEN, several other intrinsics have been modified to accommodate foreign discs. They are: FCLOSE, FREAD, FWRITE, FWRITEDIR, FREADDIR, FGETINFO, FFILEINFO, and FCHECK.

## OPENING A FILE ON A DEVICE OTHER THAN DISC

Figure 10-8 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 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 10-8, 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.

*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, system file domain. Bits (14:2) = 01.  
 ASCII/Binary: ASCII. Bit (13:1) = 1.

*aoptions* 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.

```

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
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. FRRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:44

```

Figure 10-8. Opening a File on a Device Other Than Disc

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 10-9 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.

```

00001000 00000 0 $CONTROL USLIMIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA(0:7):="DATAONE ";
00004000 00005 1 ARRAY BUFFER(0:127);
00005000 00005 1 INTEGER DFILE1, LGTH, DUMMY, IN, LIST;
00006000 00005 1
00007000 00005 1 INTRINSIC FOPEN, FREAD, FUPDATE, FLOCK, FUNLOCK, FCLOSE,
00008000 00005 1 PRINT'FILE'INFO, QUIT, FWRITE, FREAD;
00009000 00005 1
00010000 00005 1 PROCEDURE FILERROR(FILENO, QUITNO);
00011000 00000 1 VALUE QUITNO;
00012000 00000 1 INTEGER FILENO, QUITNO;
00013000 00000 1 BEGIN
00014000 00000 2 PRINT'FILE'INFO(FILENO);
00015000 00002 2 QUIT(QUITNO);
00016000 00004 2 END;
00017000 00000 1
00018000 00000 1 <<END OF DECLARATIONS>>
00019000 00000 1
00020000 00000 1 DFILE1:=FOPEN(DATA1,%5,%345,128); <<OLD DISC FILE>>
00021000 00011 1 IF < THEN FILERROR(DFILE1,1); <<CHECK FOR ERROR>>
00022000 00015 1
00023000 00015 1 IN:=FOPEN(,%244); <<$STDIN>>
00024000 00024 1 IF < THEN FILERROR(IN,2); <<CHECK FOR ERROR>>
00025000 00030 1
00026000 00030 1 LIST:=FOPEN(,%614,%1); <<$STDLIST>>
00027000 00040 1 IF < THEN FILERROR(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 FILERROR(DFILE1,4); <<CHECK FOR ERROR>>
00032000 00053 1
00033000 00053 1 LGTH:=FREAD(DFILE1,BUFFER,128); <<GET EMPLOYEE RECD>>
00034000 00061 1 IF < THEN FILERROR(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 FILERROR(LIST,6); <<CHECK FOR ERROR>>
00039000 00101 1
00040000 00101 1 DUMMY:=FREAD(IN,BUFFER(30),5); <<EMPLOYEE NUMBER>>
00041000 00110 1 IF < THEN FILERROR(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 FILERROR(DFILE1,8); <<CHECK FOR ERROR>>
00046000 00125 1
00047000 00125 1 FUNLOCK(DFILE1); <<ALLOW OTHER ACCESS>>
00048000 00127 1 IF <> THEN FILERROR(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 FILERROR(DFILE1,10); <<CHECK FOR ERROR>>
00055000 00146 1
00056000 00146 1 FCLOSE(DFILE1,0,0); <<DISP-NO CHANGE>>
00057000 00151 1 IF < THEN FILERROR(DFILE1,11); <<CHECK FOR ERROR>>
00058000 00155 1 END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00204
NO. FRRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:17

```

Figure 10-9. Opening \$STDIN and \$STDLIST

*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.

*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 FILERROR(IN,2);
```

checks the condition code. If the condition code is CCL, signifying that the FOPEN request was denied, the error-check procedure FILERROR is called.

The FILERROR 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 10-9, the statement

```
LIST:=FOPEN(,%614,%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

*formaldesignator*

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.

*aoptions*

%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 FILERROR(LIST,3);
```

checks the condition code. If the condition code is CCL, signifying that the FOPEN request was denied, the error-check procedure FILERROR is called.

The FILERROR 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.



```

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-1; <<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=0:00:04; ELAPSED TIME=0:00:59

```

Figure 10-10. Closing a New File as a Temporary File

## 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 10-10 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 unlabeled 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.

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 10-11 contains an FCLOSE intrinsic call that closes a new file as a permanent file.

The FCLOSE intrinsic call

```
FCLOSE(OUT,%11,0);
```

```

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
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 10-11. Closing a New File as a Permanent File

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 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), (fixed and undefined length files only).

*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 exclusively-opened 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. If the file is a permanent file, you must be the creator.

When the FCLOSE intrinsic is called in figure 10-12, 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 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.

```

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. ERRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:04; ELAPSED TIME=0:00:58

```

Figure 10-12. FRENAME Intrinsic Example

## 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: FILE ACCESSED EXCLUSIVELY (FSERR 91)
```

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 10-13 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 10-13) 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.



```

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):="DATAOVE ";
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
00012000 00000 1 PROCEDURE FILEERR(FILENO,QUITNO);
00013000 00000 1 VALUE QUITNO;
00014000 00000 1 INTEGER FILENO,QUITNO;
00015000 00000 1 BEGIN
00016000 00002 2 PRINT*FILE*INFO(FILENO);
00017000 00004 2 QUIT(QUITNO);
00018000 00000 1 END;
00019000 00000 1
00020000 00000 1 << END OF DECLARATIONS >>
00021000 00000 1 IN:=FOPEN(INPUT,%5,,40,DEV); <<CARD READER>>
00022000 00012 1 IF < THEN FILEERR(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 FILEERR(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 FILEERR(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 FILEERR(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 FILEERR(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 10-13. Error-Check Procedure Example

## 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.

If RIO access is used, FREAD will input the next active record, automatically ignoring de-activated records. The fact that inactive records may have been ignored is transparent to the caller.

If an RIO file is accessed using the non-RIO method, FREAD will input from the next block.

The program shown in figure 10-14 reads a card file. The FREAD statement

```
LGTH:=FREAD(IN,BUFFER,40);
```

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.

The FREAD intrinsic operates in the usual manner to read foreign discs. However, IBM diskettes number sectors starting with one rather than zero, and the diskette driver adds one to all sector addresses for IBM diskettes. Therefore, you specify record number zero to read sector number one on an IBM diskette.

```

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
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 10-14. FREAD and FWRITE Intrinsics Example

## 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 10-14.

```
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.

The FWRITE intrinsic operates in the usual manner to write to foreign discs. However, IBM diskettes number sectors starting with one rather than zero, and the diskette driver adds one to all sector addresses for IBM diskettes. Therefore, you specify record number zero to write to sector number one on an IBM diskette.

## 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.

The FREADDIR intrinsic operates in the usual manner to read foreign discs. However, IBM diskettes number sectors starting with one rather than zero, and the diskette driver adds one to all sector addresses for IBM diskettes. Therefore, you specify record number zero to read sector number one on an IBM diskette.

Figure 10-15 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 0D (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 FILEERROR(DFILE2,3);
```

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 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 file system buffer. Then, when the FREADDIR intrinsic call is issued, the record is transferred from this buffer to the buffer in the stack specified by FREADDIR. The I/O process is enhanced when FREADSEEK is used, if the record to be read can be brought into the buffer before the FREADDIR call is issued. An FREADSEEK call should not immediately be followed by the FREADDIR call; enough time must be allowed for the I/O process to bring in the record from the file.

The LIST'LOOP in figure 10-15 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 an 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 to wait for file access and thus reducing the execution time of the loop.

Note: Can also be used with FPOINT and FREAD.

## WRITING RECORDS INTO A FILE IN DIRECT-ACCESS MODE

To write information into a specific record in a disc file, you can 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 (or physical record number if in NOBUF access mode).

The FWRITEDIR intrinsic call may be issued only for disc files of fixed-length or undefined-length records.

The FWRITEDIR intrinsic operates in the usual manner to write to foreign discs. However, IBM diskettes number sectors starting with one rather than zero, and the driver adds one to all sector addresses for the IBM diskettes. Therefore, you specify record number zero to write to sector number one on an IBM diskette.

Figure 10-16 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 10-68) 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.

```

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(DFILE2,5); <<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
PROCFESSOR TIME=0:00:104; ELAPSED TIME=0:00:158

```

Figure 10-15. FREADDIR and FREADSEEK Intrinsics Example

```

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-1; <<LAST RECD 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=0:00:04; ELAPSED TIME=0:00:59

```

Figure 10-16. FWRITEDIR Intrinsic Example



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.

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 signal that another user should 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. (The MPE RIN mechanism provides a waiting queue facility). 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.

It is important to note that the FLOCK and FUNLOCK intrinsics only provide a means of signaling that the caller wants temporary exclusive use of the file. All processes must cooperate by using the same signaling system to ensure data integrity. If one process fails to call FLOCK or does so improperly, the data may be corrupted and unexpected results may occur. FLOCK locks as RIN, not the file itself.

Because the RIN's used in dynamic file locking are available system wide, 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.

Figure 10-17 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

*filenum*                      Contained in DFILE1, which was assigned the file number of DATAONE when the FOPEN intrinsic opened the file.

*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.

```

00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 BYTE ARRAY DATA(0:7):="DATAONE ";
00004000 00005 1 ARRAY BUFFER(0:127);
00005000 00005 1 INTEGER DFILE1, LGTH, DUMMY, IN, LIST;
00006000 00005 1
00007000 00005 1 INTRINSIC FOPEN, FREAD, FUPDATE, FLOCK, FUNLOCK, FCLOSE,
00008000 00005 1 PRINT 'FILE' INFO, QUIT, FWRITE, FREAD;
00009000 00005 1
00010000 00005 1 PROCEDURE FILEERROR(FILENO, QUITNO);
00011000 00000 1 VALUE QUITNO;
00012000 00000 1 INTEGER FILENO, QUITNO;
00013000 00000 1 BEGIN
00014000 00000 2 PRINT 'FILE' INFO(FILENO);
00015000 00002 2 QUIT(QUITNO);
00016000 00004 2 END;
00017000 00000 1
00018000 00000 1 <<END OF DECLARATIONS>>
00019000 00000 1
00020000 00000 1 DFILE1:=FOPEN(DATA1,%5,%345,128); <<OLD DISC FILE>>
00021000 00011 1 IF < THEN FILEERROR(DFILE1,1); <<CHECK FOR ERROR>>
00022000 00015 1
00023000 00015 1 IN:=FOPEN(,%244); <<$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=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:17

```

Figure 10-17. FLOCK and FUNLOCK Intrinsic Example

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 *aoption* specified in the FOPEN call and must not contain variable-length records.

FUPDATE operates in the usual manner to update a foreign disc file.

Figure 10-18 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.

```

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. FRRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:03; ELAPSED TIME=0:00:17

```

Figure 10-18. FUPDATE Intrinsic Example

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 FILEERROR(DFILE1,9);
```

checks for a “not equal” condition code and, if such is the case, calls the error-check procedure FILEERROR. The procedure FILEERROR 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.

## USING THE IOWAIT INTRINSIC

Figure 10-19 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 *options* specified, the program must be running in privileged mode, and this program is switched to privileged mode with the statement

```
GETPRIVMODE;
```

```

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 10-19. Using the IOWAIT Intrinsic

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

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.

*recsize*                    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 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 10-21) 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 10-21 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 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. User-defined labels can be read from and written to either disc files or labeled magnetic tape files. The tape files must be previously labeled with an ANSI-standard or IBM-standard label. (See Page 10-80 for a discussion of labeled magnetic tape files.)

### WRITING A USER FILE LABEL ON A DISC FILE

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.

In figure 10-20 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 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 FWRITELABEL, then aborts the program's process.

## READING A USER FILE LABEL ON A DISC FILE

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 10-21, 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 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 IO>>
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-1; <<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
PROCFESSOR TIME=0:00:04; ELAPSED TIME=0:00:59

```

Figure 10-20. FWRITELABEL Intrinsic Example (Disc File)

```

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. ERRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:04; ELAPSED TIME=0:00:58

```

Figure 10-21. FREADLABEL Intrinsic Example (Disc File)

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 or the FFILEINFO intrinsics. FFILEINFO is a superset of FGETINFO and is designed to replace it. However, programs may use either intrinsic exclusively or in conjunction with one another.

### USING FGETINFO

The program shown in figure 10-22 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.

If the *blksize* parameter is specified, the value returned will depend on the access mode, RIO or non-RIO. *Blksize* reflects only the size of the data area within the block when RIO access is used. In this case, the blocking factor can be computed in the usual way, block-size divided by record-size.

When the files are accessed using the non-RIO method, the *blksize* parameter will reflect the actual size of the block, including the Active Record Table area used to implement the RIO access method. This value can be used to determine the size of the data transfer, especially for file replication.

```

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-1; <<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=0:00:04; ELAPSED TIME=0:00:59

```

Figure 10-22. FGETINFO Intrinsic Example

The size of a block can be computed by:

$$B = \left\lceil \frac{F \cdot R + A}{128} \right\rceil \quad A = \left\lceil \frac{F}{16} \right\rceil$$

(A notation in the form  $\lceil x \rceil$  means “ceiling (x)” — the smallest integer greater than or equal to x. See p. 3-3). Where B = block-size (in sectors), F = blocking-factor, R = record-size (in words), and A = size of the Active Record Table (in words).

The blocking factor can be recomputed by:

$$F = \left\lfloor \frac{16 \cdot B \cdot 128}{16 \cdot B + 1} \right\rfloor$$

(A notation in the form  $\lfloor x \rfloor$  means “floor (x)” — the largest integer less than or equal to x).

Thus, the data-block size can be computed as F\*R. Note that double-integer arithmetic is necessary for the above calculation since  $B \cdot 128 \leq 32767$  (thus  $16 \cdot B \cdot 128 \leq 524272$ ).

The *recpt* parameter will always return the ordinal of the next actual record, whether it is active or inactive.

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 FILEERROR(DFILE1,4);
```

checks for a condition code of CCL and, if such is the case, calls the error-check procedure FILEERROR. 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.

## USING FFILEINFO

Although *itemvalue* is declared as a byte array, the data type of the value returned depends on the item itself. Some languages, SPL/3000 and Cobol II for example, permit you to pass any data type structure to a formal byte array parameter. This means you do not have to equivalence the data structures or to move the data from a temporary buffer to the desired variable. For this reason, the following example would retrieve the filename, *foptions*, and record pointer. SPL automatically converts the word addresses for *foptions* and *recptr* to byte addresses.

```
byte array FNAME (0:27);  
integer FOPTIONS;  
double RECPTR;  
FFILEINFO(FILENUM, 2, FOPTIONS, 9, RECPTR, 1, FNAME);
```

## 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 10-23, 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.

## USING FERRMSG

This intrinsic is called usually following a call to FCHECK. The error code returned in the call to FCHECK can then be used as a parameter in the call to FERRMSG.

For example, suppose a CCL condition is returned by a call to FCLOSE, a call to FCHECK requests the particular error code, then a call to FERRMSG can be used to retrieve a printable message associated with the code.

```
FCLOSE(FILNUM,1,0);  
IF<  
THEN BEGIN  
    FCHECK(FILNUM,ERRNUM);  
    FERRMSG(ERRNUM,MESSAGE,LENGTH);  
    PRINT(MESSAGE,-LENGTH,0);  
END  
TERMINATE;
```

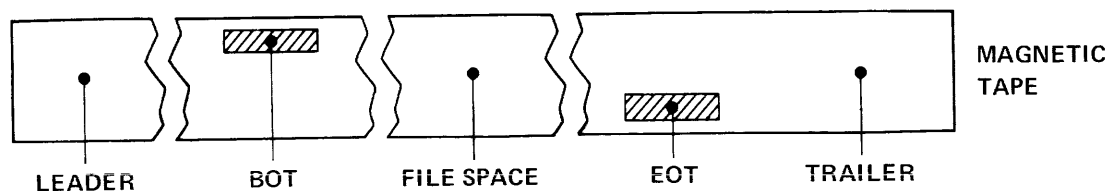
The message printed explains the FCHECK code. If the FCHECK code has no assigned meaning, the following message is returned:

```
UNDEFINED ERROR errorcode
```

## 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



```

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. ERRORS=000; NO. WARNINGS=000
PROCFSSOR TIME=0:00:04; ELAPSED TIME=0:00:58

```

Figure 10-23. FCHECK Intrinsic Example

should bring the operation to an orderly conclusion. The difference in treatment of these two physical tape markers is reflected by the file system intrinsic 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**

If the magnetic tape is unlabeled (as specified in the FOPEN intrinsic or :FILE command) and a user program attempts to write over or beyond the physical EOT 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).

If the magnetic tape is labeled (as specified in the FOPEN intrinsic or :FILE command), a CCL condition code is not returned when the tape passes the EOT marker. Attempts to write to the tape after the EOT marker is encountered cause end of volume (EOV) labels to be written. A message then is printed on the operator's console requesting another volume (reel of tape) to be mounted.

#### **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 BOT and BACKSPACE TAPE.

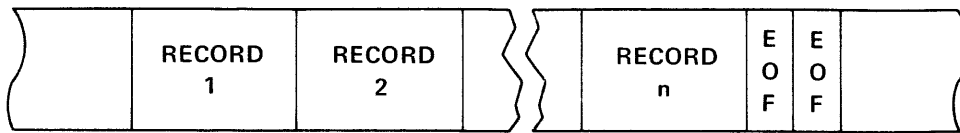
In summary, only those intrinsics which cause the magnetic tape to write information are capable of sensing the physical EOT marker. If a program designed to read a magnetic tape needed to detect the EOT marker, it could be done by using the FCONTROL intrinsic to read the physical status of the tape 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

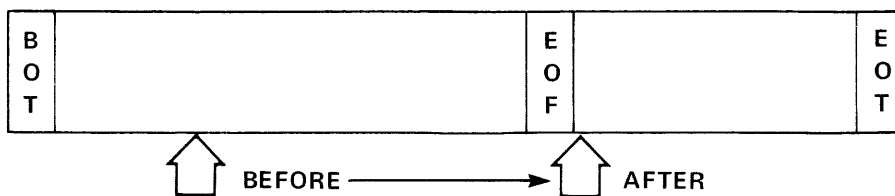
An FWRITE to magnetic tape, followed by any intrinsic call which reverses tape motion (for example, backspace a record, backspace a file, or rewind) causes the file system to write an EOF mark before initiating the reverse motion.

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.

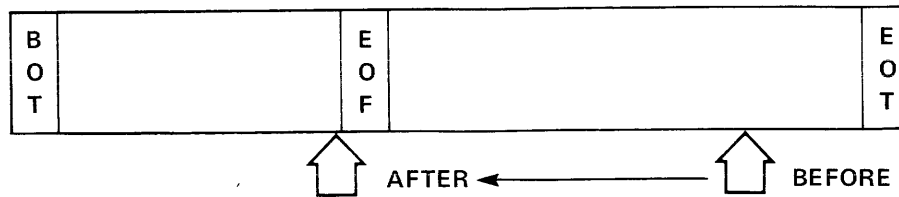


### SPACING FILE MARKS

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**

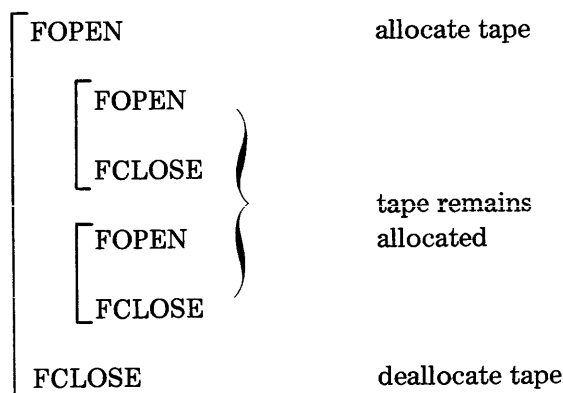
BOT (beginning of tape) and EOT (end of tape) correspond to the reflective markers on the reel of magnetic tape.

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 unlabeled magnetic tape is outlined in the flowchart of Figure 10-24.

Note that a tape closed with the temporary no-rewind disposition will be rewound and unloaded if certain additional conditions are not met. If 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.

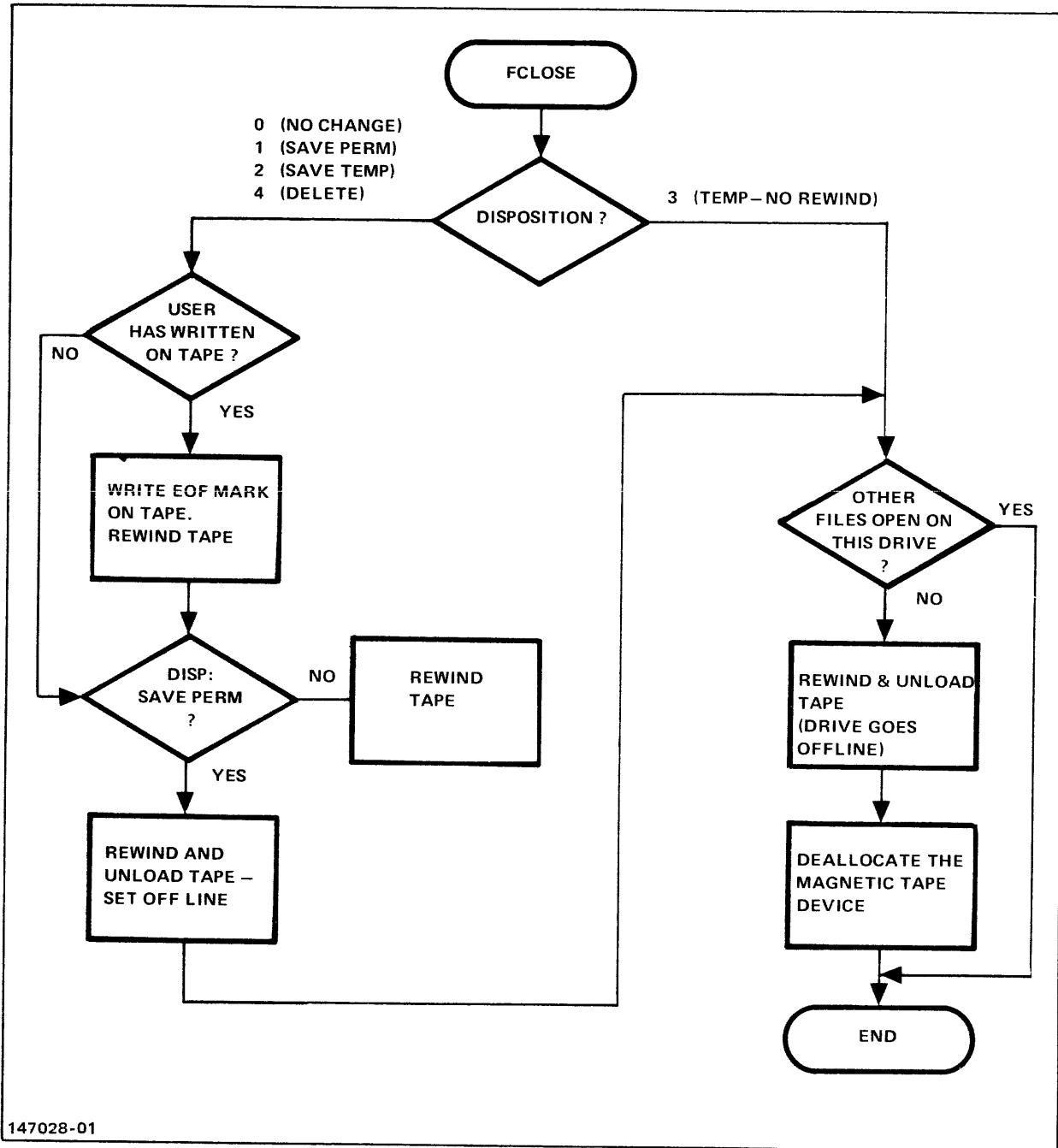


Figure 10-24. Using the FLCLOSE Intrinsic with Unlabeled Magnetic Tape

There are two standard formats for labels in common use: IBM and ANSI. They differ mainly in that the IBM labels are written in EBCDIC, not ASCII. The MPE tape labels facility can read and write labeled tapes that conform to the ANSI standard, and can read tapes that conform to the IBM standard. Only ANSI standard tapes support file lockwords.

## OPERATOR INTERVENTION

When a tape is mounted and put on-line, an interrupt occurs which causes the label on a labeled tape to be read. In this case, the volume name, and, unless the tape is the first volume of a volume set, the volume set name, are reported on the operator's console.

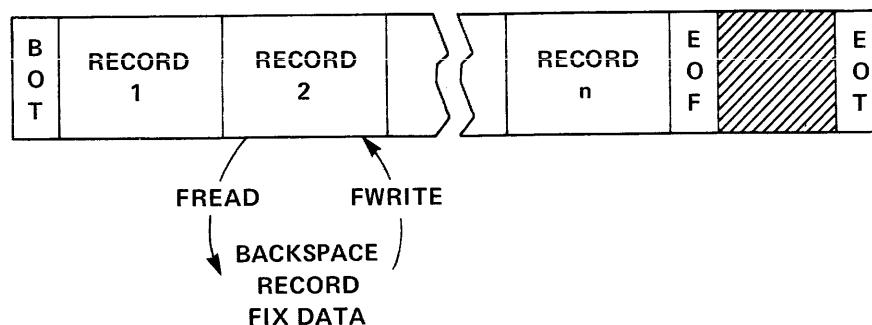
When an FOPEN for a labeled tape is executed, MPE checks to see whether a tape which satisfies the request is mounted. If so, it is assigned without operator intervention. Otherwise, a message is generated on the system console directing the operator to mount the required tape. If the operator mounts a suitable labeled tape, MPE assigns it to the requesting process, and no further operator intervention is required. If, however, the FOPEN specifies a write operation, and a blank tape is mounted, the operator must :REPLY to the request in order to assign it to the requestor. If the operator refuses the request with a :REPLY PIN,0, the FOPEN will fail with FSERR55 - Device Unavailable.

The FCLOSE intrinsic can be used to maintain position when creating or reading a labeled tape file that is part of a volume set. If you close the file with a disposition code of 3 or 4, the tape does not rewind, but remains positioned at the next file. If you close the file with a disposition code of 2, the tape rewinds to the beginning of the file but is not unloaded. A subsequent request to open the file does not reposition the tape if the sequence (*seq*) subparameter is NEXT, or default. A disposition code of 0 (no change) or 1 (save permanent) implies the close of an entire volume set.

## 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:



## MPE TAPE LABELS

MPE provides a means whereby you can read and write labels on magnetic tape files. Labeled magnetic tape files are intended to provide for:

1. A permanent identification for tape reels or volumes.
2. Additional security, to protect against accidental over-writing or unauthorized access to files.
3. Easy access to files which extend over more than one volume.
4. More than one file on a volume.
5. Retrieval of files by file names.
6. Facilitating information interchange between computer systems.

Each tape volume, when first written, is assigned a unique identifier, up to six printable characters, to permanently identify it. This identifier is the Volume Name. It is often strictly numeric, so that volumes in an installation's library can be sorted and stored by this number.

A collection of volumes containing one or a related group of files is called a volume set. The volume name of the first reel in the set is taken as the Volume Set Name.

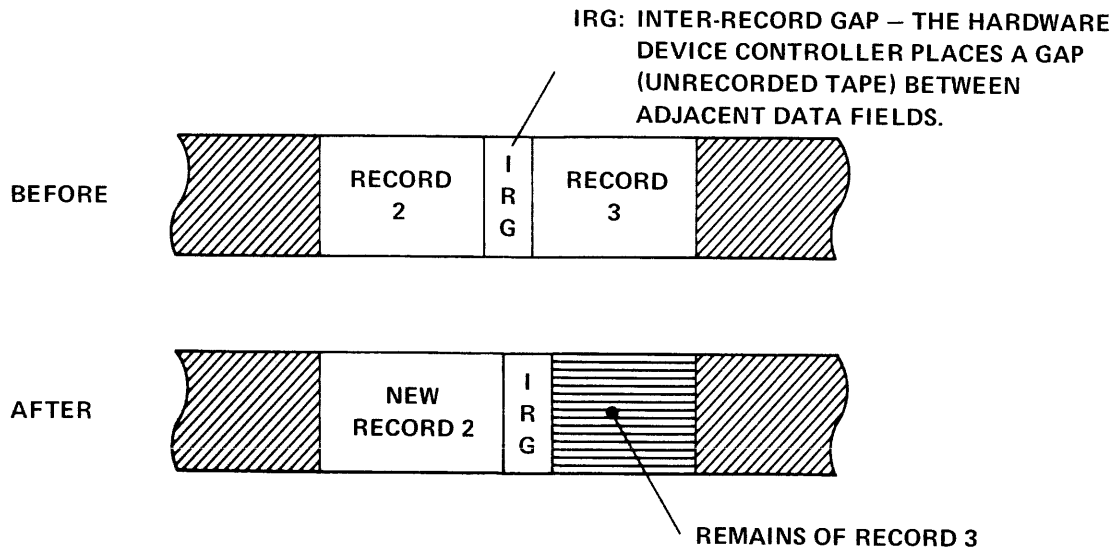
Each file on a labeled tape has a header label or labels, which contain the name of the file, the sequential position of the file in the volume set, and the sequential number of the volume of the file if the file extends over more than one volume. They may optionally contain the record and block size, a file lockword, and whether the file is ASCII or binary.

When opening a labeled tape file to be read, you must specify the Volume Set Name. This may appear either in a file equation LABEL = parameter, or in the FORMSMMSG parameter of FOPEN; if it appears in neither place, the console operator will be asked for the Volume Set Name. One may also specify either that a specified file name is to be sought within the volume set, or merely that the next sequential file is to be accessed. If no file follows, then FOPEN will return an End of Volume Set error (FSERR123).

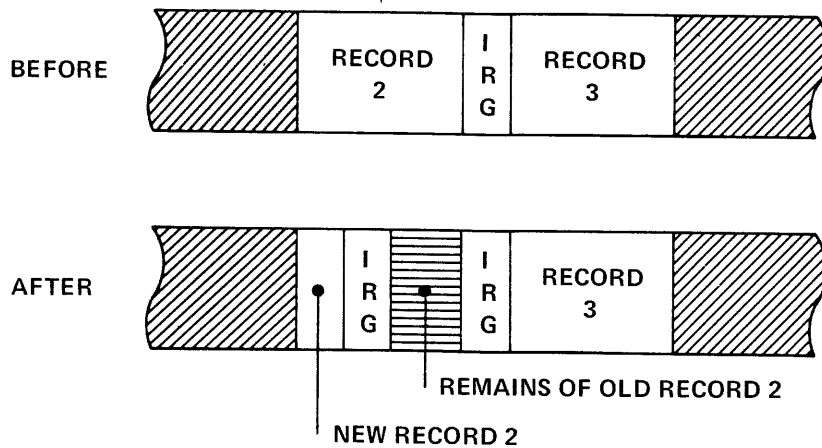
When opening a labeled tape file to be written, the volume set name is specified as for reading. One may request a specific named file, or that the next sequential file be written, or that a file be added to the end of the volume set. An End of Volume Set error will be reported if a specific named file is requested but is not found. Of course, if there are other files following the file to be written, the contents of those files will be lost.

You may close a file without closing the volume set containing it. What this means is that a subsequent FOPEN specifying the same volume set name will be able to access a file on the currently mounted volume of the volume set without operator intervention. The volume thus accessed need not be the first one in the volume set.

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 AN UNLABELED MAGNETIC TAPE FILE

Figure 10-25 contains a program that copies an unlabeled 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.

*recsize*                            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 FILEERROR(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 FILEERROR(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 FILEERROR(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 FILEERROR(MT,3); <<CHECK FOR ERROR>>
00031000 00043 1 FSPACE(MT,RECD*POSITION); <<NEXT FILE 2 RECD>>
00032000 00046 1 IF <> THEN FILEERROR(MT,4); <<CHECK FOR ERROR>>
00032100 00052 1
00033000 00052 1 FWRITE(MT,BUFFER,LGTH,0); <<TAPE FILE 2>>
00034000 00057 1 IF <> THEN FILEERROR(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 FILEERROR(MT,6); <<CHECK FOR ERROR>>
00038000 00073 1 FCONTROL(MT,8,DUMMY); <<BACK TO START FILE 1>>
00039000 00077 1 IF < THEN FILEERROR(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 FILEERROR(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 FILEERROR(MT,9); <<CHECK FOR ERROR>>
00051000 00125 1
00052000 00125 1 FCLOSE(MT,0,0); <<MAG TAPE>>
00053000 00130 1 IF < THEN FILEERROR(MT,10); <<CHECK FOR ERROR>>
00054000 00134 1
00055000 00134 1 END.
PRIMARY DB STORAGE=%007; SECONDARY DB STORAGE=%00111

```

Figure 10-25. Unlabeled Magnetic Tape Example

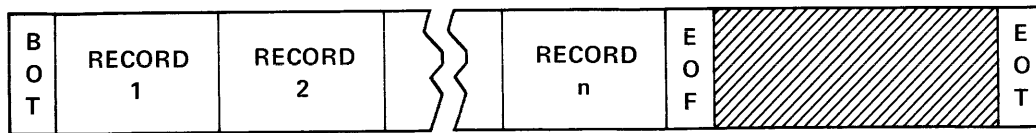
Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable MT.

The statement

```
IF < THEN FILERROR(MT,1);
```

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 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).

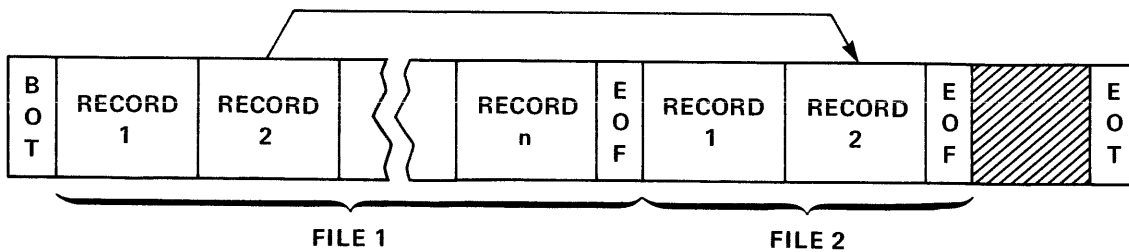
The record position is set to the next record in file 1 by incrementing RECD'POSITION with the statement

```
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:



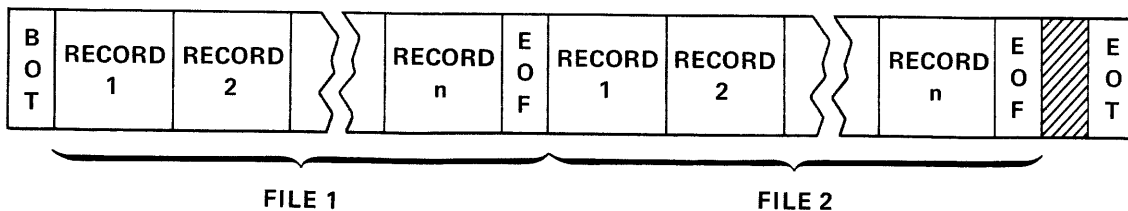
Note that the reverse tape motion after a write creates an EOF mark (see end of file 2).

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:



## OPENING A LABELED MAGNETIC TAPE FILE

Figure 10-26 shows a program that opens a labeled magnetic tape file and a disc file; reads the contents of the tape file and writes the records read to the disc file; closes the tape file; and, finally, closes the disc file as a permanent file.

The statement

```
FNO1:=FOPEN(FILID1,%1004,,DEV,LABELID);
```

calls FOPEN to open the labeled magnetic tape file. The parameters specified are

*formal designator*                      TAPEFILE, stored in the byte array FILID1.

*foptions*                                %1004, 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	1	0	0	0	0	0	0	1	0	1	Binary
					1		0			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 same as formal file designator.

Bits (10:3) = 000. (Default)

Record Format: Fixed length. Bits (8:2) = 00. (Default)

Carriage Control: No carriage control. Bit (7:1) = 0. (Default)

Labeled Tape: Labeled. Bit (6:1) = 1.

*aoptions*

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 access options:

Access Type: Update. Bits (12:4) = 0101.

*resize*

Default.

*device*

TAPE, contained in the byte array DEV.

```

$CONTROL USLINIT
BEGIN
  BYTE ARRAY FILID1(0:8):="TAPEFILE ";
  BYTE ARRAY FILID2(0:8):="          ";
  BYTE ARRAY LABELID(0:79):=".FIL001,ANS,12/31/77;";
  BYTE ARRAY DEV(0:4):="TAPE ";

  APPRAY MSGBUF(0:35);
  ARRAY INBUF(0:39);
  ARRAY FILID2(*)=FILID2;

  INTEGER FNO1,FNO2,LGTH;

  INTRINSIC FOPEN,FCLOSE,FREAD,FWRITE,READ,PRINT,PRINT'FILE'INFO,
    QUIT,CAUSEBREAK,FREADLABEL;

  PROCEDURE FILEPROR(FILENO,QUITNO);
    VALUE QUITNO;
    INTEGER FILENO,QUITNO;
    BEGIN
      PRINT'FILE'INFO(FILENO);
      QUIT(QUITNO);
    END;

  << END OF DECLARATIONS >>

  MOVE MSGBUF:="NAME OF NEW DISC FILE TO BE CREATED?";
  PRINT(MSGBUF,-8,0);
  READ(FILID2,4); << READ NAME OF NEW DISC FILE >>

  FNO1:=FOPEN(FILID1,%1001,5,,DEV,LABELID); << OPEN LABELED
    TAPE FILE >>
  IF < THEN << CHECK FOR ERROR >>
    BEGIN
      MOVE MSGBUF:="CAN'T OPEN TAPE FILE";
      PRINT(MSGBUF,-20,0);
      FILERROR(FNO1,1);
    END;

  FNO2:=FOPEN(FILID2,4,5); << OPEN NEW DISC FILE >>
  IF < THEN << CHECK FOR ERROR >>
    BEGIN
      MOVE MSGBUF:="CAN'T OPEN DISC FILE";
      PRINT(MSGBUF,-20,0);
      FILERROR(FNO2,2);
    END;

  FREADLABEL(FNO1,INBUF,40); << READ USER LABEL >>
  IF <> THEN FILERROR(FNO1,3); << CHECK FOR ERROR >>
  PRINT(INBUF,40,0);

  READ'WRITE'LOOP:

  LGTH:=FREAD(FNO1,INBUF,40); << READ RECORD FROM
    TAPE FILE >>

```

Figure 10-26. Opening a Labeled Magnetic Tape File (Sheet 1 of 2)

```

IF < THEN                                << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T READ TAPE FILE";
    PRINT(MSGBUF,-20,0);
    FILERROR(FNO1,4);
  END;
IF > THEN GO CLOSE1; << CHECK FOR END-OF-FILE >>

FWRITE(FNO2,INBUF,LGTH,0); << WRITE RECORD TO
                           DISC FILE >>
IF <> THEN                                << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T WRITE TO DISC FILE";
    PRINT(MSGBUF,-24,0);
    FILERROR(FNO2,5);
  END;

GOTO READ'WRITE'LOOP;

CLOSE1:

FCLOSE(FNO1,1,0); << CLOSE, REWIND, AND UNLOAD TAPE FILE >>
IF < THEN                                << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T CLOSE TAPE FILE";
    PRINT(MSGBUF,-21,0);
    FILERROR(FNO1,6);
  END;

CLOSE2:

FCLOSE(FNO2,1,0); << CLOSE DISC FILE AS
                  PERMANENT FILE >>
IF < THEN                                << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T CLOSE DISC FILE";
    PRINT(MSGBUF,-21,0);
    MOVE MSGBUF:="CHECK FOR DUPLICATE NAME";
    PRINT(MSGBUF,-24,0);
    MOVE MSGBUF:="FIX, THEN TYPE 'RESUME'";
    PRINT(MSGBUF,-23,0);
    GOTO CLOSE2; << TRY AGAIN >>
  END;
END.

```

Figure 10-26. Opening a Labeled Magnetic Tape File (Sheet 2 of 2)

tape label

Contained in the byte array LABELID (*formmsg* parameter). LABELID is declared with the value

```
.FIL001,ANS,12/31/77;
```

(see statement number 5 in figure 10-26). Note that the tape label begins with a period and ends with a semi-colon. This is necessary to distinguish the tape label from a forms message (another use for this parameter).

When the FOPEN intrinsic call executes, MPE sends a message to the system console, requesting the Console Operator to mount the tape labeled FIL001 (if it is not already mounted).

The statement

```
FNO2:=FOPEN(FILID2,4,5);
```

opens a new disc file.

The program then reads records from the tape file with the statement

```
LGTH:=FREAD(FNO1,INBUF,40);
```

and writes these records to the disc file with the statement

```
FWRITE(FNO2,INBUF,LGTH,0);
```

When all records in the tape file have been read, both files are closed. The disc file is saved as a permanent file.

## WRITING A TAPE LABEL

The MPE :FILE command or FOPEN intrinsic is used to write ANSI-standard tape labels (MPE will not write IBM-standard tape labels). See the *MPE Commands Reference Manual* for a discussion of writing tape labels with the :FILE command.

The program shown in Figure 10-27 opens a magnetic tape file and writes a label on the file.

The statement

```
BYTE ARRAY LABELID(0:79):=" .FIL099,ANS,12/31/77,NEXT;"
```

declares a byte array of 80 bytes and initializes it to

```
.FIL099,ANS,12/31/77,NEXT;
```

which constitutes an ANSI-standard label. Note that the tape label begins with a period and ends with a semi-colon. This is necessary to distinguish the tape label from a forms message (which is another use for the same FOPEN parameter). The LABELID byte array will be used in the FOPEN intrinsic call to specify a file label as follows:

Volume Identification: FIL099



```

$CONTROL USLINIT
BEGIN
  BYTE ARRAY FILID1(0:8):="          ";
  BYTE ARRAY FILID2(0:8):="NEWTAPE1 ";
  BYTE ARRAY LABELID(0:79):=".FILO99,ANS,12/31/77,NEXT;";
  BYTE ARRAY DEV(0:4):="TAPE ";

  ARRAY MSGBUF(0:35);
  ARRAY INBUF(0:39);
  ARRAY FIL'ID1(*)=FILID1;
  ARRAY USERLABL(0:79);

  INTEGER FNO1,FNO2,LGTH;

  INTRINSIC FOPEN,FCLOSE,PRINT'FILE'INFO,QUIT,PRINT,READ,
    FWRITELABEL,FREAD,FWRITE;

  PROCEDURE FILERROR(FILENO,QUITNO);
    VALUE QUITNO;
    INTEGER FILENO,QUITNO;
    BEGIN
      PRINT'FILE'INFO(FILENO);
      QUIT(QUITNO);
    END;

  << END OF DECLARATIONS >>

  MOVE MSGBUF:="NAME OF INPUT FILE?";
  PRINT(MSGBUF,-19,0);
  READ(FIL'ID1,-8); << READ NAME OF INPUT FILE >>

  FNO1:=FOPEN(FILID1,1,5); << OPEN OLD DISC FILE >>
  IF < THEN << CHECK FOR ERROR >>
    BEGIN
      MOVE MSGBUF:="CAN'T OPEN DISC FILE";
      PRINT(MSGBUF,-20,0);
      FILERROR(FNO1,1);
    END;

  FNO2:=FOPEN(FILID2,%1004,5,,DEV,LABELID); << OPEN NEW LABELED
    TAPE FILE >>
  IF < THEN << CHECK FOR ERROR >>
    BEGIN
      MOVE MSGBUF:="CAN'T OPEN TAPE FILE";
      PRINT(MSGBUF,-20,0);
      FILERROR(FNO2,2);
    END;

  MOVE USERLABL:=" ";
  MOVE USERLABL:=USERLABL(0),(40);
  MOVE USERLABL:="UHL1 USER HEADER LABEL NO. 1";
  FWRITELABEL(FNO2,USERLABL,40,0); << WRITE USER HEADER LABEL >>
  IF <> THEN FILERROR(FNO2,3); << CHECK FOR ERROR >>

  READ'WRITE'LOOP:

```

Figure 10-27. Writing a Tape Label (Sheet 1 of 2)

```

LGTH:=FREAD(FNO1,INBUF,40); << READ RECORD FROM DISC FILE >>
IF < THEN << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T READ DISC FILE";
    PRINT(MSGBUF,-20,0);
    FILEERROR(FNO1,4);
  END;
IF > THEN GO CLOSE; << CHECK FOR END-OF-FILE >>

FWRITE(FNO2,INBUF,LGTH,0); << WRITE RECORD TO LABELED TAPE FILE >>
IF <> THEN << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T WRITE TO TAPE FILE";
    PRINT(MSGBUF,-24,0);
    FILEERROR(FNO2,5);
  END;

CLOSE:

FCLOSE(FNO1,0,0); << CLOSE DISC FILE >>
IF < THEN << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T CLOSE DISC FILE";
    PRINT(MSGBUF,-21,0);
    FILEERROR(FNO1,6);
  END;

FCLOSE(FNO2,1,0); << CLOSE, REWIND, AND UNLOAD TAPE FILE >>
IF < THEN << CHECK FOR ERROR >>
  BEGIN
    MOVE MSGBUF:="CAN'T CLOSE TAPE FILE";
    PRINT(MSGBUF,-21,0);
    FILEERROR(FNO2,7);
  END;
END.

```

Figure 10-27. Writing a Tape Label (Sheet 2 of 2)

*Label Type:* ANS (ANSI)

*Expiration Date:* 12/31/77. This is the date after which the file can be overwritten. If you attempt to overwrite the file before this date, MPE will send a message to the Console Operator asking for confirmation that such is really desired. This affords an extra measure of protection against inadvertently destroying a tape by overwriting when a WRITE RING is left in the tape by mistake.

*sequence:* NEXT. Signifies that the file is to be positioned at the next file on the tape.

The statement

```
FNO2:=FOPEN(FILID2,%1004,5,,DEV,LABELID,1);
```

opens a new tape file and writes the tape label as specified by LABELID.

### READING A LABELED MAGNETIC TAPE FILE

Once a labeled tape file has been opened, the FREAD intrinsic may be used in the same manner as on an unlabeled tape file. The system defaults to the blocksize, recordsize and file format on the tape label if these parameters are not specified. You can call FGETINFO or FFILEINFO to get these values.

The program shown in Figure 10-26 reads a labeled magnetic tape file in sequential order.

The labeled tape file is opened with the statement

```
FNO1:=FOPEN(FILID1,%1005,5,,DEV,LABELID);
```

The file label is contained in the byte array LABELID.

The block of statements

```
READ'WRITE'LOOP:  
    .  
    .  
    .  
GOTO READ'WRITE'LOOP;
```

form a read/write loop. Records are read from the tape file in sequential order with the statement

```
LGTH:=FREAD(FNO1,INBUF,40);
```

and written to a disc file with the statement

```
FWRITE (FNO2,INBUF,LGTH,0);
```

## WRITING TO A LABELED MAGNETIC TAPE FILE

Writing records to a labeled tape file is slightly different than writing to an unlabeled tape file as follows:

If the magnetic tape is unlabeled and a user program attempts to write over or beyond the physical EOT 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).

If the magnetic tape is labeled, a CCL condition code is not returned when the tape passes the EOT marker. Attempts to write to the tape after the EOT marker is encountered cause end of volume (EOV) labels to be written. A message then is printed on the operator's console requesting another volume (reel of tape) to be mounted.

The program shown in Figure 10-27 opens an existing disc file and a new labeled tape file, reads records from the disc file and writes these records to the tape file. If an attempt is made to write records on the tape beyond the EOT marker, MPE will write EOV1 and EOV2 labels on the tape and request the Console Operator to mount another reel of tape.

The statement

```
FWRITE(FNO2,INBUF,LGTH,0);
```

writes the contents of array INBUF onto the tape file signified by FNO2. The LGTH parameter specifies the number of words to be written.

## WRITING A USER-DEFINED FILE LABEL ON A LABELED TAPE FILE

User-defined labels are used to further identify files and may be used in addition to the ANSI-standard labels. Note that user-defined labels may not be written on unlabeled magnetic tape files.

User-defined labels are written on files with the FWRITELABEL intrinsic instead of the FOPEN intrinsic (as is the case for writing ANSI-standard labels).

User-defined labels for labeled tape files are slightly different than user-defined labels for disc files in that user-defined labels for tape files must be 80 bytes (40 words) in length. The tape label information need not occupy all 80 bytes, however, and you can set unused portions of the space equal to blanks.

Figure 10-27 contains a program that opens a new tape file and writes an ANSI-standard label on this file, then writes a user-defined header label with the FWRITELABEL intrinsic.

The statement

```
FNO2:=FOPEN(FILID2,%1004,5,,DEV,LABELID,1);
```

opens a new tape file named NEWTAPE1 (the name is contained in byte array FILID2), writes an ANSI-standard label (contained in the byte array LABELID) to the file.

## Labeled Tapes

Since it is important that tape density be an attribute of an entire volume set, the density of a labeled tape is determined by the first FOPEN, the volume set open. Density specified on all subsequent FOPENs will be ignored until the entire volume set is unloaded from the tape drive. Note that the file system will rewrite VOL 1 labels in certain situations, such as in the case of rewriting a labeled tape at 6250 BPI which had previously been written at 1600 BPI. This is to prevent creating a multireel labeled volume set containing volumes of different densities.

If the user omits the DEN keyword parameter (see FOPEN in Section II), or specifies the keyword and omits the density selection when writing a new label on a previously unlabeled tape, the file system assigns a default density (in this case 6250 BPI) to the FOPEN request.

If a volume set FOPEN takes the default density, and is satisfied by the proper unlabeled tape volume, the density of the volume is left unchanged. However, if a volume set open is satisfied by a proper labeled tape volume, and the density of the tape does not match the specific density requested by the user, the VOL1 label may be rewritten at a new density. If the volume set is currently empty (which is determined by the presence of a tape mark after the VOL1 header), or if the volume set open specifies NEXT as the sequence type, the VOL1 label is rewritten. In all other cases the volume set density is unchanged.

When a reel switch occurs while writing a file to a labeled tape, each succeeding volume is written at the same density as its predecessor, i.e., all volumes are written at the same density as the first. Therefore, if a reel switch request is satisfied by a labeled tape, its VOL1 label is rewritten if the label's density is different from that of the volume set. This ensures that all volumes of a labeled volume set are written at the same density.

## Unlabeled Tapes

Since unlabeled tapes may be accessed as many different files simultaneously (providing that the FOPENs occur in the same process tree), and may be returned to load point using several different file system Intrinsic, the default density selection is somewhat different for unlabeled tapes. If a write operation occurs while an unlabeled tape is at load point, the density of the tape will be set to the density requested by the most recent FOPEN which had write access to the tape.

The default density for the first FOPEN of a variable density tape drive is 6250 BPI. For any subsequent FOPENs, the density specified by the previous FOPEN is retained. Note, however, that any FOPEN which specifies a particular density (user supplied, not default), and has write access to the file takes precedence over all previous FOPENs. Remember that density can be changed only at load point, and if the next operation is a write.

When an application does its own unlabeled tape reel management, all subsequent reels of a volume set will be automatically set to the density of the first reel, providing that no FOPENs which change the density of the tape drive occur during the writing of the reels. In this case, it would be possible to generate volumes of different densities (i.e., one volume at 1600 BPI, and one volume at 6250 BPI). Note, however, that it is not possible in any case to write a tape at two different densities, since density can only be changed at load point.

## Determining Tape Density

The FFILENO Intrinsic will return the density of a tape file if item number 46 is specified. The density will be returned as an integer, either 1600 or 6250, signifying BPI. This item number is only valid for files residing on variable density tape drives. For any other requests, the Intrinsic will return a zero.

When a tape is not at load point at the time of the call, the intrinsic returns the actual density of the tape. If the Intrinsic is called while the tape is at load point (only possible for unlabeled tapes), a value based on the FOPEN access is returned. If the tape was accessed READ only, the Intrinsic returns the actual density of the tape. If the tape was accessed in any other mode, the Intrinsic assumes that the next operation will be a WRITE, and returns the tape density requested at FOPEN time.

## 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.)

The statements

```
MOVE USERLABL:= "    ";  
MOVE USERLAB;:=USERLABL(0),(40);
```

fill the array USERLABL with 80 ASCII blanks (40 words), and the statement

```
MOVE USERLABL:= "UHL1 USER HEADER HEADER LABEL NO. 1";
```

moves the desired user label into the first 35 bytes of the array (replacing the blanks).

The statement

```
FWRITELABEL(FNO2,USERLABEL,40,0);
```

writes all 80 characters into the file as a user-defined header label.

Note that in order to write a user-defined header label, the FWRITELABEL intrinsic must be called before the first FWRITE to the file. (MPE will write user-defined trailer labels if FWRITELABEL is called after the first FWRITE).

#### READING A USER-DEFINED FILE LABEL ON A LABELED TAPE FILE

The FREADLABEL intrinsic is used to read a user-defined label on a labeled magnetic tape file. To read a user-defined header label, the FREADLABEL intrinsic must be called before the first FREAD is issued for the file. Execution of the first FREAD causes MPE to skip past any unread user-defined header labels.

If Figure 10-26, the statement

```
FREADLABEL(FNO1,INBUF,40);
```

reads a user-defined header label. The parameters specified are as follows:

FNO2	The file number as returned by the FOPEN intrinsic.
INBUF	An array to which the label is transferred.
40	Specifies the number of words to be read.

#### DENSITY SELECTION ON LABELED AND UNLABELED TAPES

The following information applies to density selection for variable density tape drives. Density, if specified, is ignored by tape drives which operate only at one density.

In figure 10-25, the statement

```
FSPACE(MT,RECD'POSITION);
```

is used to space a tape file. The parameters specified are

<i>filenum</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 10-25 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 or undefined-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,%2);
```

establishes access-mode options as outlined below. The parameters specified are

<i>filenum</i>	Designated by FILE1, which was assigned the file number by FOPEN when the file was opened. (It is a terminal in this example.)
<i>modeflags</i>	%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 access-mode options:

### Critical Output Verification:

All physical output of blocks 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 Control by User:**

MPE will issue carriage return/linefeed. Bit (13:1) = 0.

## **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.

## USER LOGGING

The MPE IV User Logging Facility provides a flexible transaction logging capability which enables you to journalize additions and modifications to your data bases and subsystem files. User logging permits you to journalize on two mediums: tape and disc. When a tape file is used, journal entries are saved on a dedicated magnetic tape file outside the domain of the system. When a disc file is used, the journal entries are saved in a special disc file you create specifically for logging transactions. It is also possible to organize your applications so that both mediums are used for logging.

Logging is done programmatically by means of three intrinsics: OPENLOG, WRITELOG, and CLOSELOG. OPENLOG provides access to the logging facility, CLOSELOG closes access to the system, and WRITELOG writes journal entries to the logging file. This creates a copy of the data-set's modifications. If the data-set is lost, the logging tape or disc file can be used in conjunction with a backup copy of the file to recover the lost transactions.

### HOW USER LOGGING WORKS

The User Logging Facility contains a buffer file, a logging process, a logging file for each process, and a data segment containing a communications area and a buffer for each active log file. (See Figure 10-28) The function of two of these segments, the logging process and the logging buffer file, varies for disc file logging and tape file logging.

When you access the logging facility and request tape files, your entries are placed in the buffer area of the logging data segment. Once this buffer area is full, the contents are written to the logging buffer file on disc. If there is no space available, two things can happen according to the *MODE* you specified.

If you indicated no wait in the *mode* parameter, the communications area of the data segment sends an error message indicating your transaction has not been completed, and you need to resubmit your request (similar to NOWAITIO).

If you indicated wait in the *mode* parameter, your process is suspended until the logging process writes the contents of the buffer to the media specified, tape or disc, and then reactivates your process (similar to WAITIO).

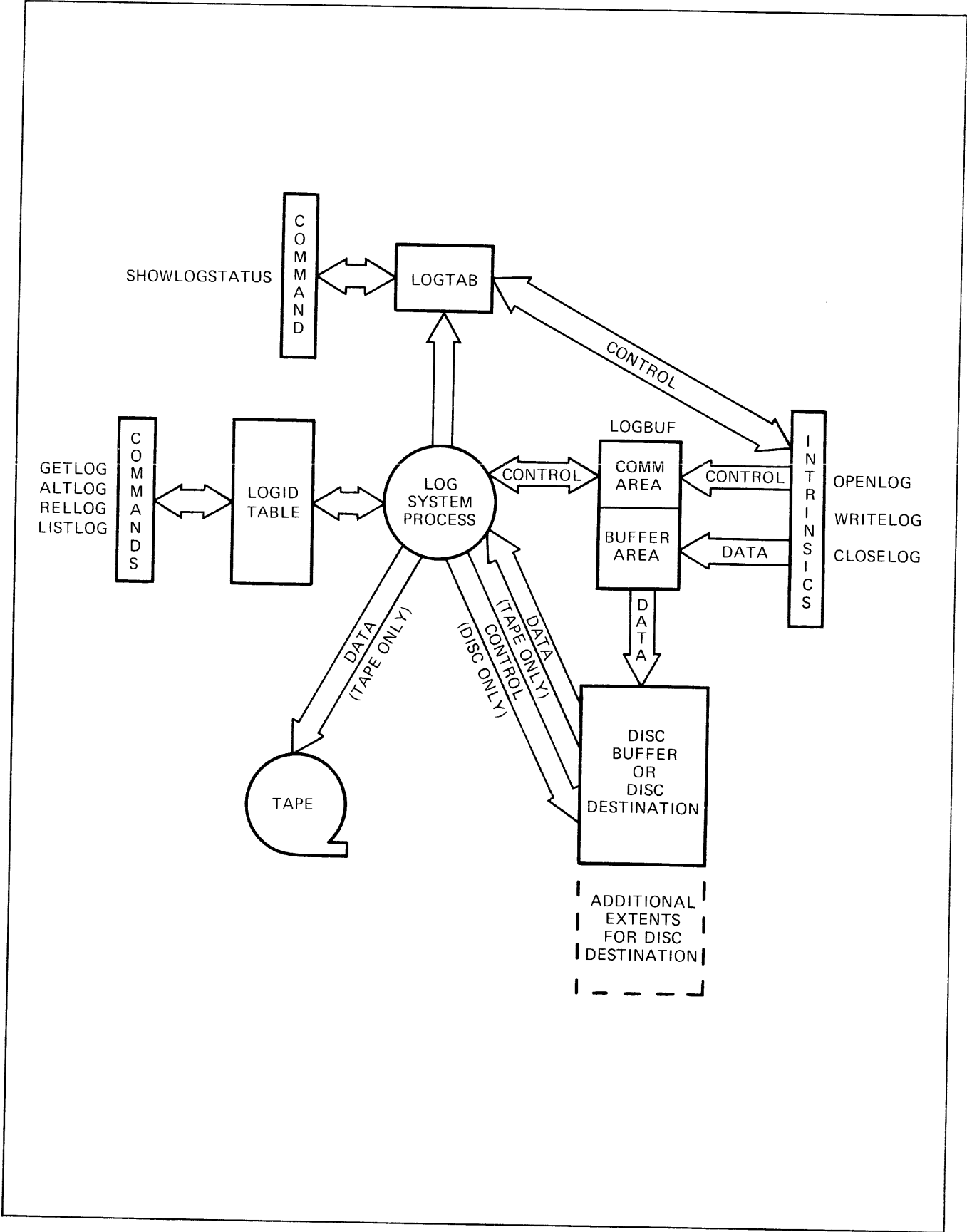


Figure 10-28. User Logging Facility

The logging process acts as an interface between the logging buffer file and the logging tape file. The logging process writes records to the mag tape destination you have requested. This process is independent of your process; thus the logging process can be writing records from the disc buffer to tape at the same time you are adding additional records to the logging buffer. Once the logging process has made space available in the logging buffer file, the logging process reactivates your process.

When a disc file is specified, the logging process and logging buffer function differently. When the WRITELOG intrinsic has been called, your entries are loaded into the buffer area of the logging data segment. Your records are moved to your disc destination file when the buffer area is full. When the destination file becomes full, you receive a warning message only if you have set the *mode* parameter for no-wait. Otherwise, the WRITELOG intrinsic suspends your process and activates the logging process. If needed, the logging process allocates additional extents to your disc destination file up to the maximum that you specified in the :BUILD command and logging continues. The logging process then activates your process.

You need the record formats to directly access the logging files.

**Logging Record Format:**

record size = 128 words  
 user area = 119 words

**LOG RECORD AT OPENLOG**

0	2	3	4	6	7	11	12	24	25	127
rec#	cksum	code	time	date	logid	log#	creator	pin		

**USER OR SUBSYSTEM LOG RECORD OR CONTINUATION RECORD**

0	2	3	4	6	7	8	9	127
rec#	cksum	code	time	date	log#	len	user area	

**LOG RECORD AT CLOSELOG**

0	2	3	4	6	7	11	12	24	25	127
rec#	cksum	code	time	date	logid	log#	creator	pin		

**CRASH MARK**

0	2	3	4	6	7	127
rec#	cksum	code	time	date		

START/RESTART (From Log Operator Command)

0	2	3	4	6	7	11	127
rec#	cksum	code	time	date	logid		

TRAILER RECORD

0	2	3	4	6	7	11	127
rec#	cksum	code	time	date	logid		

SPACE RECORD

2	3	4	6	7	127	
rec#	cksum	code	time	date		

CODE DEFINITION:

- Code = 1 Open log
- 2 User/Subsystem Record
- 3 Close log
- 4 Header
- 5 Trailer
- 6 Restart
- 7 Continuation of User of Subsystem Record
- 9 Crash Marker
- 10 End Transaction Marker
- 11 Begin Transaction Marker
- Space Null Record

NOTE

1. The checksum algorithm uses the EXCLUSIVE OR function against a base of negative one. It is calculated on the entire record except the checksum word.
2. 'PIN' is the index into the PCB table. To get an absolute PIN value, divide by 16.
3. The space record code is 8224 (%20040)

EFFECTIVE USE OF USER LOGGING

Effective use of the User Logging Facility includes writing the log file and designing a recovery procedure. You must use the logging intrinsics in your program to write data to a logging file to be used for recovery. You must also program a recovery procedure that can read the log file and apply it to a backup copy of your data-set to recover your data.

The following steps are suggested as a procedure for effective use of User Logging.

1. Select your logging identifier. You can find an explanation of the identifier in the MPE Commands manual. Make sure you use passwords/lockwords provided in the :GETLOG command if data security is necessary. Also be sure to specify the configured device class name, i.e., TAPE for mag tape, SDISC for serial disc device, and CTAPE for cartridge tape unit (such as 7911/7912). Otherwise an FOPEN failure will result.
2. Design your application and data structures.
3. Determine what information must be logged in order to recover the data structures of your application. Assume you will have a back-up copy of the data structure to which you can apply the log file to recover.

Place the appropriate calls to the User Logging Intrinsic in your application to log the data necessary for recovery. It is best to log edited and verified data by logging a full transaction at one time or wrapping transactions with start and completion records.

4. Design and program your recovery procedure. Again, assume the recovery procedure will have a back-up copy of your application data structure to which it will apply the log file. Your recovery program must recognize the User Logging file record formats. (see "How User Logging Works")
5. If you specify in the :GETLOG command that your logging identifier is associated with a disc file, you must build that file making sure the file has the proper code [`CODE=LOG`] and enough disc space to contain one day's output. Be generous with the disc file space. Divide the file into several extents, but allocate only the first extent. This minimizes the impact of a large file on your system. The user logging process allocates additional extents when necessary.
6. Have the operator start the user logging process for your logging identifier (See :LOG Command in the MPE Console Operators Manual.)
7. Run your application to log changes to your data sets to the User Logging file you specified in the :GETLOG command.

If it becomes necessary to recover your data-sets, restore the back-up copy that you have saved and run the recovery procedure you wrote to reapply changes you made to the data structures.

## SUGGESTED LOG FILE USES

Usage of a log file, especially if many users are accessing the same user log file, starts with your application. Set up a log file separate from your data base log file with information which points to your entry. This separate log file might include the following information.

1. User group and account names you are logged onto. This can be determined using the WHO intrinsic.
2. Job/session input device number using the WHO intrinsic. This would only give a history however, and not help directly in recovery.
3. Process Identification Number (PIN) of the process accessing the log file right now using the GETPROCID intrinsic.
4. Date and time of opening the actual log file. This information can be accessed by the CALENDAR and CLOCK intrinsics. (The date and time returned by these intrinsics match the format of this information in the data base log file.) Put in calls to these intrinsics immediately after a successful OPENLOG intrinsic call.
5. The fully qualified file name of the data set log file being accessed including the ASCII of the LOGID.

When you need to recover the file listing, the additional file can be read and the log file opened. Search the log file for the LOGID and the creator that match in the OPENLOG record (type 1). Verify using the date and time. Use the PIN to verify the creator in case there are duplicate creators. Then pick up the LOG number. All records with this LOG number will be in your file.



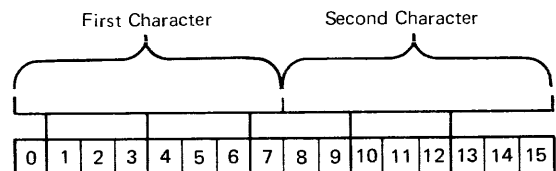
# ASCII CHARACTER SET

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
DEL	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		Private volume information (while file is open).
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.

# MAGNETIC TAPE LABELS

APPENDIX

D

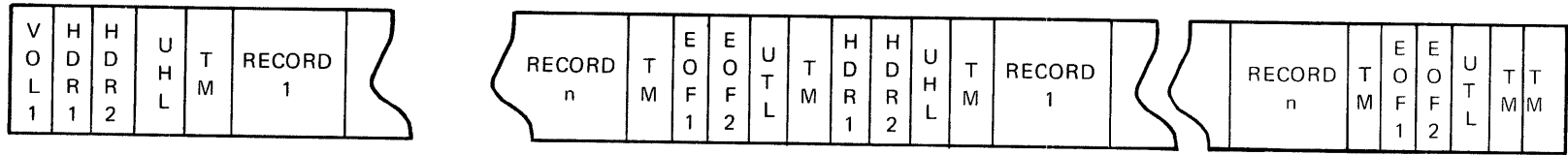
Labels conforming to ANSI-standard can be read and written on magnetic tape files by MPE. IBM-standard labels can be read, but cannot be written by MPE.

The tape labels written by MPE consist of:

Volume Header	At the beginning of each reel of tape.
File Header 1	At the beginning of each file on the reel.
File Header 2	Following File Header 1.
End-of-File 1	At the end of each file on the reel.
End-of-File 2	Following End-of-File 1.
End-of-Volume 1	At the end of a reel if the tape spans more than one volume.
End-of-Volume 2	Following End-of-Volume 1.

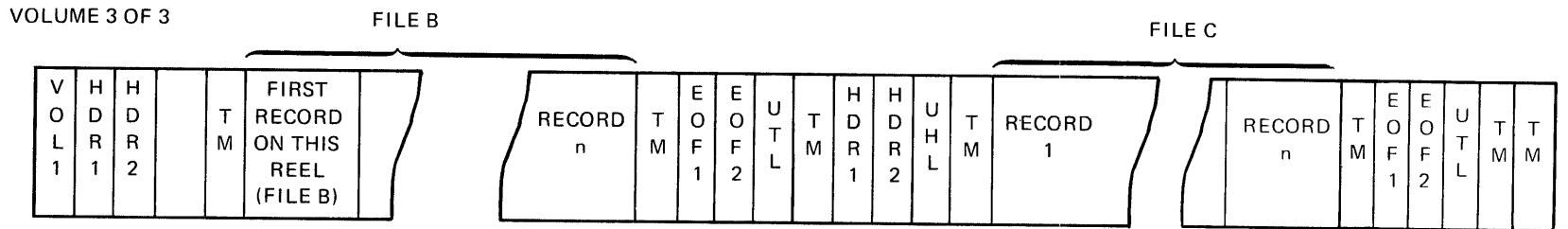
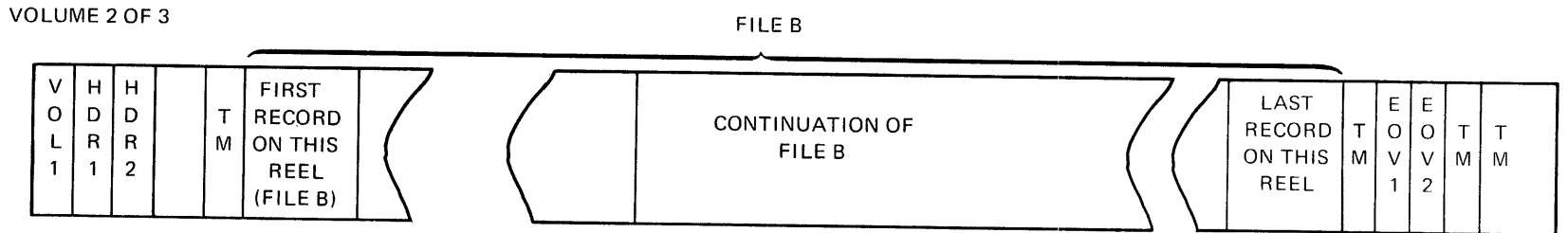
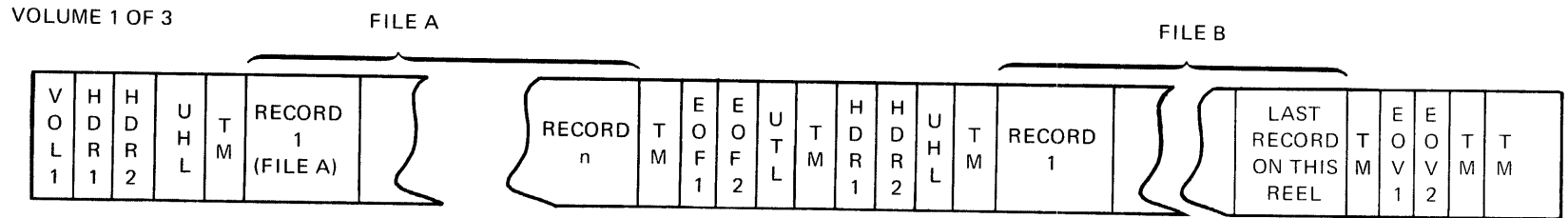
The file labels (file headers, end-of-file, and end-of-volume labels) are written on tape using the :FILE command or the FOPEN intrinsic. Each label is 80 bytes long and is formatted as shown in Figure D-1 and Table D-1.

User-supplied labels, if any, are located on the tape as shown in Figure D-1. User-supplied labels can only be written on tape labeled with MPE tape labels, and the user labels must be exactly 80 bytes, to conform to the MPE labels.



MULTIPLE FILES ON A SINGLE VOLUME

Note: When the file spans more than one volume, EOF is written instead of EOF.



MULTIPLE FILES ON MULTIPLE VOLUMES

Figure D-1. MPE Tape Labels (Conforming to ANSI-Standard)

Table D-1. Format of Header Tape Labels Written by MPE. (ANSI Standard)

VOLUME HEADER LABEL (80 BYTES)

POSITION	CONTENTS	COMMENTS
Bytes 1-4	<i>VOLn</i>	Indicates volume label ( <i>n</i> specifies the relative position of this label within the VOL type labels. MPE always uses 1.) Appears on each label.
Bytes 5-10	<i>volume id</i>	Six-character identifier as supplied by :FILE command, FOPEN intrinsic, or console operator.
Bytes 11-37	Blanks	Reserved for future use.
Bytes 38-51	Blanks	Not written by MPE. (Used for owner identification in ANSI-standard labels.)
Bytes 52-79	Blanks	Reserved for future use.
Byte 80	1	Indicates that label conforms to ANSI-standard.

FILE HEADER LABEL 1

Bytes 1-4	HDR1	Indicates file header 1 label. Appears before each file on the reel.
Bytes 5-21	<i>filename.groupname</i>	Used for file identifier in ANSI-standard labels.
Bytes 22-27	<i>volume set id</i>	Six-character identifier of the first volume in a set, as supplied by :FILE command, FOPEN intrinsic, or console operator.
Bytes 28-31	reel number	A four-digit entry from 0001 to 9999, indicating the relative position of a reel in a volume set.
Bytes 32-35	file sequence number	A four-digit entry from 0001 to 9999, indicating the relative position of a file on a reel.
Bytes 36-41	Blanks	Not written by MPE. (Reserved for generating data groups in ANSI-standard labels.)
Bytes 42-47	file creation date	Indicates date on which file is written to magnetic tape.
Bytes 48-53	file expiration date	Indicates date after which file can be overwritten.
Byte 54	%230	Indicates that file was created by MPE and the file has a lockword.
Bytes 55-60	Blanks	Reserved for future use.
Bytes 61-73	"HP MPE 3000 "	System code
Bytes 74-80	Blanks	Reserved for future use.

Table D-1. Format of Header Tape Labels Written by MPE. (ANSI Standard) Continued

FILE HEADER LABEL 2

POSITION	CONTENTS	COMMENTS
Bytes 1-4	HDR2	Indicates header 2 label . Appears after header 1.
Byte 5	record format	"F" = Fixed "V" = Variable "U" = Undefined
Bytes 6-10	block length	A five-digit entry, indicating block length in bytes.
Bytes 11-15	record length	A five-digit entry, indicating record length in bytes.
Bytes 16-23	lockword	MPE file lockword.
Bytes 24-36	Blanks	Reserved for future use.
Byte 37	record type	"A" = ASCII "B" = Binary
Byte 38	carriage control	"C" if control " " if no control
Bytes 39-80	Blanks	Reserved for future use.



# MPE DIAGNOSTIC MESSAGES

APPENDIX

E

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 LOCKGLORIN.

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 {<sup>=</sup>/<sub>#</sub>} <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 :OV.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 E-1. Program Error Messages

MESSAGE NO.	MESSAGE
1	INTEGER OVERFLOW
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

LOGIC ERROR IN THE PROGRAM.



Table E-2. Intrinsic Error Numbers (Continued)

MESSAGE NO.	INTRINSIC	MESSAGE NO.	INTRINSIC
54	XCONTRAP	84	EXPANDUSLF
55	RESETCONTROL	99	DEBUG
56	CAUSEBREAK	100	CREATE
60	TERMINATE	102	KILL
61	CTRANSLATE	103	SUSPEND
62	BINARY	104	ACTIVATE
63	ASCII	105	GETORIGIN
64	READ	106	MAIL
65	PRINT	107	SENDMAIL
66	PRINTOP	108	RECEIVEMAIL
67	PRINTOREPLY	109	FATHER
68	COMMAND	110	GETPROCINFO
69	WHO	112	GETPROCID
70	SEARCH	120	GETPRIORITY
71	MYCOMMAND	130	GETDSEG
72	SETJCW	131	FREEDSEG
73	GETJCW	132	DMOVIN
74	DBINARY	133	DMOVEOUT
75	DASCII	134	ALTDSEG
76	QUIT	135	DLSIZE
77	STACKDUMP	136	ZSIZE
78	SETDUMP	139	SWITCHDB
79	RESETDUMP	191	PTAPE
80	LOADPROC	200	GETPRIVMODE
81	UNLOADPROC	201	GETUSERMODE
82	INITUSLF	210	OPENLOG
83	ADJUSTSLF	211	WRITELOG
		212	CLOSELOG
		307	FERRMSG

Table E-3. Run-Time Error Messages

MESSAGE NO.	MESSAGE
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

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.

Table E-4.. File System Error Messages

END OF FILE (FSERR 0)  
 ILLEGAL DB REGISTER SETTING (FSERR 1)  
 ILLEGAL CAPABILITY (FSERR 2)  
 ILLEGAL PARAMETER VALUE (FSERR 8)  
 INVALID RECORD SIZE SPECIFICATION (FSERR 10)  
 INVALID RESULTANT BLOCK SIZE (FSERR 11)  
 RECORD NUMBER OUT OF RANGE (FSERR 12)  
 INVALID OPERATION (FSERR 20)  
 DATA PARITY ERROR (FSERR 21)  
 SOFTWARE TIME-OUT (FSERR 22)  
 END OF TAPE (FSERR 23)  
 UNIT NOT READY (FSERR 24)  
 NO WRITE-RING ON TAPE (FSERR 25)  
 TRANSMISSION ERROR (FSERR 26)  
 I/O TIME-OUT (FSERR 27)  
 TIMING ERROR OR DATA OVERRUN (FSERR 28)  
 SIO FAILURE (FSERR 29)  
 UNIT FAILURE (FSERR 30)  
 END OF LINE (FSERR 31)  
 SOFTWARE ABORT (FSERR 32)  
 DATA LOST (FSERR 33)  
 UNIT NOT ON-LINE (FSERR 34)  
 DATA-SET NOT READY (FSERR 35)  
 INVALID DISC ADDRESS (FSERR 36)  
 INVALID MEMORY ADDRESS (FSERR 37)  
 TAPE PARITY ERROR (FSERR 38)  
 RECOVERED TAPE ERROR (FSERR 39)  
 OPERATION INCONSISTENT WITH ACCESS TYPE (FSERR 40)  
 OPERATION INCONSISTENT WITH RECORD TYPE (FSERR 41)  
 OPERATION INCONSISTENT WITH DEVICE TYPE (FSERR 42)  
 WRITE EXCEEDS RECORD SIZE (FSERR 43)  
 UPDATE AT RECORD ZERO (FSERR 44)  
 PRIVILEGED FILE VIOLATION (FSERR 45)  
 OUT OF DISC SPACE (FSERR 46)  
 I/O ERROR ON FILE LABEL (FSERR 47)  
 INVALID OPERATION DUE TO MULTIPLE FILE ACCESS (FSERR 48)  
 UNIMPLEMENTED FUNCTION (FSERR 49)  
 NONEXISTENT ACCOUNT (FSERR 50)  
 NONEXISTENT GROUP (FSERR 51)  
 NONEXISTENT PERMANENT FILE (FSERR 52)  
 NONEXISTENT TEMPORARY FILE (FSERR 53)  
 INVALID FILE REFERENCE (FSERR 54)  
 DEVICE UNAVAILABLE (FSERR 55)  
 INVALID DEVICE SPECIFICATION (FSERR 56)  
 OUT OF VIRTUAL MEMORY (FSERR 57)  
 NO PASSED FILE (FSERR 58)  
 STANDARD LABEL VIOLATION (FSERR 59)  
 GLOBAL RIN UNAVAILABLE (FSERR 60)  
 OUT OF GROUP DISC SPACE (FSERR 61)  
 OUT OF ACCOUNT DISC SPACE (FSERR 62)  
 USER LACKS NON-SHARABLE DEVICE CAPABILITY (FSERR 63)  
 USER LACKS MULTI-RIN CAPABILITY (FSERR 64)  
 PUNCH HOPPER EMPTY (FSERR 65)  
 PLOTTER LIMIT SWITCH REACHED (FSERR 66)



Table E-4. File System Error Messages (Continued)

PAPER TAPE ERROR (FSERR 67)  
 INSUFFICIENT SYSTEM RESOURCES (FSERR 68)  
 I/O ERROR (FSERR 69)  
 TOO MANY FILES OPEN (FSERR 71)  
 INVALID FILE NUMBER (FSERR 72)  
 BOUNDS VIOLATION (FSERR 73)  
 NO-WAIT I/O PENDING (FSERR 77)  
 NO NO-WAIT I/O PENDING FOR ANY FILE (FSERR 78)  
 NO NO-WAIT I/O PENDING FOR SPECIAL FILE (FSERR 79)  
 SPOOFLE SIZE EXCEEDS CONFIGURATION (FSERR 80)  
 NO "SPOOL" CLASS IN SYSTEM (FSERR 81)  
 INSUFFICIENT SPACE FOR SPOOFLE (FSERR 82)  
 I/O ERROR ON SPOOFLE (FSERR 83)  
 DEVICE UNAVAILABLE FOR SPOOFLE (FSERR 84)  
 OPERATION INCONSISTENT WITH SPOOLING (FSERR 85)  
 NONEXISTENT SPOOFLE (FSERR 86)  
 BAD SPOOFLE BLOCK (FSERR 87)  
 SPOOLING ERROR (FSERR 88)  
 POWER FAILURE (FSERR 89)  
 EXCLUSIVE VIOLATION: FILE BEING ACCESSED (FSERR 90)  
 EXCLUSIVE VIOLATION: FILE ACCESSED EXCLUSIVELY (FSERR 91)  
 LOCKWORD VIOLATION (FSERR 92)  
 SECURITY VIOLATION (FSERR 93)  
 USER IS NOT CREATOR (FSERR 94)  
 READ COMPLETED DUE TO BREAK (FSERR 95)  
 DISC I/O ERROR (FSERR 96)  
 NO CONTROL Y PIN (FSERR 97)  
 READ TIME OVERFLOW (FSERR 98)  
 BOT AND BACKSPACE TAPE (FSERR 99)  
 DUPLICATE PERMANENT FILE NAME (FSERR 100)  
 DUPLICATE TEMPORARY FILE NAME (FSERR 101)  
 I/O ERROR ON DIRECTORY (FSERR 102)  
 PERMANENT FILE DIRECTORY OVERFLOW (FSERR 103)  
 TEMPORARY FILE DIRECTORY OVERFLOW (FSERR 104)  
 BAD VARIABLE BLOCK STRUCTURE (FSERR 105)  
 EXTENT SIZE EXCEEDS MAXIMUM (FSERR 106)  
 INSUFFICIENT SPACE FOR USER LABELS (FSERR 107)  
 INVALID FILE LABEL (FSERR 108)  
 INVALID CARRIAGE CONTROL (FSERR 109)  
 ATTEMPT TO SAVE PERMANENT FILE AS TEMPORARY (FSERR 110)  
 USER LACKS SAVE FILES (SF) CAPABILITY (FSERR 111)  
 USER LACKS PRIVATE VOLUMES (UV) CAPABILITY (FSERR 112)  
 VOLUME SET NOT MOUNTED – MOUNT PROBLEM (FSERR 113)  
 VOLUME SET NOT DISMOUNTED – DISMOUNT PROBLEM (FSERR 114)  
 ATTEMPTED RENAME ACROSS VOLUME SETS – REJECTED (FSERR 115)  
 INVALID TAPE LABEL FOPEN PARAMETERS (FSERR 116)  
 ATTEMPT TO WRITE ON AN UNEXPIRED TAPE FILE (FSERR 167)  
 INVALID HEADER OR TRAILER TAPE LABEL (FSERR 118)  
 I/O ERROR POSITIONING TAPE FOR TAPE LABELS (FSERR 119)  
 ATTEMPT TO WRITE IBM STANDARD TAPE LABEL (FSERR 120)  
 TAPE LABEL LOCKWORD VIOLATION (FSERR 121)  
 END OF TAPE VOLUME SET (FSERR 123)  
 INACTIVE RIO RECORD ACCESSED (FSERR 148)  
 MISSING ITEMNUM OR ITEMVALUE (149)

Table E-4. File System Error Messages (Continued)

INVALID ITEMNUM VALUE (150)  
 THE RECORD IS MARKED DELETED. FPOINT POSITIONED POINTER TO A RECORD THAT WAS  
 MARKED FOR DELETION (FSERR 170)  
 DUPLICATE KEY VALUE (FSERR 171)  
 NO SUCH KEY (FSERR 172)  
 TCOUNT PARAMETER LARGER THAN RECORD SIZE (FSERR 173)  
 CAN NOT GET EXTRA DATA SEGMENT (FSERR 174)  
 KSAM INTERNAL ERROR (FSERR 175)  
 ILLEGAL EXTRA DATA SEGMENT LENGTH (FSERR 176)  
 TOO MANY EXTRA DATA SEGMENTS FOR THIS PROCESS (FSERR 177)  
 EXTRA DATA SEGMENT TOO SMALL (FSERR 178)  
 THE FILE MUST BE LOCKED BEFORE ISSUING THIS INTRINSIC (FSERR 179)  
 THE KSAM FILE MUST BE REBUILT BECAUSE THIS VERSION OF KSAM DOES NOT HANDLE THE  
 FILE BUILT BY PREVIOUS VERSION (FSERR 180)  
 INVALID KEY STARTING POSITION (FSERR 181)  
 FILE IS EMPTY (FSERR 182)  
 RECORD DOES NOT CONTAIN ALL KEYS (FSERR 183)  
 INVALID RECORD NUMBER (FFINDN INTRINSIC) (FSERR 183)  
 SEQUENCE ERROR IN PRIMARY KEY (FSERR 185)  
 INVALID KEY LENGTH – NUMERIC DISPLAY OR PACKED DECIMAL (FSERR 186)  
 INVALID KEY SPECIFICATION (FSERR 187)  
 INVALID DEVICE SPECIFICATION (FSERR 188)  
 INVALID RECORD FORMAT (FSERR 189)  
 INVALID KEY BLOCKING FACTOR VALUE (FSERR 190)  
 RECORD DOES NOT CONTAIN SEARCH KEY FOR DELETION. SPECIFIED KEY VALUE POINTS TO  
 RECORD WHICH DOES NOT CONTAIN THAT VALUE. (FSERR 191)  
 SYSTEM FAILURE OCCURRED WHILE KSAM FILE WAS OPENED. (FSERR 192)  
 INVALID ID SEQUENCE (FSERR 201)  
 INVALID TELEPHONE NUMBER (FSERR 202)  
 NO TELEPHONE LIST SPECIFIED (FSERR 203)  
 INVALID ID SEQUENCE (FSERR 201)  
 INVALID TELEPHONE NUMBER (FSERR 202)  
 NO TELEPHONE LIST SPECIFIED (FSERR 203)  
 UNABLE TO ALLOCATE AN EXTRA DATA SEGMENT FOR DS/3000. (DSERR 204)  
 UNABLE TO EXPAND THE DS/3000 EXTRA DATA SEGMENT. (DSERR 205)  
 FILE NUMBER RETURNED FROM IOWAIT IS NOT A DS LINE NUMBER. (DSWARN 212)  
 THE REQUESTED DS LINE HAS NOT BEEN OPEN WITH A USER :DSLIN COMMAND. (DSERR 214)  
 MESSAGE REJECTED BY THE REMOTE COMPUTER. (DSERR 216)  
 INSUFFICIENT AMOUNT OF USER STACK AVAILABLE. (DSERR 217)  
 INVALID DS MESSAGE FORMAT. (INTERNAL DS ERROR) (DSERR 221)  
 THE LOCAL COMMUNICATION LINE HAS NOT BEEN OPENED BY THE OPERATOR. (DSERR 240)  
 THE DS LINE IS IN USE EXCLUSIVELY OR BY ANOTHER SUBSYSTEM. (DSERR 241)  
 INTERNAL DS SOFTWARE MALFUNCTION. (DSERR 242)  
 THE REMOTE COMPUTER IS NOT RESPONDING. (DSERR 243)  
 COMMUNICATIONS INTERFACE ERROR. THE REMOTE COMPUTER RESET THE LINE. (DSERR 244)  
 COMMUNICATIONS INTERFACE ERROR. RECEIVE TIMEOUT. (DSERR 245)  
 COMMUNICATIONS INTERFACE ERROR. REMOTE HAS DISCONNECTED. (DSERR 246)  
 COMMUNICATIONS INTERFACE ERROR. LOCAL TIME OUT. (DSERR 247)  
 COMMUNICATIONS INTERFACE ERROR. CONNECT TIME OUT. (DSERR 248)  
 COMMUNICATIONS INTERFACE ERROR. REMOTE REJECTED CONNECTION. (DSERR 249)  
 COMMUNICATIONS INTERFACE ERROR. CARRIER LOST. (DSERR 250)  
 COMMUNICATIONS INTERFACE ERROR. THE LOCAL DATA SET FOR THE DS LINE WHEN NOT  
 READY. (DSERR 251)

Table E-4.. File System Error Messages (Continued)

COMMUNICATIONS INTERFACE ERROR. HARDWARE FAILURE. (DSERR 252)  
COMMUNICATIONS INTERFACE ERROR. NEGATIVE RESPONSE TO THE DIAL REQUEST BY THE OPERATOR. (DSERR 253)  
COMMUNICATIONS INTERFACE ERROR. INVALID I/O CONFIGURATION. (DSERR 254)  
COMMUNICATIONS INTERFACE ERROR. UNANTICIPATED ERROR CONDITION. (DSERR 255)

Table E-5. Loader Error and Warning Messages

ILLEGAL LIBRARY SEARCH (LOAD ERR 20)  
UNKNOWN ENTRY POINT (LOAD ERR 21)  
TRACE SUBSYSTEM NOT PRESENT (LOAD ERR 22)  
STACK SIZE TOO SMALL (LOAD ERR 23)  
MAXDATA TOO LARGE (LOAD ERR 24)  
DATA SEGMENT TOO LARGE (LOAD ERR 25)  
PROGRAM LOADED IN OPPOSITE MODE (LOAD ERR 26)  
SL BINDING ERROR (LOAD ERR 27)  
INVALID SYSTEM SL FILE (LOAD ERR 28)  
INVALID PUBLIC SL FILE (LOAD ERR 29)  
INVALID GROUP SL FILE (LOAD ERR 30)  
INVALID PROGRAM FILE (LOAD ERR 31)  
INVALID LIST FILE (LOAD ERR 32)  
CODE SEGMENT TOO LARGE (LOAD ERR 33)  
PROGRAM USES MORE THAN ONE EXTENT (LOAD ERR 34)  
DATA SEGMENT TOO LARGE (LOAD ERR 35)  
DATA SEGMENT TOO LARGE (LOAD ERR 36)  
TOO MANY CODE SEGMENTS (LOAD ERR 37)  
TOO MANY CODE SEGMENTS (LOAD ERR 38)  
ILLEGAL CAPABILITY (LOAD ERR 39)  
TOO MANY PROCEDURES LOADED (LOAD ERR 40)  
UNKNOWN PROCEDURE NAME (LOAD ERR 41)  
INVALID PROCEDURE NUMBER (LOAD ERR 42)  
ILLEGAL PROCEDURE UNLOAD (LOAD ERR 43)  
ILLEGAL SL CAPABILITY (LOAD ERR 44)  
INVALID ENTRY POINT (LOAD ERR 45)  
UNABLE TO OPEN SYSTEM SL FILE (LOAD ERR 50)  
UNABLE TO OPEN PUBLIC SL FILE (LOAD ERR 51)  
UNABLE TO OPEN GROUP SL FILE (LOAD ERR 52)  
UNABLE TO OPEN PROGRAM FILE (LOAD ERR 53)  
UNABLE TO OPEN LIST FILE (LOAD ERR 54)  
UNABLE TO CLOSE SYSTEM SL FILE (LOAD ERR 55)  
UNABLE TO CLOSE PUBLIC SL FILE (LOAD ERR 56)  
UNABLE TO CLOSE GROUP SL FILE (LOAD ERR 57)  
UNABLE TO CLOSE PROGRAM FILE (LOAD ERR 58)  
UNABLE TO CLOSE LIST FILE (LOAD ERR 59)  
EOF OR I/O ERROR ON SYSTEM SL FILE (LOAD ERR 60)  
EOF OR I/O ERROR ON PUBLIC SL FILE (LOAD ERR 61)  
EOF OR I/O ERROR ON GROUP SL FILE (LOAD ERR 62)  
EOF OR I/O ERROR ON PROGRAM FILE (LOAD ERR 63)  
EOF OR I/O ERROR ON LIST FILE (LOAD ERR 64)  
UNABLE TO OBTAIN CST ENTRIES (LOAD ERR 65)  
UNABLE TO OBTAIN PROCESS DST ENTRY (LOAD ERR 66)  
UNABLE TO OBTAIN MAIL DATA SEGMENT (LOAD ERR 67)  
UNABLE TO CREATE LOAD PROCESS (LOAD ERR 68)  
SEGMENT TABLE OVERFLOW (LOAD ERR 70)  
UNABLE TO OBTAIN SUFFICIENT DL STORAGE (LOAD ERR 71)  
ATTIO ERROR (LOAD ERR 72)  
UNABLE TO OBTAIN VIRTUAL MEMORY (LOAD ERR 73)  
DIRECTORY I/O ERROR (LOAD ERR 74)  
PRINT I/O ERROR (LOAD ERR 75)  
ILLEGAL DLSIZE (LOAD ERR 76)  
PROGRAM ALREADY ALLOCATED (LOAD ERR 80)  
ILLEGAL PROGRAM ALLOCATION (LOAD ERR 81)  
PROGRAM NOT ALLOCATED (LOAD ERR 82)

Table E-5. Loader Error and Warning Messages (Continued)

ILLEGAL PROGRAM DEALLOCATION (LOAD ERR 83)  
PROCEDURE ALREADY ALLOCATED (LOAD ERR 84)  
ILLEGAL PROCEDURE ALLOCATION (LOAD ERR 85)  
PROCEDURE NOT ALLOCATED (LOAD ERR 86)  
ILLEGAL PROCEDURE DEALLOCATION (LOAD ERR 87)  
LMAP NOT AVAILABLE (LOAD WARN 88)  
PROGRAM LOADED WITH LIB=nnn (LOAD WARN 89)

Table E-6. CREATE Intrinsic Errors

UNKNOWN SUBQUEUE NAME (CREATE ERROR 20)  
SUBQUEUE 'A' REQUESTED WITHOUT FROZEN STACK (CREATE ERROR 21)  
INSUFFICIENT CAPABILITY FOR NON-STANDARD SUBQUEUE (CREATE ERROR 23)  
UNKNOWN PORTION OF MASTER QUEUE (CREATE ERROR 24)  
INSUFFICIENT CAPABILITY FOR MASTER QUEUE (CREATE ERROR 25)  
ABSOLUTE PRIORITY REQUESTED WITHOUT CAPABILITY (CREATE ERROR 26)  
ILLEGAL PRIORITY CLASS SPECIFIED (CREATE ERROR 27)  
PRIORITY OMITTED WHILE FATHER PROCESS IN MASTER QUEUE. (CREATE ERR 28)  
PRIORITY RANK RESERVED TO SUPERVISOR CAPABILITY (CREATE ERROR 29)  
LOAD ERROR (CREATE ERROR 30)  
LACK OF SYSTEM RESOURCE (CREATE ERROR 31)  
MAXIMUM ACCOUNT PRIORITY EXCEEDED (CREATE ERROR 32)

Table E-7. ACTIVATE Intrinsic Errors

ACTIVATION OF SYSTEM PROCESS NOT ALLOWED (ACTIVATE ERROR 20)  
ACTIVATION OF MAIN PROCESS NOT ALLOWED (ACTIVATE ERROR 21)

Table E-8. SUSPEND Intrinsic Error

INSUFFICIENT CAPABILITY (SUSPEND ERROR 20)

Table E-9. MYCOMMAND Intrinsic Error

PARSED PARAM OF COMIMAGE > 255 CHARACTERS

Table E-10. LOCKGLORIN Intrinsic Errors

INCORRECT PASSWORD FOR RIN  
ONLY ONE RIN CAN BE LOCKED  
RIN IS NOT ALLOCATED  
RIN IS TOO LARGE FOR THE RIN TABLE  
RIN IS NOT GLOBAL RIN

Table E-11. Private Volumes Error Messages

PRIVATE VOLUMES FACILITY NOT INVOKED (PVERR 20)  
 OPERATOR REJECTED MOUNT REQUEST (PVERR 21)  
 INSUFFICIENT DRIVES AVAILABLE TO MOUNT VOLUME SET (PVERR 22)  
 VOLUME SET TEMPORARILY IN USE BY SYSTEM (PVERR 23)  
 GROUP IN VOLUME SET SPECIFICATION DOES NOT EXIST (PVERR 24)  
 ACCOUNT IN VOLUME SET SPECIFICATION DOES NOT EXIST (PVERR 25)  
 VOLUME SET IS NOT PHYSICALLY MOUNTED ON SYSTEM (PVERR 26)  
 VOLUME SET/CLASS DEFINITION DOES NOT EXIST (PVERR 27)  
 NO HOME VOLUME SET DESIGNATED FOR nnn (PVERR 28)  
 GROUP IN HOME VOLUME SET SPECIFICATION DOES NOT EXIST (PVERR 29)  
 ACCOUNT IN HOME VOLUME SET SPECIFICATION DOES NOT EXIST (PVERR 30)  
 GROUP DOES NOT EXIST ON VOLUME SET (PVERR 31)  
 ACCOUNT DOES NOT EXIST ON VOLUME SET (PVERR 32)  
 VOLUME SET ALREADY MOUNTED BY THIS USER (PVERR 33)  
 USER DOES NOT HAVE VOLUME SET MOUNTED (PVERR 34)  
 OPERATOR DISMOUNT PENDING FOR VOLUME SET (PVERR 35)  
 DOWN PENDING FOR DISC CONTAINING MEMBER VOLUME (PVERR 36)  
 REQUEST FOR DIFFERENT MEMBERS THAN CURRENTLY IN USE (PVERR 37)  
 MUST USE HOME VOLUME SPECIFICATION IN THIS CONTEXT (PVERR 38)  
 CANNOT USE HOME VOLUME SET SPECIFICATION IN THIS CONTEXT (PVERR 39)  
 VOLUME SET CURRENTLY MOUNTED WITH DIFFERENT GENERATION (PVERR 41)  
 MOUNTED VOLUME TABLE ERROR (SYSTEM PROBLEM) (PVERR 50)  
 VOLUME SET USER TABLE ERROR (SYSTEM PROBLEM) (PVERR 51)  
 DIRECTORY ERROR (SYSTEM PROBLEM) (PVERR 52)  
 LDEV# nnn IS OUT OF RANGE (PVERR 60)  
 LDEV# nnn IS NOT CONFIGURED (PVERR 61)  
 LDEV# nnn IS NOT A DISC (PVERR 62)  
 LDEV# nnn IS NOT A REMOVABLE DISC (PVERR 63)  
 LDEV# nnn IS NOT AN ALLOWED REMOVABLE DISC-INVALID SUBTYPE (PVERR 64)  
 LDEV# nnn IS NOT IN THE USER DISC DOMAIN (PVERR 65)  
 LDEV# nnn IS NOT ON-LINE (PVERR 66)  
 LDEV# nnn IS A SERIAL DISC (PVERR 67)  
 LDEV# nnn IS RESERVED FOR USE BY SYSTEM (PVERR 68)  
 LDEV# nnn IS NOT DOWN-ED (PVERR 69)  
 LDEV# nnn HAS DOWN PENDING (PVERR 70)  
 LDEV# nnn IS IN USE BY PRIVATE VOLUMES (PVERR 71)  
 \*\* FUNCTION ABORTED \*\*  
 UNRECOGNIZED FUNCTION (PVERR 101)  
 INVALID TRACK DISPOSITION (PVERR 102)  
 UNRECOGNIZED KEYWORD (PVERR 103)  
 NO VOLUME SET CURRENTLY SPECIFIED (PVERR 104)  
 LDEV# nnn NOT DOWNED (PVERR 105)  
 LDEV# nnn NOT DOWNED OR SCRATCH (PVERR 106)  
 VOLUME SPECIFIED IS NOT A MEMBER OF THE VOLUME SET (PVERR 107)  
 SUBTYPE INCONSISTENCY BETWEEN DEVICES SPECIFIED (PVERR 108)  
 PACK SIZE INCONSISTENCY BETWEEN DEVICES SPECIFIED (PVERR 109)  
 ATTEMPTED TO COPY TO A BAD TRACK (PVERR 110)  
 NO SUSPECT TRACKS FOUND  
 NO ALTERNATE TRACKS AVAILABLE (PVERR 112)  
 TRACK NOT REASSIGNED  
 TRACK IN RESERVED AREA – MUST REFORMAT PACK (PVERR 114)  
 INVALID NUMERIC VALUE FOR KEYWORD PARAMETER (PVERR 115)  
 VOLUME NAME MORE THAN EIGHT CHARACTERS IN LENGTH (PVERR 116)

Table E-11. Private Volumes Error Messages (Continued)

VOLUME SET NAME MORE THAN EIGHT CHARACTERS IN LENGTH (PVERR 117)
KEYWORD IS MORE THAN EIGHT CHARACTERS IN LENGTH (PVERR 118)
VOLUME NAME HAS NON-ALPHA LEADING CHARACTER (PVERR 119)
VOLUME SET NAME HAS NON-ALPHA LEADING CHARACTER (PVERR 120)
KEYWORD HAS NON-ALPHA LEADING CHARACTER (PVERR 121)
VOLUME NAME CONTAINS A SPECIAL CHARACTER (PVERR 122)
VOLUME SET NAME CONTAINS A SPECIAL CHARACTER (PVERR 123)
KEYWORD CONTAINS A SPECIAL CHARACTER (PVERR 124)
NO PARAMETERS ALLOWED FOR THIS FUNCTION (PVERR 125)
MISSING NON-OPTIONAL PARAMETER (PVERR 126)
NON-NUMERIC CHARACTER IN LDEV SPECIFICATION (PVERR 127)
INVALID LDEV VALUE (PVERR 128)
TOO MANY NAMES IN VOLUME SET SPECIFICATION (PVERR 129)
MISSING KEYWORD PARAMETER VALUE (PVERR 130)
UNEXPECTED DELIMITER (PVERR 131)
UNEXPECTED PARAMETER (PVERR 132)
WARNING: VOLUME ALREADY nnn (PVWARN 140)
WARNING: SUSPECT TRACK IN ALTERNATE AREA (PVWARN 141)
WARNING: SUSPECT TRACK IN RESERVED AREA (PVWARN 142)
WARNING: nnn SUSPECT TRACKS DETECTED (PVWARN 143)

Table E-12. User Logging Error Messages

MESSAGE NO.	MESSAGE
0	No error occurred for this call.
1	User requested no wait mode and the logging process is busy.
2	Parameter out of bounds in logging intrinsic
3	Request to open or write to a logging process that isn't running.
4	Incorrect index parameter passed a logging intrinsic
5	Incorrect mode parameter passed to logging intrinsic
6	User request denied because logging process is suspended.
8	incorrect password passed to logging intrinsic.
9	Error occurred while writing logging file.
10	Invalid DST passed to logging system intrinsic.
12	System is out of disc space logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End of file on user log file.
16	Invalid logging identifier.

Table E-13. CLEANUSL Error Messages

ERROR NUMBER	MESSAGE
0	The file specified by uslfnum was empty, or unexpected end-of-file was encountered when reading the old uslfnum, or an unexpected end-of-file was encountered when writing on the new uslfnum.
1	Unexpected input/output error occurred. This can occur on the old uslfnum or the new uslfnum 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.
12	Illegal USL file format.



Table E-14. CREATEPROCESS Error Messages

ERROR NUMBER	MESSAGE
0	Process created as requested.
1	Caller lacks PH capability.
2	Required parameter (other than error) omitted.
3	Parameter address (other than error) out of bounds.
4	Out of system resources (PCB's, DST's, etc.)
5	Process not created because an invalid item number was specified.
6	Process not created because progname does not exist.
7	Process not created because progname is invalid.
8	Process not created because entryname does not exist or is invalid.
- 9	Process created with default stacksize from progfile (specified stacksize was <512).
-10	Process created with default dsize from progfile (specified dsize was <0).
-11	Process created with default maxdata from progfile (specified maxdata was <= 0).
-12	Process created with dsize rounded up to next 128 word multiple.
-13	Process created with maxdata decreased to configuration maximum.
-14	Process created with maxdata increased to dsize+globsize+stacksize (globsize is defined to be primary DB space+secondary DB space).
15	Process not created because dsize+globsize+stacksize was > configuration maximum stacksize.
16	Process not created because a "hard" load error occurred (e.g. I/O error reading progfile, etc.).
17	Process not created because an illegal value was specified for priority-class.
18	Process not created because specified \$STDIN could not be opened.
19	Process not created because specified \$STDLIST could not be opened.
20	Process not created because string to be passed to new process was invalid (pointer without length, length without pointer, or length exceeds stack size of calling process).

## 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:

$$FROM/ \left\{ \begin{array}{l} J \\ S \end{array} \right\} num, username.acctname/message$$

$\left. \begin{array}{l} jsname \\ username \\ acctname \end{array} \right\}$  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 under an account named MPE, 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/S106 BOB.MPE/DO NOT USE FILE TR7

## 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:

*OPERATOR WARNING/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 E-11. These messages may appear, asynchronously, during the course of a running job/session on the standard list device.

Table E-15. System Messages

CAN'T 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.

WARNING: PRIORITY = XXX

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.

LMAP NOT AVAILABLE

An LMAP of the process being created, or program file being :RUN, is not available because the code segments are already loaded.

\*\*POWER FAIL\*\*

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.



The lines indicated show the following information:

Line	Content
1	The <i>file name</i> : in this case, the name is IN.VOLLMER.CLIFTON
2	The <i>foptions</i> in effect, including:  Domain:       NEW = New file (as in this case). SYS = System file domain. JOB = Job temporary file domain. ALL = System <i>and</i> job temporary file domain.  Type:         A = ASCII File (as in this case). B = Binary File.  Default File Designator: *FORMAL* = Actual file designator is same as formal file designator.  \$STDIN \$STDLIST \$STDINX \$NEWPASS \$OLDPASS \$NULL  Record Format: Fixed length. (as in this case) V = Variable length. U = Undefined length (as in this case). ? = Unknown format.  Carriage Control: N = None (as in this case). C = Carriage control character expected.  File Equation Option: FEQ = :FILE allowed (as in this case). DEQ = :FILE not allowed. Labeled Tape Option: T = Not a labeled tape (as in this case) L = Labeled tape
3	The <i>aoptions</i> in effect, including:  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.  Multi-record Option: SREC = Single record access (as in this case). MREC = Multi-record access.

Line	Contents
3 (Cont.)	Dynamic      NOLOCK    = No locking permitted (as in this case). Locking        LOCK        = Locking permitted. Option:  Exclusive     DEF        = Default specification (as in this case). Access        EXC        = Exclusive access allowed. Option:        SEA        = Semi-exclusive access allowed. SHR        = Sharable file. Buffering:     BUFFER = Automatic buffering (as in this case). NOBUFF = Inhibit buffering
4,5	The <i>Device Type</i> , <i>Device Subtype</i> , <i>LDEV (Logical Device Number)</i> , <i>DRT (Device Reference Table Entry Number)</i> and <i>Unit</i> of the device on which the file resides. (These are 0, 9, 2, 4, and 1 respectively, in this case.) If the file is a spoolfile, the LDEV will be a "virtual" rather than a physical device number. See <i>ldnum</i> under FGETINFO.
6	The <i>record size</i> and <i>block size</i> of the offending record, in <i>bytes</i> or <i>words</i> as noted. (In this case, these are both specified as 256 bytes.)
7	The <i>extent size</i> (of the current extent) and the <i>max extents</i> (maximum number of extents) allowed this file.
8	The <i>recptr</i> (current record pointer) and <i>reclimit</i> (limit on number of records in the file).
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</i> , <i>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.

## A

Aborting a process, 4-20  
 Aborting a program, 4-20  
 Accessing files, 10-8  
 Accessing files already in use, 10-12  
 Access-mode options for files, 10-91  
 Access mode, user, 4-10  
 Acquiring global RIN's, 6-2  
 Acquiring local RIN's, 6-8  
 ACCEPT intrinsics, 2-4  
 ACTIVATE errors, E-13  
 ACTIVATE intrinsic  
   specifications, 2-5  
   usage, 8-10  
 Activating an extra data segment, 8-10  
 Activating processes, 7-3  
 Actual file designator, 10-10  
 ADJUSTUSLF intrinsic, 2-7  
 Allocation of devices, 10-25, 10-26  
 Allocating a terminal, 5-24  
 ALTDSEG intrinsic  
   specifications, 2-9  
   usage, 8-15  
 Aoptions  
   bit summary, 2-51  
   description, 2-61  
 Arithmetic traps, 4-30  
 ARITRAP intrinsic  
   specifications, 2-11  
   usage, 4-30  
 Arrays, searching, 4-3  
 ASCII intrinsic  
   specifications, 2-12  
   usage, 4-10  
 AS subqueue, 9-7  
 Attributes, user, 4-10  
 Available file table (AFT), 10-17  
 Avoiding deadlocks, 7-13

## B

Back referencing files, 10-10  
 BEGINLOG intrinsics, 2-13a  
 BINARY intrinsic  
   specifications, 2-14  
   usage, 4-13  
 Block factor, 2-65, 10-3  
 Blocks, 10-2  
 Block size, 10-4  
 Block transfers, 5-22  
 Bounds check, 1-11  
 BS subqueue, 9-7  
 Buffering, 10-7

## C

Calculating disc space, 10-2  
 Calendar date, 4-44  
 CALENDAR intrinsic  
   specifications, 2-15  
   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, 5-7  
   summary, 2-90  
 Carriage-return characters, 5-2  
 CAUSEBREAK intrinsic  
   specifications, 2-16  
   usage, 4-19  
 Central processor run-timer, 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 Files, 3-9 through 3-12  
   Features of intrinsics, 3-10  
 Circular subqueues, 9-5  
 Classification of devices, 10-25  
 CLEANUSL intrinsic  
   specifications, 2-17  
 CLOCK intrinsic  
   specifications, 2-18  
   usage, 4-44  
 CLOSELOG intrinsic  
   specifications, 2-19  
   usage, 10-93  
 Closing a new file as a permanent file, 10-40  
 Closing a new file as a temporary file, 10-39  
 Closing files, 10-39  
 Closing magnetic tape files, 10-73  
 COBOL II  
   FDELETE, 2-61  
   FFILEINFO, 2-63  
   Relative I/O, 10-6  
 Code segments, 8-1  
 Collecting mail, 7-12  
 COMMAND intrinsic  
   specifications, 2-20  
   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, 1-11
  - file system, 10-12
- Console Operator, 4-18
- Control-Y traps, 4-38
- Converting characters from EBCDIC 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
- Copy access, for message files, 3-3, 3-8
- CREATE errors, E-13
- CREATE intrinsic
  - specifications, 2-21
  - usage, 7-3
- CREATEPROCESS intrinsic, 2-26
  - errors, E-17
- Creating an extra data segment, 8-2
- Creating processes, 7-3
- CS subqueue, 9-7
- CTRANSLATE intrinsic
  - specifications, 2-28
  - usage, 4-13
- Current time, 4-44

## D

- DASCII intrinsic
  - specifications, 2-30
  - usage, 4-13
- Data segments, 8-1
  - changing the size, 8-15
  - deleting, 8-15
  - transferring to and from, 8-15
- Data information, 4-42
- DATELINE intrinsic, 2-32
- DBINARY intrinsic
  - specifications, 2-33
  - usage, 4-13
- DB pointer, 9-5
- Deadlocks, 7-13
- DEBUG intrinsic, 2-34
- Declaring access-mode options for files, 10-91
- Defining line-termination characters for terminal input, 5-20
- Deleting an extra data segment, 8-15
- Deleting processes, 7-8
- DENSITY selection on labeled and unlabeled tapes, 10-89
- DENSITY Specification, FOPEN, 2-88a
- Determining father process, 7-14
- Determining interactive and duplicative file pairs, 10-92
- Determining process priority and state, 7-15
- Determining son processes, 7-15

- Determining source of process activation, 7-14
- Determining user's access mode attributes, 4-10
- Device access, 10-7
- Device allocation, 10-26
- Device characteristics, 5-1
- Device control (2680A Printer), 2-61a
- Device-dependent restrictions on files, 10-24
- Devices, 10-7
  - Devices, classification of, 10-25
- Direct-access file reading, 10-49
- Direct-access file writing, 10-51
- Disabling traps, 4-29
- Disc space, 10-2
- DLSIZE intrinsic
  - specifications, 2-35
  - usage, 4-22
- DL to DB area, 4-22
- DMOVIN intrinsic
  - specifications, 2-37
  - usage, 8-15
- DMOVOUT intrinsic
  - specifications, 2-39
  - usage, 8-15
- Domains, file, 10-7
- DS subqueue, 9-7
- Duplex mode, 5-11
- Duplicate file pairs, 10-92
- Dynamic loading and unloading of library procedures, 4-2

## E

- Effective Use of User Logging, 10-96
- 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, 10-72
- ENDLOG intrinsic, 2-40a
- ENVIRONMENT File
  - Specification, FOPEN, 2-88a
- Entering non-privileged mode, 9-5
- Entering privileged mode, 9-3
- Error-check procedure, 10-45
- Errors
  - ACTIVATE errors, E-13
  - condition code, 1-10, 10-12
  - CREATE errors, E-13
  - CREATEPROCESS errors, E-17
  - file information display, E-20
  - file system errors, E-8
  - intrinsic errors, E-6
  - loader errors, E-12



- LOCKGLORIN errors, E-13
- MYCOMMAND errors, E-13
- obtaining file error information, 10-68
- operator messages, E-18
- program errors, E-5
- run-time errors, E-7
- run-time messages, E-2
- SUSPEND errors, E-13
- system messages, E-18
- user messages, E-18
- writing an error-check procedure, 10-45
- ES subqueue, 9-7
- Executing MPE commands programmatically, 4-9
- EXPANDUSLF intrinsic, 2-41
- Extended precision floating-point traps, 4-31
- Extents, 10-2
- Extra data segment, 8-2

## F

- FATHER intrinsic
  - specifications, 2-43
  - usage, 7-14
- Father process, 7-3
- FCARD intrinsic
  - specifications, 2-44
  - usage, 5-28
- FCHECK intrinsic
  - specifications, 2-48
  - usage, 10-69
- FCLOSE intrinsic
  - specifications, 2-54
  - usage, 10-38
- FCLOSE with magnetic tape files, 10-73
- FCONTROL intrinsic
  - specifications, 2-57
  - usage, 5-1, 10-79
- FDELETE intrinsic, 2-61
  - specifications, 2-61
  - usage, 10-9
- FDEVICEVONTROL intrinsic, 2-61a
- FERRMSG intrinsic, 2-62
  - usage, 10-69
- FFILEINFO intrinsic
  - specifications, 2-63
  - usage, 10-68
- FGETINFO intrinsic
  - specifications, 2-65
  - usage, 10-66
- File accessing, 10-12, 10-17
- File characteristics, 10-2
- File control operations, 10-90
- File designators, 10-10
- File-device relationships, 10-7
- File domains, 10-7
- File error information, 10-68
- File information display, E-20
- File label, 10-8
- File management system 10-1

- File marks on magnetic tape, 10-72
- File numbers, 10-12
- Files
  - access information, 10-66
  - accessing, 10-8
  - back referencing 10-10
  - block factor, 10-3
  - blocks, 10-2
  - block size, 10-4
  - buffered, 2-63
  - characteristics, 10-2
  - closing, 10-39
  - condition codes, 10-12
  - control, 10-90
  - declaring access-mode options, 10-91
  - designators, 10-8
  - device relationships, 10-7
  - disc, 10-2
  - domains, 10-7
  - duplicative pairs, 10-92
  - error-check procedure, 10-45
  - error information, 10-68
  - extents, 10-2
  - file information display, 10-14
  - file management system, 10-1
  - fixed-length records, 10-3
  - foreign disc facility, 10-32, 10-47, 10-49, 10-50, 10-51
  - how to use, 10-17
  - interactive pairs, 10-92
  - label, 10-8
  - locking and unlocking, 10-55
  - logical records, 10-2
  - magnetic tape, 10-69
  - multiple access, 10-12, 10-13
  - \$NEWPASS, 10-10
  - non-sharables devices, 10-15
  - no-wait I/O, 10-59, 10-60
  - \$NULL, 10-10
  - numbers, 10-12
  - \$OLDPASS, 10-11
    - opening, 10-27
  - permanent, 10-14, 10-40
  - physical records, 10-2
  - pointer, 10-91
  - reading, 10-47, 10-49
  - record formats, 10-3
  - records, 10-2
  - relative I/O, 10-6, 10-8
  - renaming, 10-43
  - sectors, 10-2
  - shared, special considerations, 10-16
  - spooling, 10-23
  - \$STDIN, 10-9
  - \$STDINX, 10-10
  - \$STDLIST, 10-10
  - system defined, 10-9
  - temporary, 10-7
  - types, 10-8
  - undefined-length records, 10-3

- unlocking, 10-55
- updating, 10-57
- user labels, 10-62
- user logging, 10-93
- using, 10-17
- variable-length records, 10-3
- writing, 10-49, 10-50
- Files on non-sharable devices, 10-50
- File system condition codes, 10-12
- File system errors, E-8
- File types, 10-8
- FINDJCW intrinsic
  - specifications, 2-74
  - usage, 4-47
- Fixed-length records, 10-3
- FLOCK intrinsic
  - specifications, 2-75
  - usage, 10-55
- FLUSHLOG intrinsic, 2-76a
- FMTCALENDAR intrinsic
  - specifications, 2-77
  - usage, 4-45
- FMTCLOCK intrinsic
  - specifications, 2-78
  - usage, 4-45
- FMTDATE intrinsic
  - specifications, 2-79
  - usage, 4-45
- FOPEN intrinsic
  - specifications, 2-80
  - usage, 10-26
- Foptions
  - bit summary, 2-49
  - description, 2-58
- Foreign Disc Facility, 10-32, 10-37, 10-49, 10-50, 10-51
- Formal file designator, 10-8
- Formatting Calendar date and time, 4-45
- Formatting command parameters, 4-4
- FPOINT intrinsic
  - specifications, 2-94
  - usage, 10-9
- FREAD and FWRITE for \$STDIN and \$STDLIST, 10-35
- FREAD for magnetic tape files, 10-71
- FREAD intrinsic
  - specifications, 2-95
  - usage, 10-47
- FREADBACKWARD intrinsic, 2-97
- FREADDIR intrinsic
  - specifications, 2-99
  - usage, 10-49
- FREADLABEL intrinsic
  - specifications, 2-101
  - usage, 10-63
- FREADSEEK intrinsic
  - specifications, 2-102
  - usage, 10-52
- FREEDSEG intrinsic
  - specifications, 2-103
  - usage, 8-15
- Freeing local RIN's 6-9
- FREELOCRIN intrinsic
  - specifications, 2-104
  - usage, 6-9
- FRELATE intrinsic
  - specifications, 2-105
  - usage, 10-92
- FRENAME intrinsic
  - specifications, 2-107
  - usage, 10-43
- FSETMODE intrinsic
  - specifications, 2-109
  - usage, 10-91
- FSPACE intrinsic
  - specifications, 2-111
  - usage, 10-89
- Functional return, 2-2
- FUNLOCK intrinsic
  - specifications, 2-113
  - usage, 10-56
- FUPDATE intrinsic
  - specifications, 2-114
  - usage, 10-58
- FWRITE and FREAD for \$STDLIST and \$STDIN, 10-35
- FWRITE for magnetic tape files, 10-71
- FWRITE intrinsic
  - specifications, 2-115
  - usage, 10-48
- FWRITEDIR intrinsic
  - specifications, 2-120
  - usage, 10-53
- FWRITELABEL intrinsic
  - specifications, 2-122
  - usage, 10-64

## G

- GENMESSAGE intrinsic
  - specifications, 2-123
  - usage, 4-50
- GET intrinsic, 2-126
- GETDSEG intrinsic
  - specifications, 2-127
  - usage, 8-6
- GETJCW intrinsic
  - specifications, 2-129
  - usage, 4-46
- GETLOCRIN intrinsic
  - specifications, 2-130
  - usage, 6-8
- GETORIGIN intrinsic
  - specifications, 2-131
  - usage, 7-14
- GETPRIORITY intrinsic
  - specifications, 2-132
  - usage, 7-13
- GETPRIVMODE intrinsic
  - specifications, 2-134
  - usage, 9-3

GETPROCID intrinsic  
 specifications, 2-135  
 usage, 7-15  
 GETPROCINFO intrinsic  
 specifications, 2-136  
 usage, 7-15  
 :GETRIN command, 6-2  
 GETUSERMODE intrinsic  
 specifications, 2-138  
 usage, 9-5  
 Global multiaccess, for circular files, 3-10  
 for message files, 3-3, 3-6  
 Global RIN's, 6-2

## H

HP 2680A Printer, 2-61a, 2-88a  
 Hand-shaking arrangement, 6-1  
 How to use files, 10-17  
 How user logging works, 10-94  
 Hand-shaking arrangement, 6-1  
 How to use files, 10-17  
 How user logging works, 10-94

## I

Identifying Local RIN owners, 6-9  
 INITUSLF intrinsic, 2-137  
 Input echo facility, 5-11  
 Input/output devices, 4-16  
 Input/output files, 10-11  
 Interactive file pairs, 10-92  
 Inter-job level RIN's, 6-2  
 Internal operations for file accessing, 10-17  
 Interprocess Communication (IPC), 3-1 through 3-21  
 additional features, 3-2  
 examples, 3-12  
 features of intrinsics, 3-5  
 operation, 3-1  
 using, 3-3  
 Inter-process level RIN's, 6-6  
 Inter-record gap, 10-76  
 Intrinsic errors, 10-6  
 Intrinsics  
 ACCEPT, 2-4  
 ACTIVATE, 2-5, 8-10  
 ADJUSTUSLF, 2-7  
 ALTDSEG, 2-9, 8-16  
 ARITRAP, 2-11, 4-30  
 ASCII, 2-12, 4-10  
 BEGINLOG, 2-13a  
 BINARY, 2-14, 4-13  
 CALENDAR, 2-15, 4-44  
 calling from other languages, 1-10  
 calling from SPL, 1-2  
 CAUSEBREAK, 2-16, 4-19  
 CLEANUSL, 2-17  
 CLOCK, 2-18, 4-44  
 CLOSELOG, 2-19

COMMAND, 2-20, 4-9  
 CREATE, 2-21, 7-3  
 CREATEPROCESS, 2-26  
 CTRANSLATE, 2-28, 4-13  
 DASCII, 2-30, 4-13  
 DATELINE, 2-32  
 DBINARY, 2-33, 4-13  
 DEBUG, 2-34  
 declaration, 1-2  
 definition, 1-1  
 DLSIZE, 2-35, 4-22  
 DMOVIN, 2-37, 8-15  
 DMOVOUT, 2-39, 8-15  
 error definitions, 1-10  
 error messages, E-1  
 ENDLOG, 2-40a  
 EXPANDUSLF, 2-41  
 FATHER, 2-43, 7-14  
 FCARD, 2-43, 5-28  
 FCHECK, 2-44, 2-48, 10-70  
 FCLOSE, 2-54, 10-73  
 FCONTROL, 2-57, 5-1, 10-79  
 FDELETE, 2-61, 10-9  
 FDEVICECONTROL, 2-61a  
 FERRMSG, 2-62, 10-69  
 FFILEINFO, 2-63, 10-68  
 FGETINFO, 2-65, 10-66  
 FINDJCW, 2-74, 4-47  
 FLOCK, 2-75, 10-55  
 FLUSHLOG, 2-76a  
 FMTCALNDAR, 2-77, 4-45  
 FMTCLOCK, 2-78, 4-45  
 FMTDATE, 2-79, 4-45  
 FOPEN, 2-80, 10-25  
 for Interprocess Communication, 3-5  
 for Circular Files, 3-10  
 FPOINT, 2-94, 10-9  
 FREAD, 2-95, 10-47  
 FREADBACKWARD, 2-97  
 FREADDIR, 2-99, 10-49  
 FREADLABEL, 2-101, 10-63  
 FREADSEEK, 2-103, 10-52  
 FREEDSEG, 2-103, 8-15  
 FREEOCRIN, 2-104, 6-9  
 FRELATE, 2-105, 10-92  
 FRENAME, 2-107, 10-43  
 FSETMODE, 2-109, 10-91  
 FSPACE, 2-111, 10-89  
 FUNLOCK, 2-113, 10-56  
 FUPDATE, 2-114, 10-58  
 FWRITE, 2-115, 10-48  
 FWRITEDIR, 2-120, 10-53  
 FWRITELABEL, 2-122, 10-64  
 GENMESSAGE, 2-123, 4-50  
 GET, 2-126  
 GETDSEG, 2-127, 8-6  
 GETJCW, 2-129, 4-46  
 GETLOCRIN, 2-130, 6-8  
 GETORIGIN, 2-131, 7-14

GETPRIORITY, 2-132, 7-13  
 GETPRIVMODE, 2-134, 9-3  
 GETPROCID, 2-135, 7-15  
 GETPROCINFO, 2-136, 7-15  
 GETUSERMODE, 2-138, 9-5  
 INITUSLF, 2-125  
 IODONTWAIT, 2-140, 10-62  
 IOWAIT, 2-142, 10-59  
 KILL, 2-144, 7-8  
 LOADPROC, 2-145, 4-2  
 LOCKGLORIN, 2-146, 6-3  
 LOCKLOCRIN, 2-148, 6-8  
 LOCRINOWNER, 2-150, 6-9  
 LOGSTATUS, 2-150a  
 MAIL, 2-151, 7-10  
 MYCOMMAND, 2-153, 4-4  
 OPENLOG, 2-156, 3-94  
 PAUSE, 2-157, 4-19  
 PCHECK, 2-158  
 PCLOSE, 2-159  
 PCONTROL, 2-160  
 POPEN, 2-161  
 PREAD, 2-162  
 PRINT, 2-163, 4-16  
 PRINTFILEINFO, 2-164, 10-45, 10-47  
 PRINTOP, 2-165, 4-18  
 PRINTOPREPLY, 2-166, 4-18  
 PROCTIME, 2-168, 4-44  
 PTAPE, 2-169, 5-27  
 purposes, 1-1  
 PUTJCW, 2-170, 4-47  
 PWRITE, 2-171  
 QUIT, 2-172, 4-20  
 QUITPROG, 2-173, 4-22  
 READ, 2-174, 4-16  
 READX, 2-175, 4-16  
 RECEIVEMAIL, 2-176, 7-12  
 REJECT, 2-178  
 RESETCONTROL, 2-179, 4-40  
 RESETDUMP, 2-180  
 SEARCH, 2-181, 4-3  
 SENDMAIL, 2-182, 7-11  
 SETDUMP, 2-183  
 SETJCW, 2-185, 4-46  
 STACKDUMP, 2-186  
 summary, 1-3  
 SUSPEND, 2-188, 7-8  
 SWITCHDB, 2-189, 9-5  
 TERMINATE, 2-190, 4-20  
 TIMER, 2-191, 4-42  
 types, 2-2  
 UNLOADPROC, 2-192, 4-3  
 UNLOCKGLORIN, 2-193, 6-3  
 UNLOCKLOCRIN, 2-194, 6-8  
 WHO, 2-195, 4-10  
 WRITELOG, 2-198, 10-94  
 XARITRAP, 2-199, 4-32  
 XCONTRAP, 2-201, 4-41  
 XLIBTRAP, 2-202, 4-35

XSYSTRAP, 2-203, 4-36  
 ZSIZE, 2-204, 4-27  
 IODONTWAIT intrinsic  
   specification, 2-140  
   usage, 10-62  
 IOWAIT intrinsic  
   specifications, 2-142  
   usage, 10-59  
 IPC — see Interprocess Communication  
 Issuing FREAD and FWRITE calls for  
   \$STDIN and \$STDLIST, 10-35

## J

Job control words, 4-47  
 Job main process, 7-1  
 Job or session file domains, 10-7  
 Job/session input/output devices, 4-16  
 Job temporary file directory, 10-20  
 Julian calendar, 8-8

## K

Keys, terminals, 5-9  
 KILL intrinsic  
   specifications, 2-144  
   usage, 7-8

## L

Labeled magnetic tape file  
   density selection, 10-89  
   opening, 10-81  
   reading, 10-87  
   writing, 10-84  
 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, E-12  
 Loading library procedures, 4-2  
 LOADPROC intrinsic  
   specifications, 2-145  
   usage, 4-2  
 Local RIN's, 6-6  
 Locking and unlocking files, 10-56  
 Locking and unlocking global RIN's, 6-3  
 Locking and unlocking local RIN's, 6-8  
 LOCKGLORIN errors, E-13  
 LOCKGLORIN intrinsic  
   specifications, 2-146  
   usage, 6-3  
 LOCKLOCRIN intrinsic  
   specifications, 2-148  
   usage, 6-8  
 LOCRINOWNER intrinsic

- specifications, 2-151
- usage, 6-9
- Logging, user, 10-93
  - BEGINLOG, 2-13a
  - ENDLOG 2-40a
  - FLUSHLOG, 2-76a
  - LOGSTATUS, 2-150a
  - OPENLOG, 2-156
  - CLOSELOG, 2-19
  - WRITELOG, 2-198
- Logical index number, 8-2
- Logical record pointer, 10-91
- Logical records, 10-2
- LOGSTATUS intrinsic, 2-150a

## M

- Magnetic tape considerations for files, 10-69
- Magnetic tape labels, D-1
- Magnetic tape unit, 5-3
- Mailbox, 7-10
- MAIL intrinsic
  - specifications, 2-151
  - usage, 7-10
- MAKECAT program, 4-50
- Master queue, 9-5
- Message Files, 3-1 through 3-21
- Message system 4-48
  - message catalog, 4-48
  - MAKECAT program, 4-49
- Messages
  - operator, 10-18
  - run-time, E-2
  - system, 10-18
  - user, 10-18
- Moving the DB pointer, 9-5
- Multi-access, 10-12
- Multiple access of files, 10-13
- Multiple RIN optional capability, 6-1
- MYCOMMAND errors, E-13
- MYCOMMAND intrinsic
  - specifications, 2-153
  - usage, 4-4

## N

- New files, 10-10
- \$NEWPASS, 10-10
- Nondestructive read, for message files, 3-3, 3-8
- Non-sharable device access, 10-7
- Non-sharable devices, 10-15
- No-wait I/O, 10-62
- \$NULL, 10-10

## O

- Obtaining file access information, 10-66
- Obtaining file error information, 10-68
- 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 data, 4-44
- Obtaining the current time, 4-44
- Old files, 10-11
- \$OLDPASS, 10-11
- Opening a file on a device other than disc, 10-34
- Opening a new disc file, 10-38
- Opening an old disc file, 10-31
- Opening files, 10-27
- Opening \$STDIN, 10-36
- Opening \$STDLIST, 10-36
- OPENLOG intrinsic
  - specifications, 2-156
  - usage, 10-94
- Operator messages, E-18
- Operator's Console, 4-18
- Operator intervention,
  - tape labels, 10-75a
- Optical Mark Reader, 5-28
- Optional capabilities
  - 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

## P

- 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 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, 2-157
  - usage, 4-19
- Permanent files, 10-14, 10-40
- Permanently privileged programs, 9-1
- PCHECK intrinsic, 2-158
- PCLOSE intrinsic, 2-159
- PCONTROL intrinsic, 2-160
- Physical records, 10-2
- PIN, 7-1
- POPEN intrinsic, 2-161
- Positional parameters, 1-7

- PREAD intrinsic, 2-162
- Pre-defined files, 10-10
- PRINT intrinsic
  - specifications, 2-163
  - usage, 4-16
- PRINTFILEINFO intrinsic
  - specifications, 2-166
  - usage, 10-45, 10-47
- Printing reader/punch, 5-3
- PRINTOP intrinsic
  - specifications, 2-165
  - usage, 4-18
- PRINTOPREPLY intrinsic
  - specifications, 2-166
  - usage, 4-18
- Private data area, 7-1
- Private volumes errors, E-14
- Private volumes subsystem, 10-17
- 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, 10-17
- 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
- PROCINFO intrinsic
  - specifications, 2-167a
- PROCTIME intrinsic
  - specifications, 2-168
  - usage, 4-44
- Program errors, E-5

- Program label, 7-1
- Programmatic execution of MPE commands, 4-9
- PTAPE intrinsic
  - specifications, 2-169
  - usage, 5-27
- PUTJCW intrinsic
  - specifications, 2-170
  - usage, 4-47
- PWRITE intrinsic, 2-171

## Q

- Queues, 95
- QUIT intrinsic
  - specifications, 2-172
  - usage, 4-20
- QUITPROG intrinsic
  - specifications, 2-173
  - usage, 4-22

## R

- Reading a file in direct-access mode, 10-49
- Reading a file in sequential order, 10-47
- Reading a user-defined file label
  - on a labeled tape file, 10-89
- Reading input from \$STDIN and \$STDINX, 4-16
- Reading magnetic tape files, 3-70
- Reading paper tapes without X-OFF control, 5-27
- Reading the terminal input timer, 5-19
- Reading user file labels, 10-62
- READ intrinsic
  - specifications, 2-174
  - usage, 4-16
- READX intrinsic
  - specifications, 2-175
  - usage, 4-16
- Receiving mail, 7-12
- RECEIVEMAIL intrinsic
  - specifications, 2-176
  - usage, 7-12
- Record formats, 10-3
- Records, 10-2
- REJECT intrinsic, 2-178
- Relative I/O, 10-8
  - Block format, 10-3
  - FDELETE, 2-61
  - FFILEINFO, 2-63
- Releasing global RIN's, 6-3
- REJECT intrinsic, 2-178
- Releasing global RIN's, 6-3
- Renaming a file, 10-44
- Requesting a process break, 4-17
- Requesting a reply from the Operator's Console, 4-18
- Required parameters, 1-8
- Rescheduling process, 7-13
- RESETCONTROL intrinsic
  - specifications, 2-179
  - usage, 4-40

- RESETDUMP intrinsic, 2-180
- Resetting the logical record pointer, 10-91
- Resource identification number, 6-1
- Resource management, 6-1
- Resources, 6-1
- Returns, intrinsic, 2-2
- RIN, 6-1
- Run-time errors, E-7
- Run-time messages, E-2

## S

- Scheduling processes, 9-5
- Searching arrays, 4-3
- SEARCH intrinsic
  - specifications, 2-181
  - usage, 4-3
- Sectors, 10-2
- 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, 2-182
  - usage, 7-11
- Sequential access file reading, 10-47
- Sequential access file writing, 10-49
- Session main process, 7-1
- SETDUMP intrinsic, 2-184
- SETJCW intrinsic
  - specifications, 2-185
  - usage, 4-46
- Setting parity, 5-23
- Setting terminal type, 5-25
- Setting unedited terminal mode, 5-26
- Shared files, 10-16
- Son process, 7-3
- Source of file characteristics, 10-20
- Source of process activation, 7-14
- Spacing on disc or tape files, 10-89
- Special terminal keys, 5-8
- SPL
  - calling intrinsics from, 1-2
  - description, 1-1
- Split stack, 1-11, 2-3
- Spooling, 10-23

- Stack
  - changing size, 4-22
  - sizes, 4-22
  - split stack, 1-11, 2-3
- STACKDUMP intrinsic, 2-170
- Stack segment, 8-1
- Standard traps, 4-31
- \$STDIN, 10-9
- \$STDINX, 10-10
- \$STDLIST, 10-10
- Subqueues, 9-5
- Substates, 7-2
- Subsystem break function, 5-14
- Suggested Log File Uses, 3-98
- SUSPEND errors, E-13
- Suspending processes, 7-8
- SUSPEND intrinsic
  - specifications, 2-188
  - usage, 7-8
- Suspending the calling process, 4-19
- SWITCHDB intrinsic
  - specifications, 2-189
  - usage, 9-5
- System break function, 5-13
- System defined files, 10-9
- System file domain, 10-7
- System messages, 10-18, 10-19
- System procedures, 1-1
- Systems Programming Language
  - calling intrinsics from, 1-2
  - description, 1-1
- System timer, 4-42
- System traps, 4-35

## T

- Tape density, determining, 10-89b
- Tape label, writing, 10-84
- Tape labels, MPE, 10-75, D-1
- Tape-mode option, 5-15
- Temporarily privileged programs, 9-2
- Temporary files, 10-7
- Terminal and line printer carriage-control codes, 5-6
- Terminal input timer, 5-16
- Terminals, 5-8, 5-9
- Terminal speed, 5-10
- Terminating a process, 4-20
- TERMINATE intrinsic
  - specifications, 2-190
  - usage, 4-20
- Testing mailbox status, 7-10
- Time and data intrinsics, 4-42
- TIMER intrinsic
  - specifications, 2-191
  - usage, 4-42
- Time outs, for IPC, 3-3, 3-8
- 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, 10-8  
Types of procedures, 2-2

### U

Undefined length records, 10-3  
Unlabeled magnetic tape file, 10-77  
density selection, 10-89  
Unloading library procedures, 4-2  
UNLOADPROC intrinsic  
specifications, 2-192  
usage, 4-3  
UNLOCKGLORIN intrinsic  
specifications, 2-193  
usage, 6-3  
Unlocking files, 10-56  
Unlocking global RIN's, 6-3  
Unlocking local RIN's, 6-8  
UNLOCKLOCRIN intrinsic  
specifications, 2-194  
usage, 6-8  
Updating a file, 10-58  
Updating magnetic tape files, 10-75  
User block transfers, 5-22  
User-defined job control words, 4-47  
User file labels, 10-62  
User Logging Facility, 10-94  
User messages, E-18  
User pre-defined files, 10-10  
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, 10-4  
Using FFILEINFO, 10-68

Using FGETINFO, 10-66  
Using numeric values as parameters, 1-8  
Using the FCARD intrinsic  
to operate the HP 7260A Optical Mark Reader, 5-28  
Utility functions of intrinsics, 4-1

### V

Variable-length records, 10-3  
Virtual device directory, 10-15  
WHO intrinsic  
specifications, 2-195  
usage, 4-10  
Writer ID's, 3-3, 3-8  
Writing a file system error-check procedure, 10-46  
Writing a tape label, 10-84  
Writing on a magnetic tape file, 10-88  
Writing output to \$STDLIST, 4-18  
Writing output to the Operator's Console, 4-18  
Writing records into a file in direct-access mode, 10-51  
Writing records into a file in sequential order, 10-49  
Writing to magnetic tape files, 10-77  
Writing user file labels, 10-62  
WRITELOG intrinsic  
specifications, 2-198  
usage, 10-94

### X

XARITRAP intrinsic  
specifications, 2-199  
usage, 4-32  
XCONTRAP intrinsic  
specifications, 2-201  
usage, 4-41  
XLIBTRAP intrinsic  
specifications, 2-202  
usage, 4-35  
X-OFF control, 5-21  
XSYSTRAP intrinsic  
specifications, 2-203  
usage, 4-36

### Z

ZSIZE intrinsic  
specifications, 2-204  
usage, 4-27  
Z to DB area, 4-27



## READER COMMENT SHEET

HP 3000 Computer System  
MPE Intrinsic  
Reference Manual  
30000-90010                      December 1981

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

- Is this manual technically accurate?                      Yes  No  (If no, explain under Comments, below.)
- Are the concepts and wording easy to understand?                      Yes  No  (If no, explain under Comments, below.)
- Is the format of this manual convenient in size, arrangement, and readability?                      Yes  No  (If no, explain or suggest improvements under Comments, below.)

Comments:

---

**FROM:**

**DATE** \_\_\_\_\_

**Name** \_\_\_\_\_

**Company** \_\_\_\_\_

**Address** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

FOLD

FOLD



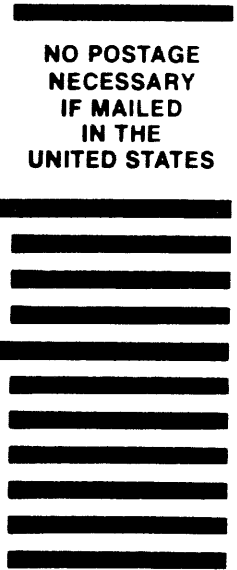
NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 1070 CUPERTINO, CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company  
Computer Systems Division  
MPE Documentation  
19447 Pruneridge Avenue  
Cupertino, California 95014



FOLD

FOLD

Part No. 30000-90010  
Printed in U.S.A. 1/81  
3MPE.320.30000-90010

