
HP 3000 Computer Systems
**MPE V Intrinsic
Reference Manual**



HP Part No. 32033-90007
Printed in U.S.A. 1989

Third Edition
E1089

HP Computer Museum
www.hpmuseum.net

For research and education purposes only.

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 use of this material.

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

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

Print History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	Jan 1985	E/F.00.00, G.00.00, G.01.00
Second Edition	Feb 1986	G.02.00
Update #1	Aug 1986	G.02.00, G.02.A0, G.02.B
Update #2	June 1987	G.02.00, G.02.A0, G.03.00
Update #3	Oct 1988	G.03.04
Third Edition	Oct 1989	G.03.08

MPE V Manual Plan

INTRODUCTORY LEVEL:

**GENERAL INFORMATION
Manual
(5953-7553)**

**MPE V/E HP 3000:
Fundamental Skills
(Part No. Pending)**

**HP 3000 GUIDE
FOR THE
NEW SYSTEM OPERATOR
(32033-90021)**

**MPE V/E HP 3000:
Advanced Skills
(Part No. Pending)**

**MPE V GENERAL USER'S
Reference Manual
(32033-90158)**

**FCOPY
Reference Manual
(32212-90003)**

STANDARD USER LEVEL:

**MPE V COMMANDS
Reference Manual
(32033-90006)**

**MPE V INTRINSICS
Reference Manual
(32033-90007)**

**MPE V SYSTEM UTILITIES
Reference Manual
(32033-90008)**

**MPE V SEGMENTER
Reference Manual
(30000-90011)**

**MPE V DEBUG/
STACK DUMP
Reference Manual
(30000-90012)**

**MPE V FILE SYSTEM
Reference Manual
(30000-90236)**

ADMINISTRATIVE LEVEL:

**MPE V SYSTEM
OPERATION AND
RESOURCE MANAGEMENT
Reference Manual
(32033-90005)**

**MPE V SECURITY AND
ACCOUNT STRUCTURE
(32033-90136)**

**MPE V STORING AND
RESTORING FILES
(32033-90133)**

**MPE V SYSTEM BACKUP
AND RECOVERY
(32033-90134)**

SUMMARY LEVEL:

**MPE V QUICK
REFERENCE GUIDE
(32033-90023)**

Preface

This manual documents the system-supplied intrinsics available with MPE (Multi-Programming Executive) on the HP 3000. It documents the intrinsics on the MPE V operating system and includes all changes and enhancements through the G.03.08 release of MPE.

Chapter 1	INTRODUCTION TO MPE INTRINSICS
	An overview of MPE V intrinsics, how to invoke them, their functions, and the optional capabilities required by system users to perform selected functions.
Chapter 2	INTRINSIC DESCRIPTIONS
	A detailed description of all MPE intrinsics, including the syntax, parameters, and condition codes for each. When applicable, functional returns, special considerations, and references to additional sources of information are included.
Chapter 3	OPTIONAL CAPABILITIES
	MPE functions which must be performed by users with specifically assigned optional capabilities.
Chapter 4	ACCESSING AND ALTERING FILES
	Background information on the MPE file system, including descriptions of some typical file-related operations performed with MPE intrinsics.
Chapter 5	OTHER APPLICATIONS OF MPE INTRINSICS
	How to perform the various functions available to users with MPE intrinsics. Examples of program dialogs are included for reference.
Appendix A	MPE DIAGNOSTIC MESSAGES
	The different types of interactive messages you might encounter while using MPE intrinsics.
Appendix B	DEVICE CHARACTERISTICS
	Details on how to alter the operation of specific peripheral devices using MPE intrinsics. The characteristics of several types of devices including terminals, printers, card readers, and magnetic tape are discussed.

Some additional sources of information you might find helpful include:

- *MPE V System Operation and Resource Management Reference Manual* (32033-90005).
- *MPE V Commands Manual* (32033-90006).
- *MPE V Utilities Reference Manual* (32033-90008).
- *MPE File System Reference Manual* (30000-90236).

Conventions

UPPERCASE

In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

COMMAND

can be entered as any of the following:

command Command COMMAND

It cannot, however, be entered as:

comm com_mand comamnd

italics

In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

COMMAND *filename*

bold italics

In a syntax statement, a word in bold italics represents a parameter that you must replace with the actual value. In the following example, you must replace ***filename*** with the name of the file:

COMMAND(***filename***)

punctuation

In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

(*filename*):(*filename*)

underlining

Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, yes is the user's response to the prompt:

Do you want to continue? >> yes

{ }

In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either ON or OFF:

COMMAND { ON
 OFF }

Conventions (continued)

[]

In a syntax statement, brackets enclose optional elements. In the following example, **OPTION** can be omitted:

COMMAND *filename* [**OPTION**]

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select **OPTION** or *parameter* or neither. The elements cannot be repeated.

COMMAND *filename* $\left[\begin{array}{l} \text{OPTION} \\ \text{parameter} \end{array} \right]$

[...]

In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

[*parameter*][...]

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:

[*parameter*][, ...]

| ... |

In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select **A**, **AB**, **BA**, or **B**. The elements cannot be repeated.

$\left\{ \begin{array}{l} \text{A} \\ \text{B} \end{array} \right\} | \dots |$

...

In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

Δ

In a syntax statement, the space symbol Δ shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

(*parameter*)Δ(*parameter*)

The symbol indicates a key on the keyboard. For example, RETURN represents the carriage return key or Shift represents the shift key.

CTRL*character*

CTRL*character* indicates a control character. For example, CTRL**Y** means that you press the control key and the Y key simultaneously.

Conventions (continued)

base prefixes

The prefixes %, #, and \$ specify the numerical base of the value that follows:

%num specifies an octal number.

#num specifies a decimal number.

\$num specifies a hexadecimal number.

If no base is specified, decimal is assumed.

bits (*bit:length*)

When a parameter contains more than one piece of data within its bit field, the different data fields are described in the format bits (*bit:length*), where *bit* is the first bit in the field and *length* is the number of consecutive bits in the field. For example, bits (13:3) indicates bits 13, 14, and 15:

```

most significant                                least significant
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0| | | | | | | | | | | | | | |13|14|15|
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
bits (0:1)                                bits (13:3)

```




Contents

1. Introduction to MPE Ininsics

Intrinsic Calls	1-1
Calling Intrinsic From SPL	1-2
Procedure Declarations.	1-2
Intrinsic Declarations.	1-2
Implementing Intrinsic Calls.	1-3
Calling Intrinsic From Languages Other Than SPL	1-5
Intrinsic Call Errors	1-6
Optional Capabilities	1-9
MPE Intrinsic and Their Functions	1-10

2. Intrinsic Descriptions

Intrinsic Name	2-1
Syntax	2-1
Functional Return	2-3
Parameters	2-4
Condition Codes	2-5
Special Considerations	2-5
Required Capability	2-5
Split-Stack Operations	2-6
Related Information	2-7
ABORTSESS	2-8
ACTIVATE	2-11
ADJUSTUSLF	2-13
ALTDSEG	2-15
ARITRAP	2-17
ASCII	2-19
BEGINLOG	2-21
BINARY	2-23
CALENDAR	2-24
CAUSEBREAK	2-25
CLEANUSL	2-26
CLOCK	2-27
CLOSELOG	2-28
COMMAND	2-31
CREATE	2-33
CREATEPROCESS	2-39
CTranslate	2-44
DASCI	2-46
DATELINE	2-48
DBINARY	2-49
DEBUG	2-50
DLSIZE	2-51
DMOVIN	2-53
DMOVOUT	2-55
DUMPPMAP	2-58

ENDLOG	2-60
EXPANDUSLF	2-63
FATHER	2-65
FCARD	2-66
FCHECK	2-71
FCLOSE	2-78
FCONTROL	2-82
FDELETE	2-89
FDEVICECONTROL	2-90
FERRMSG	2-98
FFILEINFO	2-99
FGETINFO	2-111
FINDJCW	2-120
FINDPMAPADDR	2-122
FINDPMAPNAME	2-124
FINTEXIT	2-126
FINTSTATE	2-127
FLABELINFO	2-129
FLOCK	2-134
FLUSHLOG	2-136
FMTCALENDAR	2-138
FMTCLOCK	2-139
FMTDATE	2-140
FOPEN	2-141
FPARSE	2-162
FPOINT	2-165
FREAD	2-166
FREADBACKWARD	2-169
FREADDIR	2-171
FREADLABEL	2-173
FREADSEEK	2-175
FREEDSEG	2-176
FREEOCRIN	2-178
FRELATE	2-179
FRENAME	2-181
FSETMODE	2-183
FSPACE	2-186
FUNLOCK	2-188
FUPDATE	2-189
FWRITE	2-191
FWRITEDIR	2-199
FWRITELABEL	2-201
GENMESSAGE	2-202
GETDSEG	2-207
GETINFO	2-210
GETJCW	2-212
GETLOCRIN	2-213
GETORIGIN	2-214
GETPRIORITY	2-215
GETPRIVMODE	2-217
GETPROCID	2-218
GETPROCINFO	2-219
GETUSERMODE	2-221
HPACDINFO	2-222
HPACDPUT	2-225

INITUSLF	2-232
IODONTWAIT	2-233
IOWAIT	2-235
JOBINFO	2-237
KILL	2-244
LOADPROC	2-245
LOCKGLORIN	2-247
LOCKLOCRIN	2-249
LOCRINOWNER	2-251
LOGINFO	2-252
LOGSTATUS	2-255
MAIL	2-257
MYCOMMAND	2-259
OPENLOG	2-263
PAUSE	2-266
PRINT	2-267
PRINTFILEINFO	2-269
PRINTOP	2-270
PRINTOPREPLY	2-271
PROCINFO	2-273
PROCTIME	2-278
PTAPE	2-279
PUTJCW	2-281
QUIT	2-283
QUITPROG	2-284
READ	2-285
READX	2-287
RECEIVEMAIL	2-289
RESETCONTROL	2-291
RESETDUMP	2-292
SEARCH	2-293
SENDMAIL	2-294
SETDUMP	2-296
SETJCW	2-297
STACKDUMP	2-298
STARTSESS	2-300
SUSPEND	2-305
SWITCHDB	2-307
TERMINATE	2-308
TIMER	2-309
UNLOADPROC	2-310
UNLOCKGLORIN	2-311
UNLOCKLOCRIN	2-312
WHO	2-313
WRITELOG	2-316
XARITRAP	2-318
XCONTRAP	2-321
XLIBTRAP	2-323
XSYSTRAP	2-324
ZSIZE	2-325

3. Optional Capabilities	
Privileged Mode Capability	3-1
Permanently Privileged Programs	3-1
Temporarily Privileged Programs	3-2
Entering privileged mode	3-3
Entering Non-privileged mode	3-5
Moving the DB Pointer	3-5
Scheduling Processes	3-6
Data Segment Management Capability	3-10
Creating an Extra Data Segment	3-11
Deleting an Extra Data Segment	3-25
Transferring Data from an Extra Data Segment to the Stack	3-25
Transferring Data from the Stack to an Extra Data Segment	3-25
Changing the Size of an Extra Data Segment	3-25
Process Handling Capability	3-26
Processes	3-27
Organization of User Processes	3-28
Active and Suspended Process Substates	3-28
Creating and Activating Processes	3-29
Suspending Processes	3-34
Deleting Processes	3-34
Interprocess Communication	3-36
Testing Mailbox Status	3-36
Sending Mail	3-37
Receiving (Collecting) Mail	3-38
Avoiding Deadlocks	3-39
Rescheduling Processes	3-39
Determining Source of Activation	3-40
Determining Father Process	3-40
Determining Son Processes	3-40
Determining Process Priority and State	3-41
Resource Management	3-42
Inter-Job Level (Global) RINS	3-43
Acquiring Global RINS	3-44
Releasing Global RINS	3-44
Locking and Unlocking Global RINS	3-45
Interprocess (Local) Level RINS	3-49
Acquiring Local RINS	3-49
Locking and Unlocking Local RINS	3-50
Identifying Local RIN Owners	3-51
Freeing Local RINS	3-52
User Logging	3-52
How User Logging Works	3-52
User Logging Procedures	3-59
Suggested Log File Uses	3-60

4. Accessing and Altering Files	
File Device Relationships	4-2
Non-Sharable Device Access	4-2
File Domains	4-2
Opening a File	4-4
Files on Non-Sharable Devices	4-5
How To Use Files	4-6
Internal Operations for File Accessing	4-7
Parsing and Validating File Designators	4-19
Opening a New Disk File	4-23
Opening an Old Disk File	4-26
Opening a File on a Device Other Than Disk	4-29
Using FREAD and FWRITE with \$STDIN and \$STDLIST	4-30
Opening \$STDIN	4-32
Opening \$STDLIST	4-33
Closing Files	4-36
Closing a New File as a Temporary File	4-36
Closing a New File as a Permanent File	4-39
Writing a File System Error-Check Procedure	4-41
Using FERRMSG	4-41
Using the IOWAIT Intrinsic	4-43
Declaring Access-Mode Options	4-47
5. Other Applications of MPE Intrinsics	
Dynamic Loading and Unloading of Library Procedures	5-2
Dynamic Loading	5-2
Dynamic Unloading	5-3
Searching Arrays	5-3
Formatting Command Parameters	5-5
Executing MPE Commands Programmatically	5-11
Determining the User's Access Mode and Attributes	5-12
Identifying a Job or Session with JOBINFO	5-14
Converting Numbers from Binary Code to ASCII Strings	5-15
Converting Numbers from an ASCII Numeric String to a Binary Coded Value	5-20
Translating Characters with the CTRANSLATE Intrinsic	5-22
Transmitting Program Input/Output from Job/Session	
Input/Output Devices	5-23
Reading Input from the Job/Session Input Device	5-23
Writing Output to the Job/Session List Device	5-25
Writing Output to the System Console	5-26
Writing Output to the System Console and Requesting a Reply	5-26
Suspending the Calling Process	5-26
Requesting a Process Break	5-27
Terminating a Process	5-28
Aborting a Process	5-28
Aborting a Program	5-29
Changing Stack Sizes	5-31
Changing the DL to DB Area Size	5-32
Changing the Z to DB Area Size	5-38
Enabling and Disabling Traps	5-39
Arithmetic Traps	5-39
Standard Traps	5-40
Extended Precision Floating Point Traps	5-40

Commercial Instruction Traps	5-41
Library Trap	5-45
System Trap	5-47
CONTROL-Y Traps	5-49
Time and Date Intrinsics	5-53
Obtaining System Timer Information	5-53
Obtaining the Current Time	5-54
Obtaining the Calendar Date	5-56
Obtaining Process Run Time	5-56
Formatting Calendar Date and Time Information	5-56
Job Control Words	5-57
Interprocess Communication	5-58
User-Defined Job Control Words	5-59
MPE Message Facility	5-60
Message Catalog	5-60
MAKECAT Program	5-61
Using GENMESSAGE to Insert Parameters in Messages	5-62
Application Message Facility	5-64
 A. MPE Diagnostic Messages	
Run-Time Messages	A-2
User Messages	A-10
Operator Messages	A-10
System Messages	A-11
File Information Display	A-12
 B. Device Characteristics	
Characteristics	B-1
Card Reader	B-1
Line Printer	B-1
Magnetic Tape	B-2
Line Printer and Terminal Carriage Control Codes	B-2
End-of-File Indication	B-2
Terminals	B-3
Using the FCARD Intrinsic With an HP 7260A Optical Mark Reader	B-3
ASCII and Column Image Reading Formats	B-4

Index

Figures

1-1. Calling the PRINTOP Intrinsic from SPL	1-5
1-2. Using Numeric Values as Parameters in an Intrinsic Call	1-5
1-3. Condition Code Checks	1-8
2-1. DMOVIN Data Movement.	2-54
2-2. DMOVOUT Data Movement.	2-56
2-3. Foption Bit Summary	2-109
2-4. Aoption Bit Summary	2-110
2-5. Foption Bit Summary	2-118
2-6. Aoption Bit Summary	2-118
2-7. Foption Bit Summary	2-133
2-8. Carriage Control Byte	2-168
2-9. Carriage Control Data	2-196
3-1. Using the GETPRIVMODE and GETUSERMODE Intrinsics (Program DSINIT)	3-4
3-2. Master Queue Structure	3-8
3-3. Using the GETDSEG and DMOVOUT Intrinsics (Program DSINIT)	3-14
3-4. Creating and Activating Two Son Processes (Program DSBOSS)	3-15
3-5. Using the CREATE and ACTIVATE Intrinsics (Program PROG)	3-31
3-6. Process Deletion	3-35
3-7. (Part 1)Using the LOCKGLORIN and UNLOCKGLORIN Intrinsics	3-46
3-8. BOOKFILE	3-47
3-9. User Logging Facility	3-54
4-1. File Access Interface for New Disk Files	4-9
4-2. Directory File Name and Label Address Pointer	4-10
4-3. File Access Interface for Old Disk Files	4-11
4-4. Device-Dependent Restrictions	4-14
4-5. Device Allocation Flowchart	4-17
4-6. Opening a New Disk File	4-25
4-7. Opening an Old Disk File	4-28
4-8. Opening a New Disk File	4-31
4-9. Opening \$STDIN and \$STDLIST	4-35
4-10. Closing a New File as a Temporary File	4-38
4-11. Closing a New File as a Permanent File	4-40
4-12. Error-Check Procedure Example	4-42
4-13. Using the IOWAIT Intrinsic	4-46
5-1. Using the MYCOMMAND Intrinsic (Program UTILY) (1 of 2)	5-6
5-2. Using the MYCOMMAND Intrinsic (Program UTILY) (2 of 2)	5-7
5-3. Using the WHO Intrinsic (1 of 2)	5-13
5-4. Using the WHO Intrinsic (2 of 2)	5-14
5-5. Using the ASCII Intrinsic (1 of 2)	5-17

5-6. Using the ASCII Intrinsic (2 of 2)	5-18
5-7. Using the DASCII Intrinsic	5-19
5-8. Using the BINARY Intrinsic (1 of 2)	5-21
5-9. Using the BINARY Intrinsic (2 of 2)	5-22
5-10. Using the PRINT and READ Intrinsics (1 of 2)	5-24
5-11. Using the PRINT and READ Intrinsics (2 of 2)	5-25
5-12. Using the QUIT Intrinsic (1 of 2)	5-30
5-13. Using the QUIT Intrinsic (2 of 2)	5-31
5-14. Expanding and Contracting the DL to DB Area	5-33
5-15. Using the DLSIZE Intrinsic (Program DLAREA)	5-34
5-16. Changing the DL to DB Area Size (Program DLAREA)	5-37
5-17. Using the SARITRAP Intrinsic (Program ATRAP)	5-44
5-18. Using the XCONTRAP Intrinsic (Program CONTY)	5-52
5-19. Using the TIMER Intrinsic (1 of 2)	5-55
5-20. Using the TIMER Intrinsic (2 of 2)	5-56
5-21. Using the FMTCALNDAR, FMTCLOCK, and FMTDATE Intrinsics	5-57
5-22. GENMESSAGE Intrinsic Example	5-65
B-1. FCARD Intrinsic Example (1 of 3)	B-6
B-2. FCARD Intrinsic Example (2 of 3)	B-7
B-3. FCARD Intrinsic Example (3 of 3)	B-8

Tables

1-1. Compatible Data Types	1-6
1-2. Summary of MPE Intrinsic	1-12
2-1. CREATEPROCESS Itemnum/Item Values	2-42
2-2. File System Error Codes	2-72
2-3. FCONTROL Itemnum/Item Values	2-83
2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values	2-91
2-5. FFILEINFO Itemnum/Item Values	2-100
2-6. FFILEINFO File codes	2-106
2-7. FLABELINFO Itemnum/Item Values	2-131
2-8. Foption Parameter Descriptions	2-143
2-9. Aoption Parameter Descriptions	2-147
2-10. Carriage Control Directives	2-194
2-11. HPACDINFO Itemnum/Item Values	2-223
2-12. HPACDPUT Itemnum/Item Values	2-226
2-13. ACD Status Returns	2-228
2-14. JOBINFO Itemnum/Item Values	2-240
2-15. LOGINFO Itemnum/Item Values	2-254
2-16. PROCINFO Error1,Error2 Descriptions	2-274
2-17. PROCINFO Itemnum/Item Values	2-275
4-1. Classification of Devices	4-15
A-1. PROGRAM TYPE Error Messages	A-5
A-2. "ERROR:INTRINSIC" Message Numbers	A-6
A-3. "RUN-TIME" Error Messages	A-7
A-4. "CREATE" Error Messages	A-7
A-5. "ACTIVATE" Error Messages	A-8
A-6. "SUSPEND" Error Messages	A-8
A-7. "MYCOMMAND" Error Messages	A-8
A-8. "LOCKGLORIN" Error Messages	A-8
A-9. "LOADER" Error and Warning Messages	A-8
A-10. "LOADER" Error and Warning Messages (Continued)	A-9
A-11. System Messages	A-11

Section Divider

1. Introduction to MPE Intrinsic



Introduction to MPE Ininsics

Many programs use procedures or subroutines to handle recurring tasks. In the Multi-Programming Executive (MPE, the HP 3000 operating system), many of these tasks are performed through a set of system-supplied procedures known as intrinsics. Since these intrinsics must always be available to MPE, they are also always available to any process on the system (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.

Most intrinsics are coded in SPL/3000 (Systems Programming Language for the HP 3000 Computer System) and are defined by a procedure declaration consisting of:

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

As part of their function, several intrinsics also return values to the processes that invoke them. These intrinsics are called type procedures.

Intrinsics are no different from procedures you may write yourself, except that the code is invisible to you, and they are declared (in SPL) with the **INTRINSIC** statement rather than the **PROCEDURE** statement.

Intrinsic Calls

Intrinsic calls programmatically invoke MPE intrinsics (that is, from within a program). In SPL programs (refer to “Calling Intrinsics From SPL” in this section), you can write the intrinsic calls explicitly or through procedure statements. Some languages, such as BASIC, COBOL II, FORTRAN, and Pascal, allow the option of directly calling MPE intrinsics. Within these languages the compiler will make calls to the intrinsics for you when you use input/output statements of the languages. Other languages such as APL, COBOL, and RPG will not allow direct calls. Within these languages intrinsic calls must be made through a subroutine written in a language that allows direct calls or through language commands that do the calls for you.

All MPE intrinsics are treated as external procedures by user programs. External linkages from user programs are satisfied when the user programs are segmented (via the `:PREP` command) and allocated residence in memory (at `:RUN` time). Refer to the *MPE Segmenter Reference Manual* (30000-90011) 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. The intrinsic can be declared in the same manner as any other SPL procedure or in an intrinsic declaration statement.

Procedure Declarations.

To declare an intrinsic as an SPL procedure add the **EXTERNAL** option to the procedure head and delete the procedure body. The **EXTERNAL** option means that the procedure body (code) is linked to the main program by the operating system after compilation. An example of calling the **DBINARY** intrinsic as a procedure follows:

```

**HEAD**
double
PROCEDURE DBINARY
    (dval,string,length);
    value length;
    double dval;
    byte array string;
    integer length;
    option external;
**BODY**

**TYPE**
**NAME**
**FORMAL PARAMETERS**
**VALUE PART**
**SPECIFICATION PART**
**OPTION PART**
```

Intrinsic Declarations.

Writing the complete head for some intrinsics can be very time-consuming, so a shortcut is provided in SPL/3000. Since SPL/3000 provides no construct for input and output, it provides a simple interface for intrinsics. This interface is the **INTRINSIC** declaration. The **INTRINSIC** declaration can be used with any system-known intrinsics defined in this manual. The format of the **INTRINSIC** declaration statement is:

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

In the *intrinsicname* list, you name all intrinsics to be called 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, enter:

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

Regardless of whether an intrinsic is declared as a procedure or in an **INTRINSIC** declaration statement, you must know the number and type of parameters the intrinsic uses in order to call it correctly.

Parameters can be passed to a procedure (intrinsic) either “by reference” or “by value”. When a parameter is passed by reference, 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.

Implementing Intrinsic Calls.

You call an intrinsic in your program exactly as you would call any SPL procedure, by entering the intrinsic name, followed by a parameter list enclosed in parentheses. These parameters must follow the positional format shown in each intrinsic description in Chapter 2 and must be separated by commas. For example, a call to the **FREAD** intrinsic could be written as:

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

If the Option Variable notation (0-V) appears in the intrinsic syntax as shown in Chapter 2, some of the intrinsic parameters are optional. However, since all intrinsic parameters are positional, 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 of an **FOPEN** call is omitted, you would write:

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

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

Input parameters, in some intrinsic calls, are passed to the intrinsic as words whose individual bits or fields of bits signify certain functions or options. Bit (0:1) is the high order (that is, most significant), left-most bit. Throughout this manual, bit groups are denoted using the standard SPL notation. Thus, bits (13:3) indicates bits 13, 14, and 15. 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 ensure the compatibility of your current program with future releases of MPE.

Output parameters, in some cases, are passed by an intrinsic to words referenced by a calling program. Bits within these words described as “reserved for MPE” are set to zero by the system 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 Chapter 2 to determine the parameter types and their positions in the parameter list.
2. Declare variables or arrays 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, the description of the **PRINTOP** intrinsic is shown in Chapter 2 as:

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

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

The mnemonics **LA**, **IV**, and **IV** over *message*, *length*, and *control* denote logical array (parameters are logical unless otherwise specified), integer by value, and integer by value, respectively. Refer to the beginning of Chapter 2 for a description of all mnemonics.

The array name to be used as the *message* parameter must be declared as an array at the beginning of the program. Similarly, the variables for *length* and *control* must be declared as integers.

Figure 1-1 shows the intrinsic **PRINTOP** being called from an SPL program after being declared with the **INTRINSIC** declaration statement. **MESSAGE** is an array, and the variables **LENGTH** and **CONTROL** are integers. The percent sign (%) means the value (') is to be treated as octal. The string is treated as a decimal value if it begins with a plus sign, a minus sign, or a number.

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

```

$CONTROL USLINIT
<< USING THE INTRINSIC DECLARATION STATEMENT >>
BEGIN
    ARRAY MESSAGE(0:9):="MESSAGE TO OPERATOR ";
    INTEGER LENGTH,CONTROL;
    INTRINSIC PRINTOP;
    LENGTH:=10;
    CONTROL:=%60;
    PRINTOP(MESSAGE,LENGTH,CONTROL);
END.

```

Figure 1-1. Calling the PRINTOP Intrinsic from SPL

```

$CONTROL USLINIT
<< USING NUMERIC VALUES AS PARAMETERS >>
BEGIN
    ARRAY MESSAGE(0:9):="MESSAGE TO OPERATOR ";
    INTRINSIC PRINTOP;
    PRINTOP(MESSAGE,10,%60);
END.

```

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

Calling Intrinsic From Languages Other Than SPL

Direct calls to intrinsics are allowed in some languages. These languages include BASIC, COBOL II, FORTRAN, and Pascal. To implement a call in these languages follow the steps used for calling an intrinsic from SPL. APL, COBOL, and RPG do not allow direct calls; to call an intrinsic from these languages you must call the intrinsic through a subroutine written in a language that allows direct calls. Additionally since most intrinsics are written in SPL, when a call is made from a program not written in SPL, careful consideration must be taken to ensure proper mapping of data types. Table 1-1 provides a listing of compatible data types between the languages that allow direct intrinsic calls and SPL. For more information on calling intrinsics from languages other than SPL refer to the appropriate language reference manual.

Table 1-1. Compatible Data Types

SPL	BASIC	COBOL II	FORTRAN	PASCAL
Integer	Integer	Computational 1-4 Digits	Integer/ Integer*2	Defined in the range -32768..32767
Logical	No Equivalent	Display	Logical	Boolean, or defined in the range -32768..32767
Byte	String (x\$)	Display	Character/ Character*n	Character
Double	No Equivalent	Computational 5-9 Digits	Integer*4	Integer
Real	Real	No Equivalent	Real	Real
Long	Long	No Equivalent	Long	Long real

Intrinsic Call Errors

Some intrinsics alter the “condition code” which is stored in two bits (6:2) in the Status Register. These two bits have four states which are assigned as follows:

Is CCG, or Condition Code Greater Than (>).

Is CCL, or Condition Code Less Than (<).

Is CCE, or Condition Code Equal (=).

Undefined.

Since bits (6:2) of the Status Register are affected by many instructions, check for condition codes immediately upon return from an intrinsic, as in the “IF” statements in Figure 1-3. A condition code is always CCG, CCL, or CCE, and has the general meaning indicated below. The specific meaning depends upon the intrinsic called (refer to Chapter 2 for a description of these meanings).

Condition Code State	SPL Branch Word	General Meaning
CCE	=	Condition Code Equal. This generally indicates that the request was granted.
CCG	>	Condition Code Greater Than. 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 Than. The request was not granted, but the error condition may be recoverable. Beyond this condition code, some intrinsics provide additional error information to the program through their return values or reference parameters.

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. An abort error terminates the calling process.

“Soft interrupts” are interrupts generated by software events, where the CPU is interrupted from processing and the operating system switches execution to an interrupt procedure. This sequence of events is known as a trap. A soft interrupt within MPE means that when I/O completes, the CPU will be interrupted. Scheduling information for the process that initiated the I/O will be updated and the next time the process runs it will trap to a predetermined interrupt procedure. 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, write a procedure into your program to be used instead of the default system trap handler in case of an abort error. This method will permit you to recover from such errors in certain cases. For more information on implementing traps, refer to Section 5, “Other Applications of MPE Intrinsics”.

When a program is aborted in a batch job, MPE removes the job from the system unless a :CONTINUE command, defined in the *MPE V Commands Reference Manual* (32033-90006), precedes the command which causes the error. If the program is aborted in an interactive session, MPE returns control to the terminal. Abort error messages are described in Appendix A, "MPE DIAGNOSTIC MESSAGES".

When 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. This is done to ensure that all parameters in the intrinsic call reference addresses 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, or, in the case of file systems intrinsics, the condition code is set to CCL and the program continues.

When an intrinsic is invoked by a process running in privileged mode and the DB register points to a data segment (i.e. the intrinsic is operating in split-stack mode), the results depend on the particular intrinsic. Most intrinsics abort immediately in this case. Others are allowed to execute following a bounds check that ensures that all parameters in the intrinsic call reference addresses that lie within the data segment. Any boundary violation results in an abort error.

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

```
$CONTROL USLINIT
<< CONDITION CODE CHECKS >>
BEGIN
  ARRAY MESSAGE(0:9):="MESSAGE TO OPERATOR ";
  ARRAY OKBUF(0:9):="MESSAGE TRANSMITTED ";
  ARRAY ERRBUF(0:9):="I/O ERROR OCCURRED ";
  INTRINSIC PRINTOP,PRINT;
  PRINTOP(MESSAGE,10,%60);
  IF = THEN
    PRINT(OKBUF,10,%60);
    GOTO STOP;
  IF < THEN
    PRINT(ERRBUF,9,%60);
  STOP:
  END.
```

Figure 1-3. Condition Code Checks

Optional Capabilities

Users with standard MPE capabilities 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. These optional capabilities are assigned by the System Manager when creating the user's account. The System Manager can alter the capabilities for any account, group, or user on the system with the :ALTACCT, :ALTGROUP, or :ALTUSER command.

Since many intrinsics require additional capabilities to work, the program which calls them must be prepared with these capabilities specified. The creator of the program must have a capability to assign it to a program. The user need not have a specific capability to run a program, but in order to run the program, it must reside in a group with the specific capability.

The MPE optional capabilities, and what they allow you to do, are explained below.

The process handling (PH) 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 data segment management (DS) capability allows you to create and access extra data segments from processes during a job or session.

The multiple resource identification number (MR) capability allows you to simultaneously lock as many global Resource Identification Numbers (RINs) as desired.

The privileged mode (PM) capability allows you to access all areas of the system and use all features of the hardware. With this capability you may access all system tables and invoke all system instructions, including those in the privileged central processor unit (CPU) instruction set. In short, this capability allows you to use the computer on the same terms as the operating system itself.

Caution



The normal checks and limitations that apply to 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. Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.

The user logging (LG) capability provides a flexible transaction-logging capability which allows you to journalize additions and modifications to your data bases and subsystem files. User logging permits you to journalize on either tape or disk. If the data base is lost, the logging tape or disk file can be used to recover the lost transactions.

Programmatic sessions (PS) capability allows programmatic creation of sessions on any terminal on the system (available on G.01.00 release or later).

MPE Intrinsic and Their Functions

MPE intrinsic allow you to access and alter files, request various utility functions, and access and manage system resources. When intrinsic are used with certain optional capabilities, it is possible to manipulate processes, data segments, and system resources.

To help you determine what tasks you can accomplish with MPE intrinsic, refer to Table 1-2, which lists each intrinsic, its purpose, and the capability needed to use it.

The volume set usage (UV) or create volumes (CV) capabilities allows you to maintain files on private disk volumes. If your file group has been structured to use the Private Volumes Subsystem, MPE checks to determine if your home volume set is mounted when you create a new disk file with the **FOPEN** intrinsic.

Similarly, when you close and save a disk file with the **FCLOSE** intrinsic, it is automatically stored on your home volume set if your account is structured with either of these capabilities.

(Refer to the *MPE V System Operation and Resource Management Reference Manual* (32033-90005), for more information on Private Volumes.) Optional capabilities are discussed in more detail in Section3.

Table 1-2. Summary of MPE Ininsics

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
ABORTSESS	Aborts the specified session from the system.	standard
ACTIVATE	Activates a process.	process handling (PH)
ADJUSTUSLF	Adjusts directory space in a USL file.	standard
ALTDSEG	Alters the size of an extra data segment.	data segment management (DS)
ARITRAP	Enables or disables internal interrupt signals from all hardware arithmetic traps.	standard
ASCII	Converts a one-word binary number to a numeric ASCII string.	standard
BEGINLOG	Marks the beginning of a user logging transaction.	user logging (LG) or system supervisor (OP)
BINARY	Converts a number from an ASCII string to a binary word.	standard
CALENDAR	Returns the calendar date.	standard
CAUSEBREAK	Places a session in BREAK mode.	standard
CLEANUSL	Deletes inactive entries from a USL file.	standard
CLOCK	Returns the actual time according to system timer.	standard
CLOSELOG	Closes access to the user logging facility.	User Logging (LG) or system supervisor (OP)
COMMAND	Executes an MPE command programmatically.	standard
CREATE	Creates a process.	process handling (PH)
CREATEPROCESS	Creates a process and can assign \$STDIN and \$STDLIST to any file.	process handling (PH)
CTranslate	Converts a string of characters between EBCDIC and ASCII or between EBCDIK and JIS (KANA8).	standard

Table 1-2. Summary of MPE Ininsics (continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
DASCII	Converts a double-word (32-bit) binary number to an ASCII string.	standard
DATELINE	Returns the current date and time.	standard
DBINARY	Converts a number from an ASCII string to a double-word binary value.	standard
DEBUG	Invokes the DEBUG facility.	standard
DLSIZE	Expands or contracts the area between DL and DB.	standard
DMOVIN	Copies data from an extra data segment into the stack.	data segment management (DS)
DMOVOUT	Copies data from the stack to an extra data segment.	data segment management (DS)
DUMPPMAP	Fills a file with external FPMAP data stored in a program or SL file, or only that for a specified segment.	data segment management (DS)
ENDLOG	Marks the end of a user logging transaction.	user logging (LG) or system supervisor (OP)
EXPANDUSLF	Changes length of a USL file.	standard
FATHER	Requests the Process Identification Number (PIN) of father process.	process handling (PH)
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 device.	standard
FDELETE	Deactivates a Relative I/O (RIO) record.	standard
FDEVICECONTROL	Provides control operations to a printer or a spooled devicefile.	standard

Table 1-2. Summary of MPE Ininsics (continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FERRMSG	Returns message corresponding to FCHECK error message.	standard
FFILEINFO	Provides access to file information.	standard
FGETINFO	Requests access and status information about a file.	standard
FINDJCW	Searches Job Control Word Table for named JCW.	standard
FINDPMAPADDR	Searches the FPMAP in a program or SL file for a specific address in a particular segment.	standard
FINDPMAPNAME	Searches the FPMAP in a program or SL file for a segment name or an entry point name.	standard
FINTEXTIT	Causes return from user's interrupt procedure.	standard
FINTSTATE	Enables/disables the software interrupt facility for a calling process.	standard
FLABELINFO	Returns information from the file label of a disk file.	standard
FLOCK	Dynamically locks a file. (If locking more than one file.)	standard (Multiple RIN (MR))
FLUSHLOG	Flushes contents of user logging memory buffer to logging file.	user logging (LG) or system supervisor (OP)
FMTCALENDAR	Formats specified calendar date.	standard
FMTCLOCK	Formats specified time of day.	standard
FMTDATE	Formats specified calendar date and time of day.	standard
FOPEN	Opens a file.	standard
FPARSE	Parses/validates file designators.	standard
FPOINT	Sets the logical record pointer for a disk file.	standard
FREAD	Reads logical record from file to user's stack.	standard

Table 1-2. Summary of MPE Ininsics (continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FREADBACKWARD	Reads logical record backward from current record pointer. Data is presented as if read forward.	standard
FREADDIR	Reads a specific logical record from a direct access file to the user's data stack.	standard
FREADLABEL	Reads a user file label.	standard
FREADSEEK	Moves a record from a disk file to a buffer in anticipation of a FREADDIR intrinsic call.	standard
FREEDSEG	Releases an extra data segment.	data segment management (DS)
FREELCRIN	Frees all local Resource Identification Numbers (RINs) from allocation to a job.	standard
FRELATE	Determines if file pair is interactive, duplicative, or both.	standard
FRENAME	Renames a disk file.	standard
FSETMODE	Activates or deactivates file access modes.	standard
FSPACE	Moves a physical record pointer forward or backward on a tape or disk file.	standard
FUNLOCK	Dynamically unlocks a file.	standard
FUPDATE	Updates (writes) a logical record in a disk file.	standard
FWRITE	Writes a logical or physical record or portion of a record from user's stack to a file on any device.	standard
FWRITEDIR	Writes a specific logical record from the user's stack to a disk file.	standard
FWRITELABEL	Writes a user file label.	standard
GENMESSAGE	Accesses the MPE message system.	standard

Table 1-2. Summary of MPE Intrinsic (continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
GETDSEG	Creates an extra data segment.	data segment management (DS)
GETINFO	Gets info/parm values from :RUN command or CREATEPROCESS intrinsic.	standard
GETJCW	Returns the value of the system-defined Job Control Word, JCW.	standard
GETLOCRIN	Acquires local RINs.	standard
GETORIGIN	Determines source of activation call for process.	process handling (PH)
GETPRIORITY	Changes the priority of a process.	process handling (PH)
GETPRIVMODE	Dynamically enters privileged mode.	privileged mode (PM)
GETPROCID	Requests PIN of a son process.	process handling (PH)
GETPROCINFO	Requests status data on a father or son process.	process handling (PH)
GETUSERMODE	Dynamically returns to non-privileged mode.	privileged mode (PM)
HPACDINFO	Lists security information from the Access Control Definition (ACD) of a specified file or device.	standard
HPACDPUT	Manipulates security information in the Access Control Definition (ACD) of a specified file or device.	standard
INITUSLF	Initializes a USL file to the empty state.	standard
IODONTWAIT	Initiates completion operations for an I/O request.	privileged mode (PM)
IOWAIT	Initiates completion operations for an I/O request.	privileged mode (PM)
JOBINFO	Returns job/session related information.	standard
KILL	Deletes a son process.	process handling (PH)

Table 1-2. Summary of MPE Ininsics (continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
LOADPROC	Dynamically loads a library procedure.	standard
LOCKGLORIN	Locks a global RIN.	standard
LOCKLOCRIN	Locks a local RIN.	standard
LOCRINOWNER	Determines PIN of a process locking a local RIN.	standard
LOGINFO	Obtains information from user logging buffer.	user logging (LG) or system supervisor (OP)
LOGSTATUS	provides information about an opened user logging file.	user logging (LG) or system supervisor (OP)
MAIL	Tests mailbox status.	process handling (PH)
MYCOMMAND	Parses (delineates and defines) parameters for user-supplied command image.	standard
OPENLOG	Provides access to the user logging facility.	user logging (LG) or system supervisor (OP)
PAUSE	Suspends process for a specific number of seconds.	standard
PRINT	Prints character string on job/session list device.	standard
PRINTFILEINFO	Prints a file information display on a job/session list device.	standard
PRINTOP	Prints a character string on the System Console.	standard
PRINTOPREPLY	Prints a character string on the System Console and solicits a reply.	standard
PROCINFO	Provides access to process information.	standard
PROCTIME	Returns the accumulated CPU time for a process.	standard
PTAPE	Copies input from paper tapes which do not contain X-OFF control characters to a disk file.	standard

Table 1-2. Summary of MPE Ininsics (continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
PUTJCW	Assigns the value of a particular JCW in JCW Table.	standard
QUIT	Aborts a process.	standard
QUITPROG	Aborts the entire user process structure.	standard
READ	Reads an ASCII string from \$STDIN into an array.	standard
READX	Reads an ASCII string from \$STDINX into an array.	standard
RECEIVEMAIL	Receives mail from another process.	process handling (PH)
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 (PH)
SETDUMP	Enables the stack analysis facility.	standard
SETJCW	Sets bits in the system Job Control Word, JCW.	standard
STACKDUMP	Dumps selected parts of stack to file.	standard
STARTSESS	Initiates a session on the specified terminal.	programmatic sessions (PS)
SUSPEND	Suspends a process.	process handling (PH)
SWITCHDB	Switches DB register pointer.	privileged mode (PM)
TERMINATE	Terminates a process.	standard
TIMER	Returns system timer information.	standard
UNLOADPROC	Dynamically unloads a library procedure.	standard

Table 1-2. Summary of MPE Ininsics (continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
UNLOCKGLORIN	Unlocks a global RIN.	standard
UNLOCKLOCRIN	Unlocks a local RIN.	standard
WHO	Returns information about a user.	standard
WRITELOG	Writes a record to a logging file.	user logging (LG) or system supervisor (OP)
XARITRAP	Enables or disables the user-written software arithmetic trap.	standard
XCONTRAP	Enables or disables the CONTROL-Y trap.	standard
XLIBTRAP	Enables or disables the software library trap.	standard
XSYSTRAP	Enables or disables the system trap.	standard
ZSIZE	Alters current Z to DB area.	standard



Section Divider

2. Intrinsic Descriptions

Intrinsic Descriptions

This section contains descriptions of all MPE intrinsics, arranged alphabetically. Each intrinsic description provides the intrinsic name, describes the call, defines parameters, explains condition codes, and where applicable gives information on the functional return, special considerations and areas of additional discussion.

Intrinsic Name

Following the intrinsic name, you will find a brief summary of its function.

Syntax

The syntax area contains the complete intrinsic call description, enclosed in a box. The intrinsic call descriptions are in the format shown below:

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

Required parameters, such as ***pin***, are shown in ***bold italics***. Optional parameters, such as *susp*, are shown in *standard italics*. The mnemonics which appear over the parameters indicate their type, and whether it must be passed by reference (the default), or by value. (Refer to Section I for a discussion of passing parameters by reference and by value.) For example IV, which appears above ***pin***, indicates that the parameter is an integer variable which must be passed by value. The mnemonics have the following meanings:

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

In addition to the mnemonics shown over the parameters, the mnemonic O-V is shown for some intrinsics to denote “option variable”. Option variable means that the intrinsic contains optional parameters and a complete call to this intrinsic need only specify the optional parameters desired by the programmer. Additionally, O-P is shown for those intrinsics (i.e. GETPRIVMODE) which can only be called when running in privileged mode. The ACTIVATE intrinsic, for example, contains two parameters: *pin*, which is a required integer that must be passed by value; and *susp*, an optional logical that, if included in the intrinsic call, must be passed by value. Additionally, the intrinsics option variable, which restates that some parameters (*susp* in this instance) are optional.

Note



If a byte array is passed to a parameter which requires a logical array, SPL will convert it to a logical array by shifting one bit to the right. Because of addressing modes inherent in the HP 3000 architecture, this shift can cause the wrong half bank of memory to be addressed, possibly resulting in the process aborting with a bounds violation. SPL will print a warning message when this condition occurs. This condition can be avoided by setting a byte array equivalent to a logical array for the message parameter.

Functional Return

Certain intrinsics return a value to the calling program (“type procedures”). A type procedure returns the value of a specified type (e.g. integer, real) in place of its name. (The result is actually returned to the top of stack and thus can be used in the rest of the expression). If the intrinsic is not a type procedure, this portion of the intrinsic description is omitted. The symbol `:=` is the SPL assignment identifier which means “is assigned the value of” or “receives the value of”. This convention is used throughout this chapter for consistency. It should not be interpreted that the value would be unavailable in languages other than SPL. The intrinsic call description format for type procedure intrinsics is illustrated below with the `READ` intrinsic:

I	LA	IV
<code>length:=READ(<i>message</i>,<i>expected</i>);</code>		



The `READ` intrinsic returns the positive length of the input actually read to an integer variable. The type, (e.g. integer, double), is signified by a mnemonic above the descriptive word. Thus, the `READ` intrinsic is in effect an integer procedure, *message* is a required logical array, and *expected* is a required integer parameter which must be passed by value.

A program using an intrinsic that returns a value may choose to omit the assignment (that is, the return variable is not required). The compiler will generate code to cause the return value to be deleted if a variable is not specified. Thus, the following examples are both legal calls of the `READ` intrinsic:

<code>LEN:=READ(MESS,20);</code>	<code>** User program may access LEN **</code>
<code>READ(MESS,20)</code>	<code>** Return variable unavailable **</code>

Parameters

All parameters are described. In the intrinsic descriptions in Chapter 2, required parameters are shown in ***bold italics***, and optional parameters are shown in *standard italics*. Throughout the rest of this manual, this distinction is not shown for required and optional parameters, and all parameters are shown in *standard italics*.

For some parameters certain bit settings have particular meanings; when significant these bit settings and their meanings will be noted. In other instances, (e.g. with logical parameters), many different values will have the same result (i.e. setting bit (15:1)=1 has the same interpretation as entering TRUE). For more information on the various options for a given parameter type refer to the *Systems Programming Language Reference Manual* (30000-90024).

Bit groups are denoted using the standard SPL notation. Thus bit (15:1) indicates bit 15; bits (0:3) indicates bits 0, 1, and 2. Bit 15 is on the right and is the least significant bit.

Condition Codes

Condition codes are included for each intrinsic. (Refer to “Intrinsic Call Errors” in Section I for a detailed description of the meaning and use of condition codes.)

Special Considerations

The special considerations portion of the description is omitted unless the intrinsic requires some special circumstances for proper execution. Therefore, unless explicitly stated the intrinsic does not:

- Operate in split-stack mode.
- Require special capabilities.
- Require a privileged call.

Required Capability

When you run a program file, the file system checks your group capabilities to see if you can access the file. Also, the capability of the program file (established at PREParation time) is checked against the capability of the group in which the file resides. If the capability of the file 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 and the user have sufficient capability to call them. If an intrinsic requires a special capability, it will be noted in the discussion of that intrinsic. Optional capabilities are discussed in Sections I and III.

Split-Stack Operations

During normal operation, DB (the Data Base register) points to the user's process stack. Some operations with extra data segments require that DB be set to the base of the extra data segment, while DL (the Data Limit register) and all other data registers (Z, the stack limit register; Q, the stack marker register; and S, the top of stack register) remain associated with the stack. When a process is operating in this mode it is said to have a split-stack. When accessing some parts of the operating system which use extra data segments, you are considered to be operating in split-stack mode, implicitly. It is also possible, if you are a privileged user, to force your process to operate in split-stack mode explicitly by calling the SWITCHDB intrinsic.

You should be aware, however, that it is possible for a privileged user to inadvertently destroy the operating system when operating in split-stack mode. Operating safely in split-stack mode requires extensive knowledge of the compiler: specifically, how the compiler assigns storage. For example, during normal operation, DB-relative variables point to the user's stack, but to different locations during split-stack mode operation. Thus, it is possible to unintentionally change data in areas which are normally reserved for MPE. Refer to the *System Programming Language Textbook* (30000-90025) for more information on the process stack.

When a process is operating in split-stack mode, whether implicitly or explicitly, you must recognize that some of the intrinsics you can normally call may not be called when DB does not point to the stack.

Such intrinsics, if called by a privileged process while the DB register is not set to the user's stack, can result in a system failure. If you are a normal user, (not operating in privileged mode), you need not concern yourself with this restriction, and you can assume that unless it is stated otherwise, an intrinsic will not operate in split-stack mode. However, if you have privileged mode capability, exercise extreme caution when calling intrinsics which operate in split-stack mode.

Caution



The normal checks and limitations that apply to users with standard (default) capabilities 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, upon request, investigate and attempt to resolve problems resulting from the user of privileged mode code. This service, which is not provided under the standard service contract, is available on a time and materials billing basis. Hewlett-Packard will not support, correct or attend to any modification of the MPE operating system software.

Related Information

Where applicable, the "Related Information" section will reference parts of this or other Hewlett-Packard manuals for additional information on the use of a particular intrinsic.

ABORTSESS

Enables a program to abort a specified job or session from the system. (Intrinsic number 196)

Syntax

IV DV IA
ABORTSESS(<i>jsid</i> , <i>jsnum</i> , <i>jsstatus</i>);

Parameters

jsid

integer by value (required)

Indicates whether the *jsnum* parameter refers to a job number or a session number:

Value	Meaning
1	Session number
2	Job number

Because jobs and sessions can have identical numbers, the *jsid* parameter is required to indicate whether *jsnum* describes a job or a session.

jsnum

double by value (required)

The job or session number of the job/session to be aborted.

jsstatus

integer array (required)

Returns the status of the intrinsic call. The array must be at least two elements in length. The second element always returns a zero. The first element returns a zero if no error occurs. If an error occurs, the first element returns one of the following error values:

Value	Meaning
0	SUCCESSFUL
1	JOBSECURITY set to HIGH, or job/session not yours
2	Job/session does not exist
3	Job/session being introduced and cannot be aborted when in the INTRO state
4	Job number must be a positive integer >0.
5	Execution of this command has been disabled.
6	Programmatic execution of this command has been disabled.
100	Undefined error.
7013	Invalid job type, a value of 1 or 2 was expected.

Operation Notes

All resources held by the aborted job/session are released, and any I/O activity is terminated. Although the job/session is abnormally terminated, MPE issues log records and updates CPU and connect times. The following message is displayed on the job/session list device:

SESSION ABORTED BY SYSTEM MANAGEMENT

ABORTSESS can be applied to waiting or scheduled jobs, or executing jobs or sessions. If the spooler input file (**\$STDIN**) for a batch job has been created and not yet opened, the entire file is deleted. If ABORTSESS is issued before the output spoolfile is complete, only that portion of the file already opened is printed and an error message is issued indicating that the job was aborted.

ABORTSESS can be successfully called only if the program is executing in a job/session that satisfies at least one of the following conditions:

- The job/session is on the system console.
- The job/session was allowed the ABORTJOB command.
- JOBSECURITY is set to LOW and the user and account names of the job/session are the same as the user and account names of the job/session being aborted.
- JOBSECURITY is set to LOW and the job/session has account manager (AM) capability, and the account name of the job/session is the same as the account name of the job/session being aborted.
- JOBSECURITY is set to LOW and the job/session has system manager (SM) capability.

ABORTSESS

Related Information	Intrinsics	STARTSESS
	Manual	<i>MPE V Commands Manual</i> (32033-90006)

ACTIVATE

Activates a newly created process, or a process suspended with the **SUSPEND** intrinsic. Requires process handling (PH) capability. (Intrinsic number 104)

Syntax

```

O-V      IV  LV
ACTIVATE(pin,allow);

```

Parameters

pin

integer by value (required)

Indicates the parent or child process to be activated. To indicate a parent process, set *pin* to zero. To indicate a child process, set *pin* to the process identification number (PIN) of the process. The called process must always be expecting an activation from the caller (refer to the **SUSPEND** and **CREATE** intrinsics).

allow

logical by value (optional)

Passes suspend or execute information to the process being activated:

- When *allow* is not specified or is zero, the called process is activated by the operating system but does not commence execution immediately. Instead, control returns to the calling process, which continues execution. This is the default.
- When *allow* is specified and is not zero, the calling process is suspended while the called process is activated and commences execution. Bits (14:2) specify the anticipated source of the call that reactivates the calling process:

Bits	Value/Meaning
15:1	Parent activation bit:

- | | |
|---|--|
| 0 | Does not expect activation by a parent |
| 1 | Expects activation by a parent |

Bits	Value/Meaning
14:1	Child activation bit:

- | | |
|---|---------------------------------------|
| 0 | Does not expect activation by a child |
| 1 | Expects activation by one child |

ACTIVATE

Bits	Value/Meaning
0:14	Reserved for MPE

Operation Notes

A newly created process can only be activated by its parent. A suspended process can be reactivated by its parent, or any of its children, as specified in the *allow* parameter of the **ACTIVATE** and **SUSPEND** intrinsics. When a process is activated, it runs until it is either suspended or deleted.

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

- The group in which the program files resides does not have process handling (PH) capability, or the program was not prepared with process handling capability.
- The required parameter *pin* is omitted
- A request to activate the father would result in activation of a job or session main process or a system process.

Condition Codes

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

Related Information

Additional Discussion	“Creating an Extra Data Segment” and “Creating and Activating Processes” in Chapter 3.
Intrinsics	CREATE , SUSPEND

ADJUSTUSLF

Adjusts directory space in a USL file by moving the start of the information block forward (or backward) on a user subprogram library (USL) file, thereby increasing (or decreasing) the space available for the file directory block. The overall length of the file does not change. This intrinsic is intended for programmers writing compilers. (Intrinsic number 83)

Syntax

I	IV	IV
<i>uslerror</i> := ADJUSTUSLF(<i>uslfnm</i> , <i>adjustment</i>);		

Functional Return

uslerror

integer

Returns the error number if an error occurs (condition code returns CCL). If no error occurs, no value is returned. The error number returned corresponds to the following:

Value	Meaning
0	The file specified by <i>uslfnm</i> was empty, an unexpected end-of-file was encountered while reading the <i>uslfnm</i> , or an unexpected end-of-file was encountered while writing the <i>uslfnm</i> .
1	Unexpected I/O error occurred.
4	Request exceeds the maximum file directory size (32,768 words).
5	Insufficient directory space.
6	Insufficient USL file information block space.

ADJUSTUSLF

Parameters	<i>uslfnum</i>	integer by value (required) The file number of the USL file (returned by FOPEN).
	<i>adjustment</i>	integer by value (required) The assigned record count: ■ If a positive value, the information block is moved toward the end-of-file in the USL file, increasing the directory block and decreasing the information block. ■ If a negative value, the information block is moved toward the start of the USL file, decreasing the directory block and increasing the information block.
Condition Codes	CCE	Request granted.
	CCG	Not returned.
	CCL	Request denied. An error number was returned to <i>uslferror</i> indicating the reason for this failure.
Related Information	Intrinsics	CLEANUSL , EXPANDUSLF , INITUSLF
	Manuals	<i>MPE Segmenter Reference Manual</i> (30000-90011)

ALTDSEG

Reduces the storage required by the extra data segment when moved into main memory and expands storage as required, allowing for a more efficient use of memory. Data segment management (DS) capability is required. (Intrinsic number 134)

Syntax

LV	IV	I
ALTDSEG(<i>index</i> , <i>increment</i> , <i>size</i>);		

Parameters

<i>index</i>	logical by value (required) Indicates the index of the extra data segment, obtained from the GETDSEG call.
<i>increment</i>	integer by value (required) Specifies the value, in words, the extra data segment is to be changed. A positive value requests an increase; a negative value requests a decrease.
<i>size</i>	integer by reference (required) Returns the new size of the extra data segment after incrementing or decrementing has occurred.

Operation Notes

Incrementation and decrementation is accomplished in even multiples of four words, which are rounded up. For example (in words):

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

When a data segment is created through GETDSEG, the required virtual space is allocated by the system to accommodate the original length of the data segment. This virtual space is allocated in increments of 512 words. For example, creation of an extra data segment with a length of 600 words results in the allocation of 1024 words of virtual space.

ALTDSEG

Note



ALTDSEG cannot increase the size of a data segment to exceed the virtual space originally allocated through GETDSEG.

When GETDSEG is called in privileged mode, the ALTDSEG intrinsic must also be called in privileged mode.

Condition Codes

CCE	Request granted.
CCG	Request granted. An illegal decrement was requested to a total segment size of ≤ 0 ; the current size remains in affect. An illegal increment was requested for a total segment size greater than the virtual space originally assigned; the size of the virtual space is granted and returned to the <i>size</i> parameter.
CCL	Request denied. An illegal <i>index</i> parameter was specified.

Related Information

Additional Discussion	“Changing the Size of an Extra Data Segment” in Chapter 3.
Intrinsics	FREEDSEG, GETDSEG

ARITRAP

Collectively enables all arithmetic traps or disables all arithmetic traps. (Intrinsic number 51)

Syntax

LV
`ARITRAP(trapstate);`

Parameters

trapstate

logical by value (required)

Enables/disables arithmetic traps.

Operation Notes

The possible interrupts listed below are collectively called the arithmetic traps:

- 3000 Mode double precision divide by zero
- 3000 Mode double precision overflow
- 3000 Mode double precision underflow
- 3000 Mode floating point divide by zero
- 3000 Mode floating point overflow
- 3000 Mode floating point underflow
- Decimal divide by zero
- Decimal overflow
- Integer divide by zero
- Integer overflow
- Invalid ASCII digit
- Invalid decimal digit
- Invalid decimal operand length
- Invalid source words count

ARITRAP

There is a difference between arming and enabling traps. Enabling a trap means that the occurrence of a trap condition is not ignored. Arming a trap is required so that, on a trap condition, a user-written routine is invoked and can take appropriate recovery actions. The following list summarizes what can happen when an arithmetic trap condition occurs:

- If a trap is both enabled and armed, the user-written trap handler is invoked.
- If a trap is enabled but not armed, an error message is output and the process aborts.
- If a trap is disabled, even though it is armed, the trap is ignored, and execution of the process continues without any interruption.

Condition Codes	CCE	Request granted. All arithmetic traps were originally disabled.
	CCG	Request granted. At least one arithmetic trap was originally enabled.
	CCL	Not returned.
Related Information	Additional Discussion	“Enabling and Disabling Traps” in Chapter 4.
	Intrinsics	XARITRAP

ASCII

Converts a binary number to a specified base and represents it as a numeric ASCII string. (Intrinsic number 63)

Syntax

I LV IV BA
numchar:=ASCII(*binvalue*,*base*,*asciieqv*);

Functional Return

numchar integer (assigned functional return)

Returns the number of characters in the resulting ASCII equivalent.

Parameters

binvalue logical by value (required)

The binary number to be converted to an ASCII string:

■ For octal conversions, *binvalue* must be an integer.
■ For decimal conversions, *binvalue* must be an integer.
■ For hexadecimal conversions, *binvalue* must be an integer

base integer by value (required)

One of the following conversion indicators (or the process aborts):

Value	Meaning
8	Convert to octal (pad with zeros)
10	Convert to decimal (left-justify)
-10	Convert to decimal (right-justify)
16	Convert to hexadecimal (pad with zeros)

asciieqv byte array (required)

Returns the converted value. Must be long enough to contain the result (<= 6 characters):

■ For octal conversions (*base*=8), 6 characters (including leading zeros) are returned; *numchar* returns the number of significant (right-justified) characters (excluding leading zeros). If *binvalue*=0, the length returned is 1.

- For decimal conversions, *binvalue* is considered a twos complement integer ranging from -32768 to +32767. If *binvalue*=0, only one zero character is returned in *asciieqv*; *numchar* returns the total number of characters (including the sign). For example, if *binvalue*=0, the length returned is 1, and if *binvalue*=327, the length returned is 3.
- For decimal left-justified conversions (*base*=10), leading zeros are removed and the numeric ASCII result is left-justified in *asciieqv*; the most significant digit (or the "-" sign) is in *asciieqv*(0), the next most significant digit is in *asciieqv*(1), and so on.
- For decimal right-justified conversions (*base*=-10), the result is right-justified in *asciieqv*; the least significant digit is in *asciieqv*(0), the next least significant digit is in *asciieqv*(-1), and so on.
- For right-justified conversions, the character array where the converted value is to be placed must specify the rightmost byte where data is placed. For example, if *asciieqv* is a 10-byte array declared as:

```
BYTE ARRAY STRING (0:9);
```

then you must specify it in the ASCII intrinsic call as follows (for right justification):

```
NUMCHAR:=ASCII(WORD,-10,STRING(9));
```

The result is right-justified in *asciieqv*, with the rightmost digit of the result contained in the last (rightmost) byte of *asciieqv*.

- For hexadecimal conversions (*base*=16), 4 characters (including leading zeros) are returned. The digits can be 0..9 and A..F. *Numchar* returns the number of significant (right-justified) characters (excluding leading zeros). For example, if *binvalue*=32, *numchar* is 2 and *asciieqv* will be 0020.

Related Information	Additional Discussion	“Converting Numbers from Binary Code to ASCII Strings” in Chapter 5.
	Intrinsics	DASCII, BINARY, DBINARY

BEGINLOG

Posts a special record to the user logging file to mark the beginning of a logical transaction. When **BEGINLOG** is called, the logging memory buffer is flushed to ensure that the record gets to the logging file. User logging (LG) or system supervisor (OP) capability is required. (No intrinsic number assigned)

Syntax

D	LA	I	I	I
BEGINLOG(<i>index,data,length,mode,logstatus</i>)				

Parameters	<i>index</i>	double by reference (required)
		Passes access information to the user logging system. This is the value returned in the <i>index</i> parameter of the OPENLOG intrinsic.
	<i>data</i>	logical array (required)
		Passes the information to be logged. A log record contains 128 words where 119 are available for writing data. For the most efficient use of log file space, structure arrays with lengths in multiples of 119 words.
	<i>length</i>	integer by reference (required)
		Passes the length of the data. A positive value indicates word; a negative value indicates bytes. If the length is greater than 119 words (or 238 bytes), the information in <i>data</i> is divided into two or more physical log records.
	<i>mode</i>	integer by reference (required)
		Passes a value indicating whether the logging process should suspend the process if it cannot complete the request for service immediately. If it is not possible to log the transaction and <i>mode</i> is set to NOWAIT, the BEGINLOG intrinsic indicates, through <i>logstatus</i> , that it could not complete the request:

Value	Meaning
0	WAIT
1	NOWAIT

BEGINLOG

logstatus

integer by reference(required)

Returns a value indicating the success/failure of the intrinsic call:

Value	Meaning
0	Successful
1	NOWAIT mode requested, and user logging process is busy
2	Parameter out of bounds in logging intrinsic
3	Request to open or write to a logging process that is not running
4	Incorrect index parameter passed to a logging intrinsic
5	Incorrect mode parameter passed to a logging intrinsic
6	User request denied because logging process is suspended
7	Illegal capability. Must have user logging (LG) and system supervisor (OP) capabilities to use a logging intrinsic
8	Incorrect password passed to a logging intrinsic
9	Error occurred while writing to the logging file
10	Invalid DST passed to logging system intrinsic
12	System out of disk space, user logging cannot proceed
13	No more logging entries
14	Invalid access to logging file
15	End-of-file on user logging file
16	Invalid logging identifier

Operation Notes

Use the **data** parameter of this intrinsic to post user data to the logging file. This use of **BEGINLOG** is similar to using the **WRITELOG** intrinsic, but also denotes the beginning of a logical transaction.

Related Information

Additional Discussion

“User Logging” in Chapter 3.

Intrinsics

ENDLOG, WRITELOG

BINARY

Converts a numeric (octal or decimal) ASCII string to a twos complement binary value. (Intrinsic number 62)

Syntax

I	BA	IV
<i>bineqv</i> :=BINARY(<i>asciieqv</i> , <i>length</i>);		



Functional Return

bineqv integer

The binary equivalent of the numeric string.

Parameters

asciieqv byte array (required)

Passes the octal or signed-decimal number (in ASCII characters) to be converted. If the character string begins with a percent sign (%), it is treated as an octal value. If the character string begins with a dollar sign (\$), it is treated as a hexadecimal value. If the character string begins with a plus sign, a minus sign, or a number, it is treated as a decimal value. Leading blanks are not allowed and are treated as illegal characters.

length integer by value (required)

The length (number of bytes) of the byte array containing the ASCII-coded value. If the value of *length* is 0, the intrinsic returns 0 to the calling process. When the value of *length* is negative, the intrinsic causes the process to abort.

Condition Codes

CCE	Request granted. A one-word binary value is returned to the process.
CCG	Request denied. A word overflow, possibly resulting from too many characters (<i>asciieqv</i> number too large), occurred in the value returned.
CCL	Request denied. An illegal character was encountered in <i>asciieqv</i> . For example, the digits 8 or 9 were specified in an octal value.

Related Information

Additional Discussion	“Converting Numbers from an ASCII Numeric String to a Binary Coded Value” in Chapter 5.
Intrinsics	DBINARY

CALENDAR

Returns the calendar date, including the day of year and the year of century. (Intrinsic number 43)

Syntax

L <i>date</i>:=CALENDAR;

Functional Return

date logical (assigned functional return)

Returns the calendar date in the following format:

Bits	Value/Meaning
7:9	Day of year
0:7	Year of century

Related Information

Additional
Discussion

“Obtaining the Calendar Date” in Chapter 5.

Intrinsics

ALMANAC

CAUSEBREAK

Interrupts the program (the entire process structure). The CAUSEBREAK intrinsic is the programmatic equivalent to pressing **Break** in a session, and is not applicable in jobs. The program is suspended while in BREAK mode. Execution of the program can resume where the interruption occurred by entering the RESUME command, or be aborted by entering the ABORT command. (Intrinsic number 56)

Syntax

```
CAUSEBREAK;
```

Condition Codes	CCE	Request granted.
	CCG	Not returned.
	CCL	Request denied. The intrinsic was not called from an interactive session.

Related Information	Additional Discussion	“Requesting a Process Break” in Section 5.
	Commands	ABORT, RESUME

CLEANUSL

Deletes all inactive entries from currently managed USL files and returns the file number of the new USL file. Therefore, you must test the condition code immediately upon return from the intrinsic. Unpredictable results occur if an error number is used as a file number. (No intrinsic number assigned)

Syntax

I	IV	BA
<i>filenum</i> := CLEANUSL(<i>uslfnum</i> , <i>formaldesig</i>);		

Functional Return

filenum

integer (assigned functional return)

Returns the new file number, if no error occurs, or if an error occurs condition code returns CCL and *filenum* returns one of the following:

Value	Meaning
0	Unexpected end-of-file (EOF) marker on either the old or the new USL file
1	Unexpected I/O error on either the old or the new USL file
7	Unable to open new USL file
12	Invalid USL file

Parameters

uslfnum

integer by value (required)

The file number of the file.

formaldesig

byte array (required)

Passes the name to be given to the cleaned file. The array must end with a blank. If it contains all blanks, it deletes the inactive entries.

Condition Codes

CCE

Request granted. The new file number is returned to *filenum*.

CCG

Not returned.

CCL

Request denied. An error number is returned to *filenum*.

Related Information

Intrinsics

ADJUSTUSLF, EXPANDUSLF, INITUSLF

Manuals

MPE Segmenter Reference Manual (30000-90011)

CLOCK

Returns the time (hours, minutes, seconds, and tenths of seconds) according to the system timer. (Intrinsic number 44)

Syntax

D
time:=CLOCK;

Functional Return

time **double by reference (assigned functional return)**
Returns the time, as monitored by the system timer, in the following format:

	Bits	Value/Meaning
Word 1:	8:8	The minute of the hour
	0:8	The hour of the day
Word 2:	8:8	The tenths of seconds
	0:8	The seconds

Related Information

Additional Discussion “Time and Date Intrinsics” in Chapter 5.

CLOSELOG

Closes access to the user logging facility. User logging (LG) or system supervisor (OP) capabilities are required. (Intrinsic number 212)

Syntax

D I I
CLOSELOG(<i>index</i> , <i>mode</i> , <i>logstatus</i>);

Parameters

index

double by reference (required)

Passes access capabilities to the user logging system. This is the value returned in the *index* parameter of the OPENLOG intrinsic.

mode

integer by reference (required)

Passes a value, indicating whether the user logging facility should suspend the process if it cannot complete the request for service immediately:

Value	Meaning
0	WAIT
1	NOWAIT

logstatus

integer by reference (required)

Returns one of the following values, indicating the success/failure of the intrinsic call:

Value	Meaning
0	Successful
1	NOWAIT mode requested; user logging process busy
2	Parameter out of bounds in logging intrinsic
3	Request to open or write to a logging process that is not running
4	Incorrect <i>index</i> parameter passed
5	Incorrect <i>mode</i> parameter passed
6	User request denied because logging process is suspended
7	Illegal capability; user logging (LG) or system supervisor (OP) capabilities required
9	Error occurred while writing
10	Invalid DST passed to logging system intrinsic
12	System out of disk space; user logging cannot proceed
13	No more logging entries
14	Invalid access to user logging file
15	End-of-file encountered
16	Invalid logging

CLOSELOG

Operation Notes

The number of users and log entries are independent of the number of times the `OPENLOG`/`CLOSELOG` intrinsics are called within an application. The user logging facility establishes a logging buffer (`LOGBUFF`) entry and increments the user count only at the first `OPENLOG` call. Another counter keeps track of the number of times `OPENLOG` and `CLOSELOG` are called. The counter is incremented for every `OPENLOG` and decremented for every `CLOSELOG`. This ensures that the entry in `LOGBUFF` is released only for the last `CLOSELOG` call (call counter = 0).

Related Information

Additional Discussion	“User Logging” in Chapter 3.
Intrinsics	<code>OPENLOG</code>

COMMAND

Executes an MPE command programmatically. (Intrinsic number 68)

Syntax

BA	I	I
COMMAND(<i>cmdimage</i> , <i>cmderror</i> , <i>parmnum</i>);		

Parameters*cmdimage***byte array (required)**

Passes an ASCII string of <=278 characters consisting of a command and parameters, terminated by a carriage return. The carriage return character (%15) must be the last character of the command string. Do not include a prompt character in this string.

*cmderror***integer by reference (required)**

Returns any error code set by the command. This is the same error code that appears on a job/session list device if the command is part of an input stream (a command interpreter rather than a file system error code). If no error occurs, *cmderror* returns zero (0). If a warning is detected, *cmderror* returns a negative number (for example, "-383" is returned if CIWARN 383 is detected).

*parmnum***integer by reference (required)**

Returns a number indicating the nature of the error if *cmderror*0:

- If *parmnum* is positive, this is a file system error number.
- If *parmnum* is negative, it is the number of the column where the error occurred (the CI would print a carat there).

COMMAND

Operation Notes

The system manager can disable specific commands from programmatically being executed. If disabled, the **COMMAND** intrinsic will return the following message on the screen:

```
PROGRAMMATIC EXECUTION OF THIS COMMAND HAS BEEN
DISABLED (CIERROR 11)
```

to the calling program, indicating that system access has been disabled.

The **COMMAND** intrinsic will ignore any non-executable command and return the following message on the screen:

```
COMMAND NOT PROGRAMMATICALLY EXECUTABLE (CIERR 12)
```

User-defined commands (UDCs) cannot be executed with the **COMMAND** intrinsic. **RUN** and other process creation commands are allowed if you have process handling (PH) capability. Command files and the implied **RUN** are not allowed.

The *cmdimage* parameter can contain any MPE command except the following:

- ABORT
- BYE
- DATA
- EOD
- EOJ
- EXIT
- HELLO
- JOB
- SETCATALOG
- REDO
- RESUME

Note



CIERROR 9103 is returned if *cmdimage* contains a command that is not supported.

Condition Codes

CCE	Request granted. Although, a CI warning may have been detected.
CCG	Request denied. An executor-dependent error, such as an erroneous parameter, prevented execution of the command. The numeric error code is contained in <i>cmderror</i> . The command with the error terminated before executing completely.
CCL	Request denied. The command was undefined.

Related Information

Additional Discussion	“Executing Commands Programmatically” in Chapter 5.
-----------------------	---

CREATE

Creates a process as a child of the calling process. Process handling (PH) capability is required. (Intrinsic number 100)

Syntax

```

O-V      BA      BA      I  IV  LV
CREATE(formaldesig,entryname,pin,parm,loadflag,
      IV      IV      IV      LV      IV
      stacksize,dlsiz,maxdata,priorityclass,rank);

```

Parameters	<i>formaldesig</i>	byte array (required) Passes the name, account (optional), and group (optional) of the file containing the program to be run. A lockword must be specified (if applicable). The last element of the array must be a blank. (Equivalent to the <i>progfile</i> parameter of the RUN command.)
	<i>entryname</i>	byte array (optional) Passes the declared entry point (label) in the program where execution is to begin when the process is activated. The last element of the array must be a blank. Specify the program's primary entry point with an array consisting of one blank character. (Equivalent to the <i>entrypoint</i> parameter of the RUN command.) Default: The program's primary entry point is used.
	<i>pin</i>	integer by reference (required) Returns the process identification number (PIN) of the newly created process. This PIN is used by other intrinsics to reference the new process. If an error is detected, a value of zero is returned.
	<i>parm</i>	integer by value (optional) Passes information to the new process. The new process can access this information by using the GETINFO intrinsic. (Equivalent to the <i>parm=</i> parameter of the RUN command.) Default: 0

CREATE

loadflag **integer by value (optional)**

Specifies the loading options as follows:

	Bits	Value/Meaning
	15:1	Active:
0		Calling process not activated when the new process terminates
1		MPE reactivates the calling process (parent) when the new process terminates

Default: 0

	Bits	Value/Meaning
	14:1	Loadmap (equivalent to the LMAP parameter of the RUN command):
0		No map produced
1		A list 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.

Default: 0

	Bits	Value/Meaning
	13:1	Debug; ignored if nonprivileged and the new process requires privileged mode or if there is no READ/WRITE access for the new process program file (equivalent to the DEBUG parameter of the RUN command.):
0		Breakpoint not set
1		Debug called prior to executing the first instruction of the new process

Default: 0


Bits Value/Meaning

12:1 Privilege (equivalent to the **NOPRIV** parameter of the **RUN** command):

0 Program loaded in the mode specified when the program was prepared

1 Program loaded in nonprivileged mode

Default: 0

Bits Value/Meaning

10:2 Library Search; specifies which libraries to search for the program (equivalent to the **LIB=** parameter of the **RUN** command):

00 search only the system libraries.

01 Search the account public library, then search the system libraries.

10 Search the group library, then the account public library, and then the system libraries.

Default: 00

Bits Value/Meaning

9:1 **NOCB** bit; controls where the control blocks of device files are established (equivalent to the **NOCB** parameter of the **RUN** command):

0 Establish file control blocks in the process control block extension (PCBX) area

1 Establish file control blocks in an extra data segment

Default: 0

stacksize **integer by value (optional)**

The number of words assigned to the local stack area bounded by the initial Q; and Z registers. A value of -1 will be assigned by the MPE Segmenter by default (equivalent to omitting this parameter.)

dlsiz **integer by value (optional)**

The number of words in the user stack area bounded by the DL and DB registers. A value of -1 indicates that the MPE Segmenter will assign the default values (equivalent to omitting this parameter.)

CREATE

<i>maxdata</i>	integer by value (optional) The maximum size allowed for the process stack in words. This is specified at program preparation. A value of -1 indicates that the MPE Segmenter will assign the default value. If not specified in either the intrinsic call or the program file, the stack will remain the same.
<i>priorityclass</i>	logical by value (optional) The integer equivalent of two ASCII characters specifying the priority class where the new process is scheduled (AS, BS, CS, DS, ES). It is computed as: $(\text{ASCII of first character} * 256) + (\text{ASCII of second character})$ The integer equivalents are: AS = 16,723 BS = 16,979 CS = 17,235 DS = 17,491 ES = 17,747 If a user is nonprivileged, the processes can be rescheduled into any of the five subqueues except the AS queue. This function is limited by the maximum priority assigned to the account by the system manager. If a user is privileged, the processes can be scheduled into any subqueue, including the AS. A process in the AS or BS linear queue does not give up CPU voluntarily; it could loop infinitely and prevent other processes from accessing the CPU. (Equivalent to the PRI= parameter of the RUN command.) Default: The calling process priority.
<i>rank</i>	integer by value (optional) For backward-compatibility with pre-MPE IV based operating systems only.

Operation Notes

The CREATE intrinsic does the following:

1. Loads the program to be run by the new process into virtual memory.
2. Creates the process as the child of the calling process.
3. Initializes the process data stack.
4. Schedules the process.
5. Returns the new process identification number (PIN) to the requesting process.

The process is not created and the *pin* parameter is returned with a value of zero if:

- The value (a nonexistent subqueue) specified is illegal for the *priorityclass* parameter.
- The *formaldesig* is illegal.
- The *entryname* is illegal.

The process is not created and the *pin* parameter is returned unmodified if:

- The program file of the creating process does not have process handling (PH) capability.
- A required parameter (*formaldesig* or *pin*) is omitted.
- A reference parameter is not within the required range.

The request is granted and the process is created with *maxdata* allowed if:

- The *maxdata* exceeds the configured *maxdata*, where *maxdata* is either the value passed as a parameter, or a value recomputed by the Loader under Condition 1 of CCG.

The request is granted and the process is created if:

- The stack space exceeds the maximum *stacksize* defined during the system configuration. The *dlsiz*e may have been modified to satisfy Condition 2 under CCG.

The request is denied and PIN of zero is returned if:

- The *progrname* is illegal.
- The *entryname* is illegal.

CREATE

Condition Codes	CCE	Request granted. The new process has been created.
	CCG	Request granted. The <i>maxdata</i> and /or <i>dlsiz</i> e parameters given were illegal, but other values were assigned as follows: 1. If the <i>maxdata</i> specified exceeds the maximum Z-DL allowed by the configuration, the configured maximum value is assigned. 2. If the areas from Base to DB is not zero, then the DL area pads this area to round it up to 128.
	CCL	Request denied. The specified <i>formaldesig</i> ; or <i>entryname</i> does not exist.
Related Information	Additional Discussion	“Creating an Extra Data Segment” and “Creating and Activating Processes” in Chapter 3.
	Intrinsics	ACTIVATE, SUSPEND
	Commands	RUN

CREATEPROCESS

Creates a process and allows you to assign \$STDIN and \$STDLIST to any file. Process handling (PH) capability is required. (Intrinsic number 101)

Syntax

O-V	I	I	BA
CREATEPROCESS(<i>createstatus</i> , <i>pin</i> , <i>formaldesig</i> ,			
	IA	IA	
<i>itemnum</i> , <i>item</i> ,);			

Parameters

createstatus integer by reference (required)

Returns a value indicating the type of success or failure. A negative value indicates that the associated message is a warning. An error terminates the intrinsic call while a warning means the call continues. Check the Job Control Word (JCW) before continuing. The following is a list of the possible values returned:

CREATEPROCESS

Value	Meaning
0	Successful
1	Process handling (PH) capability required
2	<i>Pin</i> or <i>formaldesig</i> parameter missing
3	<i>Pin</i> or <i>formaldesig</i> parameter address out of bounds
4	Out of system resources (e.g., PCB's, DST's)
5	Process not created; invalid <i>itemnum</i> specified
6	Process not created; <i>formaldesig</i> does not exist
7	Process not created; <i>formaldesig</i> invalid
8	Process not created; entry name invalid or does not exist
-9	Process created; LIBSEARCH bits ignored
-10	Process created; <i>itemnum</i> =19 ignored
-11	Process created with default <i>mazdata</i> from the program file (specified <i>mazdata</i> was < 0)
-12	Process created with <i>dlsiz</i> e rounded up to next 128-word multiple
-13	Process created with <i>mazdata</i> decreased to configuration maximum.
-14	Process created with <i>mazdata</i> increased to <i>dlsiz</i> e + <i>globsize</i> + <i>stacksize</i> (<i>globsize</i> is defined to be primary DB <i>space</i> + <i>secondary</i> DB space)
15	Process not created; reserved <i>item</i> was specified
16	Process not created; hard load error occurred (for example, I/O error reading <i>formaldesig</i>)

Value	Meaning
17	Process not created; illegal value specified for <i>itemnum</i> =7
18	Process not created; specified \$STDIN could not be opened
19	Process not created; specified \$STDLIST could not be opened
20	Process not created; string to be passed to new process invalid (pointer without length, length without pointer, or length exceeds stack size of calling process)

<i>pin</i>	integer by reference (required) Returns the process identification number (PIN) of the newly created process. This PIN is used by other intrinsics to reference the new process. If an error is detected, a value of zero is returned.
<i>formaldesig</i>	byte array (required) Passes the name of the program file to be run. Terminate the string with any nonalphanumeric character other than a period (.) or a backslash (/). (Equivalent to the <i>progfile</i> parameter of the RUN command.)
<i>itemnum</i>	integer array (optional) An array of integers whose values specify which value is to be returned in the corresponding element of <i>item</i>. (Refer to Table 2-1.)
<i>item</i>	logical array (optional) Returns the value of the item specified in the corresponding <i>itemnum</i> array element. (Refer to Table 2-1.)

CREATEPROCESS

Table 2-1. CREATEPROCESS Itemnum/Item Values

Itemnum	Mnemonic	Item Description
0	I	Indicates the end of the option list. No value is required in the corresponding <i>item</i> .
1	I	The address containing the entry point name in the program where the new process is to begin execution (equivalent to the <i>entrypoint</i> parameter of the RUN command.) The name consists of a string of characters terminated by a blank.
2	I	A value passed to the new process (equivalent to the PARM= parameter of the RUN command.)
3	I	The load options for loading the program file of the new process, designated as: Bits (0:16) Load options Item 3 has the same definition as the <i>loadflag</i> parameter of the CREATE intrinsic. If both <i>itemnums</i> 3 and 19 are specified, the LIBSEARCH option is ignored or if LIBSEARCH is set to a value other than its default (00), a warning is returned indicating the LIBSEARCH option was ignored.
4,5,6	I	Equivalent to the STACK= , DL= , and MAXDATA= parameters of the RUN command.
7	I	The priority class where the new process is scheduled (AS, BS, CS, DS, ES); is computed as (equivalent to the PRI= parameter of the RUN command.): (ASCII of first character * 256) + (ASCII of second character) The integer equivalents are: AS = 16,723 BS = 16,979 CS = 17,235 DS = 17,491 ES = 17,747 If in user mode (nonprivileged), any priority class can be specified except AS; this is limited by the maximum priority assigned to the account by the system manager. If PM capability has been assigned, processes can be scheduled into all subqueues. A process in the linear queue AS or BS does not give up the CPU voluntarily, it could loop infinitely and prevent other processes from accessing the CPU. Default: The calling process priority class.

Table 2-1. CREATEPROCESS Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
8	I	The address (type-coerced) containing a definition of the file to be used as \$STDIN for the new process (equivalent to the STDIN= parameter of the RUN command). Must contain an ASCII string (terminated by a carriage return) specifying the file to be used as \$STDIN (that is, everything after the FILE formaldefsig=; portion of the file equation). If <i>itemnum</i> =8 is not specified, the default \$STDIN and \$STDLIST (current for the creating (parent) process) are used for the new process creation.
9	I	The address (type-coerced) containing a definition of the file to be used as \$STDLIST for the new process (equivalent to the STDLIST= parameter of the RUN command). (The corresponding <i>item</i> in the item array is the address, type-coerced to an integer, of a character array containing the definition of the file to be used as \$STDLIST for the new process. If <i>itemnum</i> =9 is not specified, the default \$STDIN and \$STDLIST (current for the creating (parent) process) are used for the new process creation.
10	I	Specifying this item is equivalent to calling the ACTIVATE intrinsic with the newly created process. The value passed is equivalent to the <i>allow</i> parameter of ACTIVATE , except bit (31:1) is the parent activation bit and bit (30:1) is the child activation bit.
11	I	The address (type-coerced) containing information to be passed to the new process. The length of the string in <i>itemnum</i> =12 must be specified (equivalent to the INFO= parameter of the RUN command), accessible through the GETINFO intrinsic.
12	I	The length (in bytes) of the string specified by <i>itemnum</i> =11. Accessible through the GETINFO intrinsic.

Condition Codes

CCE	Request granted.
CCG	Request granted. Check the createstatus parameter to see if warning codes were returned.
CCL	Request denied. Check the createstatus parameter to determine why the request was denied.

Related Information

Additional Discussion	“Creating and Activating Processes” in Chapter 3.
Intrinsics	CREATE
Commands	RUN

CTranslate

Converts a string of characters between EBCDIC and ASCII, or between EBCDIK (HP-specific version of EBCDIC) and KANA8 (8-bit, Japanese International Standard (JIS) version of USASCII code). (Intrinsic number 61)

Syntax

0-V

IV

BA

BA

CTranslate(*transcode*,*inbuffer*,*outbuffer*,

IV

BA

bufferlength, *transtable*)

Parameters

<i>transcode</i>	integer by value (required) Passes one of the following translation identifiers: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0</td><td>User-supplied table specified in the <i>transtable</i> parameter</td></tr><tr><td>1</td><td>Convert EBCDIC to ASCII</td></tr><tr><td>2</td><td>Convert ASCII to EBCDIC</td></tr><tr><td>3</td><td>Reserved for MPE</td></tr><tr><td>4</td><td>Reserved for MPE</td></tr><tr><td>5</td><td>Convert EBCDIK to KANA8 (JIS)</td></tr><tr><td>6</td><td>Convert KANA8 (JIS) to EBCDIK</td></tr></table>	Value	Meaning	0	User-supplied table specified in the <i>transtable</i> parameter	1	Convert EBCDIC to ASCII	2	Convert ASCII to EBCDIC	3	Reserved for MPE	4	Reserved for MPE	5	Convert EBCDIK to KANA8 (JIS)	6	Convert KANA8 (JIS) to EBCDIK
Value	Meaning																
0	User-supplied table specified in the <i>transtable</i> parameter																
1	Convert EBCDIC to ASCII																
2	Convert ASCII to EBCDIC																
3	Reserved for MPE																
4	Reserved for MPE																
5	Convert EBCDIK to KANA8 (JIS)																
6	Convert KANA8 (JIS) to EBCDIK																
<i>inbuffer</i>	byte array (required) Passes the string of characters to be translated.																
<i>outbuffer</i>	byte array (optional) Passes the translated character string. If an <i>outbuffer</i> parameter is not specified, all translation occurs within <i>inbuffer</i> . The <i>inbuffer</i> and <i>outbuffer</i> parameters can specify the same array.																
<i>bufferlength</i>	integer by value (required) The number of characters to be converted.																

	<i>transtable</i>	byte array (required when <i>transcode</i>=0, otherwise optional) Passes the user-defined translation table. The contents and order of <i>transtable</i> define the translation process. <i>Transtable</i> must be <= 256 bytes. <i>Transtable</i> is constructed so that each byte corresponds to a byte value in the source string to be translated. For example, the first byte in <i>transtable</i> gives the code to be substituted for source bytes whose value is 0.
Condition Codes	CCE	Request granted. Translation performed successfully.
	CCG	Not returned.
	CCL	Request denied. An error occurred.
Related Information	Additional Discussion	“Translating Characters with the CTranslate Intrinsic” in Chapter 5.

DASCII

Converts a 32-bit binary number to a specified base and represents it as a numeric ASCII string. (Intrinsic number 75)

Syntax

I	DV	IV	BA
<i>numchar</i> :=DASCII(<i>binvalue</i> , <i>base</i> , <i>asciieqv</i>);			

Functional Return

numchar

integer (assigned functional return)

Returns the number of characters in the resulting string.

Parameters

binvalue

double by value (required)

The binary number to be converted to an ASCII string.

base

integer by value (required)

One of the following translation identifiers:

Value	Meaning
8	Convert to Octal (pad with zeros)
10	Convert to Decimal (left-justified)
-10	Convert to Decimal (right-justified)
16	Convert to hexadecimal (pad with zeros)

If any other number is entered in this parameter, the intrinsic causes the process to abort.

asciieqv

byte array (required)

Returns the converted value. Must be long enough to contain the result (≤ 11 characters).

- For octal conversions (*base*=8), 11 characters (including leading zeros) are returned; *numchar* returns the number of significant (right-justified) characters (excluding leading zeros). If *binvalue*=0, the length returned is 1.
- For decimal conversions (*base*=10), *binvalue* is considered a twos complement integer ranging from -2,147,483,648 to +2,147,483,647. If *binvalue*=0, only one zero character is returned in *asciieqv*; *numchar* returns the total number of characters (including the sign). For example, if *binvalue*=0, the length returned is 1.

- For hexadecimal conversions (*base*=16), 8 characters (including leading zeros) are returned. The digits can be 0..9 and A..F. *Numchar* returns the number of significant (right-justified) characters (excluding leading zeros). For example, if *binvalue*=32, *numchar* is 2 and *asciieqv* will be 0020.

Related Information

Additional
Discussion

Intrinsics

“Converting Numbers from Binary Code to ASCII Strings” in Chapter 5.

ASCII

DATELINE

Returns the current date and time, including the day of week, month, day, year, hours, and minutes. (No intrinsic number assigned)

Syntax

BA DATELINE(<i>datebuffer</i>);

Parameters

datebuffer

byte array (required)

Returns the date and time information, in a 27-character array, in the format:

FRI, MAY 19, 1989, 12:06 PM

Related Information

Intrinsics

ALMANAC

DBINARY

Converts a numeric ASCII string to a binary value. The numeric ASCII string can be octal, hexadecimal, or decimal. (Intrinsic number 74)

Syntax

DBAIVdbineqv:=DBINARY(dasciieqv,length);

Functional Return	dbineqv	double (optional) Returns the converted binary value.
-------------------	---------	--

Parameters	dasciieqv	byte array (required) Passes the octal or signed decimal number (in ASCII characters) to be converted. If the character string begins with a percent sign (%), it is treated as an octal value. If the character string begins with a dollar sign (\$), it is treated as a hexadecimal value. If the string begins with a plus sign, minus sign, or a number, it is treated as a decimal value. Leading blanks are not allowed and are treated as illegal characters.
	length	integer by value (required) The length (number of bytes) of the ASCII-coded value: ■ If the value of <i>length</i> is 0, the intrinsic returns 0 to the calling process. ■ If the value of <i>length</i> is negative, the intrinsic causes the process to abort.

Condition Codes	CCE	Request granted. A binary value is returned to the process.
	CCG	Request denied. A word overflow, possibly resulting from too many characters (<i>dasciieqv</i> number too large), occurred in the value (<i>dbineqv</i>) returned.
	CCL	Request denied. An illegal character was encountered in <i>dasciieqv</i> . For example, the digits 8 or 9 were specified in an octal value.

Related Information	Additional Discussion	“Converting Numbers from an ASCII Numeric string to a Binary Coded Value” in Chapter 5.
	Intrinsics	BINARY

DEBUG

Invokes the Debug facility from an interactive program and allows object code to be analyzed. (Intrinsic number 99)

Note



Consult the *MPE Debug/Stack Dump Reference Manual* (30000-90012); before attempting to use the Debug facility.

Syntax

DEBUG;

Operation Notes

The Debug facility has its own set of commands and a symbolic window display feature. The intrinsic call acts as a hard-coded breakpoint. Execution of the calling program is halted and the debug facility prompt is displayed. If the call is made from a batch program, it is ignored.

Related Information

Manuals *MPE Debug/Stack Dump Reference Manual*
(30000-90012).

DLSIZE

Causes the area between DL and DB to be expanded or contracted within the stack. (Intrinsic number 135)

Syntax

I	IV
<i>dldb</i> size:=DLSIZE(<i>size</i>);	

Functional Return*dldb*size

integer (optional)

Returns the granted number of words for the DL-to-DB area. This value is negative, unless error condition CCL is returned; a positive value is possible.

Parameters*size*

integer by value (required)

The desired number of words for the DL-to-DB area, represented by a negative value. A *size* of 0 is permitted and resets the DL-to-DB area to the original value assigned when the process was created (initial DL). (This is the only way to contract the DL-to-DB area.) The size actually granted is the absolute value of *size*, rounded up so that the distance between the beginning of the segment and DB is a multiple of 128 words.

Operation Notes

If expanding, all information in the area between DL and DB is saved. If contracting, all information in the area between DL and DB is lost.

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

Condition Codes

CCE

Request granted. The value returned is $\geq$ the value requested.

CCG

Request denied. The requested size exceeded the maximum limit allowed. The maximum limit allowable is granted, and its size is returned.

CCL

Request denied. An illegal *size* parameter was specified, the size requested is less than the initial DL size, or the *size* parameter was a positive integer.

DLSIZE

The original area size, assigned when the stack segment was created, is granted.

Related Information	Additional Discussion	“Changing the DL to DB Area Size” in Section 5.
	Intrinsics	None
	Commands	None
	Manuals	None

DMOVIN

Copies data from an extra data segment into a data area. Data segment management (DS) capability is required. (Intrinsic number 132)

Syntax

IV	IV	IV	LA
DMOVIN(<i>index</i> , <i>displacement</i> , <i>number</i> , <i>location</i>);			

Parameters

index

logical by value (required)

The index of the extra data segment, obtained from a GETDSEG intrinsic call.

displacement

integer by value (required)

The displacement, in words, of the first word in the string to be transferred from the first word in the data segment. This value must be ≥ 0 .

number

integer by value (required)

The size, in words, of the data string to be transferred. This value must be ≥ 0 .

location

logical array (required)

Contains the array (buffer) in the stack where the data string is to be moved.

DMOVIN

Operation Notes

When DMOVIN is called, a bounds check is initiated to ensure that the data is taken from within the data segment boundaries and moved to an available data area. For example, in Figure 2-1, to move four words from locations 422 through 425 of the extra data segment (whose *index* is 21) to array ARA (located in your data area), the intrinsic call would be:

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

The *index* is 21 (refer to GETDSEG); *displacement* within the data segment is 422; the *number* of words to move into the data area is 4; and the *location* to begin transferring the data is the address of ARA[10]. If ARA is an array of words, the four words are moved to ARA[10] through ARA[13], as shown in Figure 2-1.

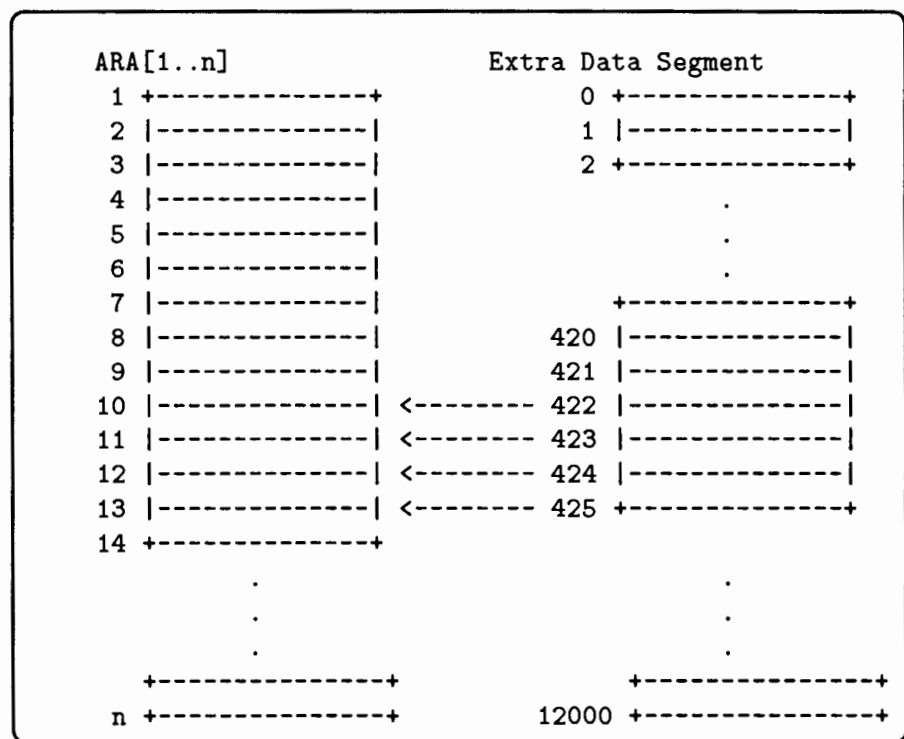


Figure 2-1. DMOVIN Data Movement.

Condition Codes

CCE	Request granted.
CCG	Request denied. A bounds-check failure occurred.
CCL	Request denied. Illegal <i>index</i> or <i>number</i> parameter.

Related Information

Additional Discussion	The GETDSEG intrinsic in this chapter and "Creating an Extra Data Segment" in Chapter 3.
Intrinsics	DMOVOUT, GETDSEG
Special Consideration	Data segment management (DS) capability required.

DMOVOUT

Copies data from the data area to an extra data segment. Data segment management (DS) capability is required. (Intrinsic number 133)

Syntax

LV	IV	IV	LA
DMOVOUT(<i>index,displacement,number,location</i>);			

Parameters

- | | |
|---------------------|---|
| <i>index</i> | <p>logical by value (required)</p> <p>The index of the extra data segment, obtained from a GETDSEG intrinsic call.</p> |
| <i>displacement</i> | <p>integer by value (required)</p> <p>The displacement, in words, of the first word in the string to be transferred from the first word in the data segment. This value must be ≥ 0.</p> |
| <i>number</i> | <p>integer by value (required)</p> <p>The size, in words, of the data string to be passed. This value must be ≥ 0.</p> |
| <i>location</i> | <p>logical array (required)</p> <p>Passes the data string to be moved to the extra data segment.</p> |



DMOVOUT

Operation Notes

When **DMOVOUT** is called, a bounds check is initiated to ensure that the data is taken from within an available data area and moved to an area within the extra data segment boundaries. For example, in Figure 2-2, to move four words from **ARA[10]** to the data segment whose **index** is 2 (from a **GETDSEG** call), starting at location 201 within the segment, the intrinsic call could be:

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

The **index** is 2; the **displacement** within the data segment is 201; the **number** of words to be moved to the data segment is 4; and the starting **location** of the data is the address of **ARA[10]**. If **ARA** is an array of words, the four words, from **ARA[10]** to **ARA[13]**, are moved to word offset 201 through 204 of the data segment, as shown in Figure 2-2.

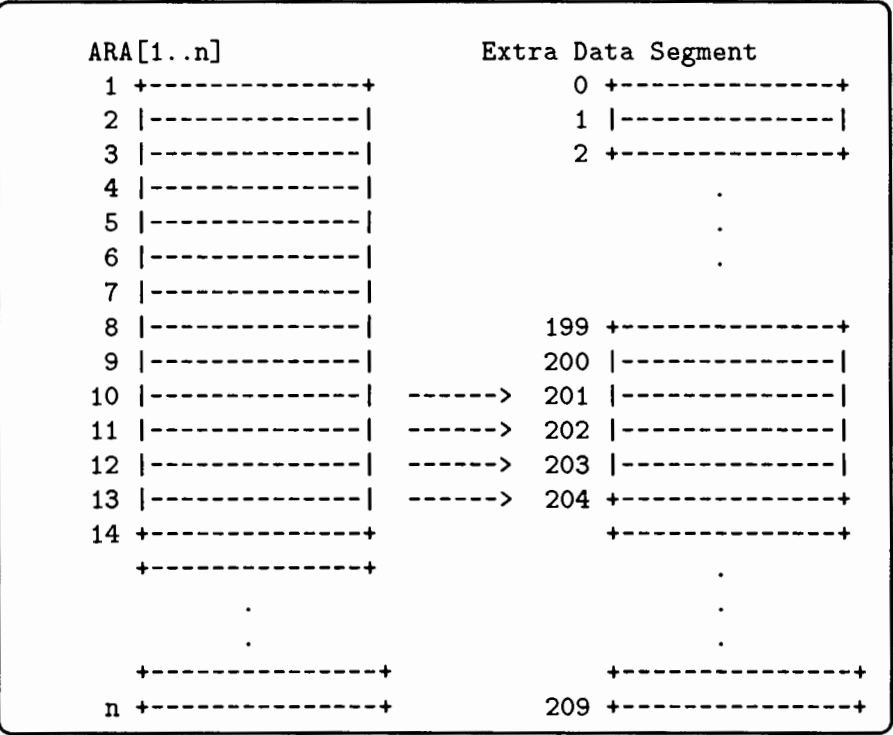


Figure 2-2. DMOVOUT Data Movement.

Condition Codes

- | | |
|-----|--|
| CCE | Request granted. |
| CCG | Request denied. Out of bounds-check failure. |
| CCL | Request denied. Illegal index or number parameter. |

Related Information

Additional
Discussion

The **GETDSEG** intrinsic in this chapter and “Creating an Extra Data Segment” in Chapter 3.

Intrinsics

DMOVIN

Special
Considerations

Data Segment Management (DS) capability required.

DUMPPMAP

Fills a file with external FPMAP records representing either all of the FPMAP data stored in a program or SL file, or only that for a specified segment. One external record is created as each FPMAP record is encountered.

Syntax

IV	IV	IV
DUMPPMAP(<i>progfnum</i> , <i>pmapfnum</i> , <i>xmapreclen</i> ,		
I	IV	I
<i>reccount</i> , <i>segnum</i> , <i>status</i>);		

Parameters

progfnum

integer by value

The number of the program or SL file whose FPMAP is to be dumped, as returned from the FOPEN intrinsic. The caller is responsible for opening the file with MULTIRECORD, READ access.

pmapfnum

integer by value

The number of the file that is to contain the external FPMAP records, as returned from the FOPEN intrinsic. The caller is responsible for opening the file with write access and records of size at least *xmapreclen* words.

xmapreclen

integer by value

The number of words in each external FPMAP record to be written to file *pmapfnum*. If the caller requests more words than are currently defined for an external FPMAP record, the extra words in each record written are set to zero.

reccount

integer by reference

The number of external FPMAP records which were (or could have been) generated from the FPMAP data found in the program or SL file. If the caller's *pmapfnum* file is large enough, it is also the number of records written to that file.

segnum

integer by value

The number of the segment whose FPMAP data is to be dumped. When *segnum* is -1, all FPMAP records for all segments in the file are dumped.

status

integer by reference

Returns a word that contains a Condition Code that tells you if the procedure call was successful and, if not, why it failed.

Condition Codes

	Error No.	Meaning
CCE	0	No errors detected.
CCG	4	External PMAP file was too small.
CCL	10	Program/SL file did not contain a PMAP.
	11	Program/SL file code was not that of a program/SL file, or, SL loader ID was not compatible with this version of the PMAP intrinsic.
	12	File system error on the program/SL file.
	13	File system error on the external PMAP file.

Related Information

Manual

MPE Segmenter Reference Manual (30000-90011).

ENDLOG

Posts a record to the logging file marking the end of a logical transaction. When the record is posted, **ENDLOG** flushes the user logging memory buffer to ensure that the record gets to the logging file. User logging (LG) or system supervisor (OP) capability is required. (No intrinsic number assigned)

Syntax

D	LA	I	I	I
ENDLOG(<i>index</i>,<i>data</i>,<i>length</i>,<i>mode</i>,<i>logstatus</i>);				

Parameters	<i>index</i>	double by reference (required) Passes the access to the user logging system. This is the value returned in the <i>index</i> parameter of the OPENLOG intrinsic.
	<i>data</i>	logical array (required) Passes the information to be logged. A log record contains 128 words, where 119 words are available for writing data. For the most efficient use of log file space, structure arrays with lengths in multiples of 119 words.
	<i>length</i>	integer by reference (required) Passes the length of the data in <i>data</i> . A positive value indicates words, while a negative value indicates bytes. If the length is greater than 119 words (or 238 bytes), the information in <i>data</i> is divided into two or more physical log records.
	<i>mode</i>	integer by reference (required) Passes a value indicating whether the logging process should suspend your process if it cannot complete your request for service immediately. If it is not possible to log the transaction and <i>mode</i> is set to 1 (NOWAIT), the ENDLOG intrinsic indicates through <i>logstatus</i> that it could not complete your request.

Value	Meaning
0	WAIT
1	NOWAIT

logstatus integer by reference (required)

Returns one of the following values, indicating the success/failure of the intrinsic call:

Value	Meaning
0	Successful
1	NOWAIT mode requested, and the user logging process is busy
2	Parameter out of bounds in logging intrinsic
3	Request to open or write to a logging process that is not running
4	Incorrect <i>index</i> parameter passed to a logging intrinsic
5	Incorrect <i>mode</i> parameter passed to a logging intrinsic
6	User request denied because logging process is suspended
7	Illegal capability. Must have user logging (LG) and system supervisor (OP) capabilities to use a logging intrinsic
8	Incorrect password passed to a logging intrinsic
9	Error occurred while writing to the logging file
10	Invalid DST passed to a logging system intrinsic
12	System out of disk space; user logging cannot proceed
13	No more logging entries
14	Invalid access to logging file
15	End-of-file encountered on user logging file
16	Invalid logging identifier

Operation Notes

Use the *data* parameter of this intrinsic to post user data to the logging file. This use of ENDLOG is similar to using the WRITELOG intrinsic, but also denotes the end of a logical transaction.

ENDLOG

Related Information	Additional Discussion	“User Logging” in Chapter 3.
	Intrinsics	WRITELOG, OPENLOG
	Special Consideration	User logging (LG) or system supervisor (OP) capabilities required.

EXPANDUSLF

Changes length of a USL file by creating a USL file with the *increment* length longer or shorter than the USL file specified by *uslfnum*. The old USL file is copied to the new file with the same file name; the old USL file is then deleted. (Intrinsic number 84)

Syntax

IIVIV
filenum:=EXPANDUSLF(*uslfnum*,*increment*);

Functional Return

filenum integer (optional)
Returns the new file number. If an error occurs (condition code returns CCL), an error number is returned. The condition code must be tested immediately upon return from this intrinsic. If an error number is used instead of a file number, in the *uslfnum* parameter, unpredictable results occur. Following are the possible error numbers and their meanings:

Value	Meaning
0	File specified by <i>uslfnum</i> was empty, an unexpected end-of-file was encountered when reading the old USL file, or an unexpected end-of-file was encountered when writing the new <i>uslfnum</i>
1	Unexpected I/O error occurred; can occur on the old USL file, or the new <i>uslfnum</i> where the information is being copied to
7	Unable to open the new USL file
8	Unable to close (purge) the old USL file
9	Unable to close (save) the new USL file
10	Unable to close \$NEWPASS
11	Unable to open \$OLDPASS

Parameters

uslfnum integer by value (required)
The file number of the file.
increment integer by value (required)
The number of records the length of the USL file is to be changed:
■ If a positive value, the new USL file is longer than the old USL.

EXPANDUSLF

- If a negative value, the new USL file is shorter than the old USL.

Condition Codes

CCE

Request granted. The new file number is returned.

CCG

Not returned.

CCL

Request denied. An error number was returned to *filenum*.

Related Information

Intrinsics

ADJUSTUSLF

Manuals

MPE Segmenter Reference Manual (30000-90011)

FATHER

Returns the process identification number (PIN) of the parent calling process. Process handling (PH) capability is required. (Intrinsic number 109)

Syntax

I
pin:=FATHER;

Functional Return*pin***integer**

Returns the process identification number (PIN) of the parent process.

Condition Codes

CCE

Request granted. The parent is a user process.

CCG

Request granted. The parent is a job or session main process (logon CI).

CCL

Request granted. The parent is a system process.

Related Information

Additional
Discussion

“Determining Father Process” in Chapter 3.

Special
Considerations

Split-stack calls permitted. Process Handling (PH) capability required.

FCARD

Drives the HP 7260A optical mark reader (OMR). (No intrinsic number assigned)

Syntax

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

Parameters

recode

integer by reference (required)

Passes positive integer represented as an input or output parameter.

As an input parameter, *recode* requests one of the following options (functions) G value:

- 0 Open the reader and terminal as a file and return the *filenum* to the program through SPL/3000 conventions.
- 1 Read a single card whether in ASCII or in column image format.
- 2 Select the previously read card by routing the card into the select output hopper (providing option 002 of the HP 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 the latest FCARD call issued specifying *recode* equal to 11 or 12.
- 4 Temporarily suspend the program awaiting an operator action (press the READY switch). This particular call to FCARD will maintain control and will not be completed until the operator presses the READY switch.
- 10 Cause the HP 7260A motor to come to a stop and deactivate MUTE for the associated terminal, if muted. When MUTE is activated and the HP 7260A is in its READY state, data transmission from the computer and from the HP 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*) 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 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* (S value) is described below:

- 0 Indicates that the call to FCARD, has been successfully performed. For the following conditions, when output *recode=0*, the specified parameters are significant to the program:
 - If the request was to open a file (*recode=0*), then *filenum* is significant.
 - 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.

- If the request was to select the previously read card (*recode*=2), then *status* is significant.
 - If the request was to perform a temporary suspension of the program (*recode*=4), then *status* is significant.
 - 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: 1, 2, 3, 4, 10, 11, 12, 13, 17, 18, or 20.
 - 2 Indicates that FCARD was unable to open the HP 7260A/terminal pair as a file. This error is not recoverable; the program should indicate an error and terminate itself.
 - 4 Indicates that FCARD has encountered a file read or write error while accessing the HP 7260A. This error is not recoverable; the program should indicate an error and proceed to a normal termination.
 - 5 Indicates that FCARD was unable to close the terminal file. This error is not recoverable; the program should indicate an error and proceed to a normal termination.
 - 6 Indicates that a logical end-of-data (:JOB, :EOJ, :EOD, and :DATA) was encountered while reading data in response to either a read or retransmit request.
 - 7 Indicates that FCARD has encountered a file error on a request for either enabling or disabling the echo function.
 - 8 Indicates that FCARD has detected a data dropout condition while the HP 7260A was transmitting. You should request a retransmission of the data or *status* (refer to *recode*=3).

<i>filenum</i>	integer by reference (required) Passes a word supplying the file number of the file associated with the reader/terminal file. This file number is returned to the program from FCARD with the output <i>recode</i> =0. It must be provided to FCARD on all requests.
<i>bufadr</i>	integer array (required) Contains the array to which the record is to be transferred. The declared length of <i>bufadr</i> should be set to 120 words.
<i>count</i>	integer by reference (required) Returns a positive integer to the program upon completion of a read (<i>recode</i> =1) or a retransmit (<i>recode</i> =3) request indicating the number of columns which have been transferred from the HP 7260A.
<i>status</i>	integer by reference (required) Returns an integer indicating whether the HP 7260A has successfully performed or responded to the read, select, retransmit, or temporary suspend request. If <i>status</i> is equal to zero, then the request has been successfully performed. If <i>status</i> is not equal to zero, then it contains an octal value specifying the HP 7260A condition. The values indicate the following conditions: %7 Input hopper empty or hopper full. Can either be returned upon a read request (<i>recode</i> =1) or upon a retransmit request, if there is no data to retransmit (<i>recode</i> =3). %11 Pick fail. Can either be returned upon a read request (<i>recode</i> =1) or upon a retransmit request, if there is no data to transmit (<i>recode</i> =3). %13 Select hopper full. Indicates that the OMR select hopper was full when the select request (<i>recode</i> =2) was issued. %14 Successful select. Indicates that the HP 7260A has successfully selected the card upon the select request (<i>recode</i> =2).

FCARD

- %22 *READY status*. Indicates that the HP 7260A *READY* push button has been pressed (*recode*=4). Would also indicate that the HP 7260A is ready but there is no data to be retransmitted input (*recode*=3).
- %37 Not ready. Can either be returned upon a read request (*recode*=1) or upon a retransmit request (*recode*=3). This *status* is provided by the HP 7260A if the operator has pressed the *STOP* button or if a lamp has burned out in the read head.

Operation Notes

The FCARD intrinsic allows you to control the operation of the Optical Mark Reader (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 HP 7260A, the file number of the HP 7260A/terminal file for which the function has been performed and the number of columns read at the completion of a read request.

For more details on the OMR *status* parameter, refer to the *HP 7260A Operating and Service Manual*(07260-90001).

Related Information

Additional Appendix B, "Device Characteristics".
Discussion

FCHECK

Returns specific details about error conditions that occurred when a file system intrinsic returns a condition code indicating an I/O error. FCHECK applies to files on any device. (Intrinsic number 10)

Syntax

0-VIVIIID

FCHECK(*filenum*,*fserrorcode*,*translog*,*blocknum*,*numrecs*);

Parameters	<i>filenum</i>	integer by value (optional) The file number of the file for which error information is to be returned. If <i>filenum</i> is not specified or set to zero, error information is returned about the last failed FOPEN call.
	<i>fserrorcode</i>	integer by reference (optional) by reference Returns a file system error code indicating the type of error that occurred. Refer to Table 2-2.
	<i>translog</i>	integer by reference (optional) Returns the number of words read or written if an I/O error occurred. (This value is recorded in the transmission log.)
	<i>blocknum</i>	double by reference (optional) Returns the physical record count for a nonspoolfile or the logical record count for a spoolfile: ■ For fixed-length and undefined-length record files, the physical count is the number of physical records transferred to or from the file since FOPEN. ■ For variable length record files, the physical count is the last rewind, rewind/unload, space forward or backward to tape mark.
	<i>numrecs</i>	integer by reference (optional) by reference by reference by reference by reference by reference by reference by reference by reference by reference by reference by reference by reference Returns the number of logical records in the bad block (blocking factor).

Table 2-2. File System Error Codes

Code (Decimal)	Meaning
0	Successful (no errors) or end-of-file (EOF).
1	Illegal DB register setting (typically, a request in split-stack mode when it is illegal).
2	Illegal capability.
3	Required parameter is missing.
4	Disk space allocation disabled on all disks in domain.
5	DRT number >511.
6	Device has no available spare blocks.
7	Unformatted or uninitialized media on device.
8	Illegal parameter value.
9	Invalid file type specified in <i>foptions</i> .
10	Invalid record size specification.
11	Invalid resultant block size.
12	Record number out of range.
13	Can't open file multiaccess, out of FMAVT entries.
16	More than 255 opens of a file.
17	Magnetic tape runaway.
18	Device powered up.
19	Forms control was 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.
27	I/O time-out.
28	Timing error or data overrun.
29	Start I/O (SIO) failure.
30	Unit failure.
31	End-of-line (EOL) special character terminator.
32	Software abort of input/output operation.
33	Data lost.
34	Unit not online.
35	Data-set not ready.
36	Invalid disk address.
37	Invalid memory address.
38	Tape parity error.
39	Recovered tape error.
40	Operation inconsistent with access type.
41	Operation inconsistent with record type.
42	Operation inconsistent with device type.
43	Write exceeds record size.
44	Update at record zero.
45	Privileged file violation.

Table 2-2. File System Error Codes (continued)

Code (Decimal)	Meaning
46	Out of disk space.
47	Input/output error on a file label.
48	Invalid operation due to multiple file access.
49	Unimplemented function.
50	Nonexistent account.
51	Nonexistent group.
52	Nonexistent permanent file.
53	Nonexistent temporary file.
54	Invalid file reference.
55	Device unavailable.
56	Invalid device specification.
57	Out of virtual memory.
58	No passed file.
59	Standard label violation.
60	Global RIN not available.
61	Out of group disk space.
62	Out of account disk space.
63	User lacks non-sharable device capability.
64	Program not prepped with multiple RIN capability.
65	Punch hopper empty.
66	Plotter limit switch reached.
67	Paper tape error.
68	Insufficient system resources.
69	I/O error.
70	I/O error while printing header/trailer.
71	Too many files open.
72	Invalid file number.
73	Bounds violation.
74	No room left in stack segment for another file entry.
76	Input buffer absent in IOWAIT.
77	NOWAIT input/output operation is pending.
78	No NOWAIT I/O pending for any file.
79	No NOWAIT I/O pending for any special file.
80	Spoolfile size exceeds configuration.
81	No SPOOL class defined in system.
82	Insufficient space in SPOOL class to honor this input/output request.
83	I/O error on spoolfile.
84	Device unavailable for spoolfile.
85	Operation is inconsistent with spooling, (e.g. attempt to read hardware status).
86	Spooling internal error.
87	Bad spoolfile block.
88	Nonexistent spoolfile.
89	Power failure.
90	Exclusive violation: file being accessed.

Table 2-2. File System Error Codes (continued)

Code (Decimal)	Meaning
91	Exclusive violation: file being accessed.
92	Lockword violation.
93	Security violation.
94	User is not creator.
95	Read not completed due to BREAK.
96	Disk I/O error.
97	No CONTROL-Y PIN.
98	Read time overflow.
99	BOT and backspace tape.
100	Duplicate permanent file name.
101	Duplicate temporary file name.
102	I/O error on directory.
103	Permanent directory overflow.
104	Temporary directory overflow.
105	Bad variable block structure.
106	Extent size exceeds maximum.
107	Insufficient space for user labels.
108	Invalid file label.
109	Invalid carriage control.
110	Attempt to save permanent file as temporary.
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 rejected.
116	Invalid tape label FOPEN parameters.
117	Attempt to write on an unexpired tape file.
118	Invalid header or trailer tape label.
119	I/O error positioning tape for tape labels.
120	Attempt to write IBM standard tape label.
121	Tape label lockword violation.
122	Tape label table overflow.
123	End of tape volume set.
124	Attempt to append labeled tape.
125	Expiration date can't be later than that of preceding file.
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").

Table 2-2. File System Error Codes (continued)

Code (Decimal)	Meaning
133	Picture number must be between 0 and 31.
134	Extended capability parm must be 0 (OFF) or 1 (ON).
137	Defective track on foreign disk.
138	Track does not exist on foreign disk.
139	Deleted record on IBM diskette.
148	Inactive RIO record.
149	Missing item number or return variable.
150	Invalid item number.
151	Undefined file type.
152	Unrecognized key word in <i>FOPEN device</i> parameter.
153	Expecting “;” or “carriage return” in <i>FOPEN 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 <i>FDEVICECONTROL</i> from environment file.
161	Too many parameters in device string overflow.
162	Expecting “=” after keyword in <i>device</i> parameter.
163	“ENV” backreference in file equation incorrect.
164	The <i>device</i> parameter too large or missing carriage return.
165	Invalid density specification.
166	<i>FFILEINFO</i> failed in accessing remote spoolfile.
167	Spoolfile label error, can't insert environment file name.
168	Item not supported on remote system.
170	The record is marked deleted. <i>FPOINT</i> positioned pointer to a record that was marked for deletion.
171	Duplicate key value (KSAM error).
172	No such key (KSAM error).
173	The <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

Table 2-2. File System Error Codes (continued)

Code (Decimal)	Meaning
181	handle the file built by previous version.
182	Invalid key starting position (KSAM error).
183	File is empty (KSAM error).
184	Record does not contain all keys (KSAM error).
185	Invalid record number (KSAM FFINDN intrinsic error).
186	Sequence error in primary key (KSAM error).
187	Invalid key length (KSAM error).
188	Invalid key 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 (KSAM error).
192	System failure occurred while KSAM file was opened.
193	\$STDIN/\$STDLIST cannot be redirected to KSAM files.
194	KSAM files not allowed for global AFTs.
195	Global files cannot be remote files.
196	Language not supported (KSAM error).
197	Native language internal error (KSAM error).
198	Invalid version number in KSAM file (KSAM error).
201	Invalid ID sequence.
202	Invalid telephone number (CS error).
203	No telephone list specified (CS error).
204	Unable to allocate an extra data segment for DS/DSN 3000.
205	Unable to expand the DS/DSN 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 time-out.

Table 2-2. File System Error Codes (continued)

Code (Decimal)	Meaning
246	Communications interface error. Remote computer has disconnected.
247	Communications interface error. Local time-out.
248	Communications interface error. Connect time-out.
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. No response to dial request by the operator.
254	Communications interface error. Invalid input/output configuration.
255	Communications interface error. Unanticipated condition.
302	Invalid item number for FDEVICECONTROL.
303	Invalid access for item number to FDEVICECONTROL.
304	Attempt to change terminal parity in 8-bit mode.
305	Invalid format in terminal configuration file.
306	Checksum error in terminal configuration file.
307	Passed value to FDEVICECONTROL less than minimum.
308	Passed value to FDEVICECONTROL greater than maximum.
309	Passed value to FDEVICECONTROL is unsupported.
310	Count to FDEVICECONTROL insufficient to return information.
311	Count to FDEVICECONTROL greater than available store information.
312	Passed special character has previously defined function.

Operation Notes

FCHECK is used to determine the error conditions of the last failed FOPEN intrinsic call (even if a file number was not returned) by setting the *filenum* parameter to zero. In this case, only *fserrorcode* returns valid information.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. The file number passed by <i>filenum</i> is invalid, or a bounds violation occurred while processing this request (<i>fserrorcode</i> is 73).

Related Information

Special Considerations	Split-stack calls permitted.
------------------------	------------------------------

FCLOSE

Terminates access to a file on any device. (Intrinsic number 8)

Syntax

IV	IV	IV
FCLOSE(<i>filenum</i> , <i>disposition</i> , <i>securitycode</i>);		

Parameters

filenum

integer by value (required)

Passes the file number of the file to be closed.

disposition

integer by value (required)

Passes the disposition of the file, significant only for files on disk and magnetic tape. This ***disposition*** can be overridden by a corresponding parameter in a FILE command entered prior to program execution. The ***disposition*** options are:

Bits	Value/Meaning
------	---------------

13:3	Domain disposition:
------	---------------------

- | | |
|-----|--|
| 000 | No change. The disposition remains as it was before the file was opened. If the file is new, it is deleted by FCLOSE; otherwise, the file is assigned to the domain it belonged to previously. An unlabeled tape file is rewound and a labeled tape is rewound and unloaded. |
| 001 | Close as a permanent file. If the file is a disk file, it is saved in the system file domain. A new or old temporary file on disk has an entry created for it in the system file directory. If a file of the same name already exists in the directory, an error code is returned and the file remains open. If the file is a permanent file on disk, this domain disposition has no effect. If the file is stored on magnetic tape, the tape is rewound and unloaded. |

Bits Value/Meaning**13:3 Domain disposition (Continued):**

- 010 Close as a temporary job file (rewound). The file is retained in your temporary (job/session) file domain and can be requested by any process within your job/session. If the file is a disk file, the file name is checked. If a file of the same name already exists in the temporary file domain, an error code is returned and the file remains open. If a 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 opened file.
- 011 Close as a temporary job file (not rewound). This option has the same effect as domain disposition 010, except that tape files are not rewound. In the case of unlabeled magnetic tape, if this FCLOSE is the last done on the device (with no other FOPEN calls outstanding), the tape is rewound and unloaded. If the file resides on a labeled magnetic tape, the tape is positioned to the beginning of the next file on the tape.
- 100 Release the file. The file is deleted from the system.

FCLOSE

Bits	Value/Meaning
12:1	Disk space disposition (valid only for fixed-length, undefined-length, or variable-length record formats): 0 Does not return any disk space allocated beyond the end-of-file marker. 1 Returns any disk space allocated beyond the end-of-file (EOF) marker to the system. The EOF becomes the file limit; records cannot be added to the file beyond the EOF.
0:11	Reserved for MPE and should be set to zero.

When a file is opened with the **FOPEN** intrinsic, the file count (maintained by MPE for each file) is incremented by one; when a file is closed, 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; the saved disposition is the smallest nonzero disposition parameter specified for all **FCLOSE** calls issued against the file.

For example, the file **XYZ** is opened three successive times by a process. The first **FCLOSE** disposition is 1, the second is %14, and third is %12. The third disposition on the file **XYZ** is the return of disk space.

securitycode integer by value (required)

The type of security initially applied to the file (significant for new permanent files only). The valid options are:

Value	Meaning
0	Unrestricted access; can be accessed by any user, unless prohibited
1	Private file creator security; can be accessed only by the creator

Operation Notes

FCLOSE deletes buffers and control blocks where the process accessed the file. It also deallocates the device where the file resides, and it can change the **disposition** of the file. If FCLOSE calls are not issued for all files opened by the process, the calls are issued automatically by MPE when the process terminates. All magnetic tape files are left offline after FCLOSE calls, to indicate that the system operator can remove them.

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 the file is closed with a **disposition** code of 3, the tape does not rewind but remains positioned at the next file.
- If the file is closed 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 if the sequence (*seq*) subparameter of *formmsg* in FOPEN specifies NEXT or the default (1).
- A **disposition** code of 1 (rewind and unload) implies the close of an entire volume set.
- If unlabeled magnetic tape is closed with a **disposition** code of 0, 1, or 4, and the tape was written to while open, FCLOSE writes three EOFs at the end of the tape before performing a rewind or rewind/unload. This ensures that all tapes have an acceptable number of EOF marks at the end. The three EOFs are written only after the last FCLOSE occurs, before the rewind, and only if the tape was written on.
- For circular files, deletion of disk space beyond the end-of-file is not allowed.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. The file was not closed; an incorrect filenum was specified or another file with the same name and disposition exists. Any outstanding write I/Os that failed (for example, buffered writes that are done in background) or an illegal disposition (6 or 7) may have been specified; FCHECK detects this and returns error 49.

**Related Information**

Additional Discussion	"Closing Files", "File Domains" and "Internal Operations for File Accessing" in Chapter 4.
Intrinsics	FOPEN
Manuals	MPE File System Reference Manual(30000-90236).

FCONTROL

Performs various control operations on a file or on the device where the file resides, including:

- Supplying a printer or terminal carriage control directive.
- Verifying I/O.
- Reading the hardware status word for the device where the file resides.
- Setting a terminal's timeout interval.
- Repositioning a file at its beginning.
- Writing an end-of-file marker.
- Skipping forward or backward to a tape mark.

(Intrinsic number 13)

Syntax

IV IV L
FCONTROL(<i>filenum</i> , <i>itemnum</i> , <i>item</i>);

Parameters	<i>filenum</i>	integer by value (required) The file number of the file for which the control operation is to be performed.
	<i>itemnum</i>	integer by value (required) Specifies which operation is to be performed. (Refer to Table 2-3.
	<i>item</i>	logical by reference (required) Passes/returns a value associated with a control operation as indicated by the corresponding <i>itemnum</i> parameter. (Refer to Table 2-3.)

Table 2-3. FCONTROL Itemnum/Item Values

Itemnum	Mnemonic	Item Description
0	L	General device control: The value specified is passed to the appropriate device driver. A value from the driver is returned in item. Rejected for spooled device files.
1	U	Line control: Passes a carriage control directive to the #1 driver of the specified nonspooled device file, regardless of whether the file is opened CCTL or NOCCTL. For spooled devices, use FWRITE to pass carriage control directives (refer to the FWRITE intrinsic) If <i>item</i> passes either the prespacing (%400 or %100) or postspacing (%401 or %101) carriage control directives, one of the following values will be returned in <i>item</i> indicating the setting of the device prior to the FCONTROL call: 0 Postspacing 1 Prespacing
2	I	Complete I/O: Ensures that the requested I/O has been physically completed. Valid only for buffered files. Posts the block being written (full or not). Ignored for message files. Any <i>item</i> identifier is ignored.
3	L	Device status: Returns a record containing information about the state of the device associated with the file immediately after the last I/O operation (including FOPEN) on the file. The record size and contents are device-dependent. Rejected for spooled device files.
4	L	Set timeout interval: Indicates a timeout interval is to be applied to input from a file. If input is requested from a file but is not received in this interval, the FREAD request terminates prematurely with CCL. The interval is specified in seconds and returned in <i>item</i> . If this interval is zero, any previously established interval is cancelled, and no timeout occurs. Only valid for terminal and message files. Only affects the next read and is ignored if the addressed file is not being read from the terminal. If this code is applied to an interprocess communication (IPC) file, <i>item</i> specifies the length of time that a process waits when reading from an empty file or writing to a full one and the timeout remains enabled until it is explicitly cancelled. Denotes a word in the stack that contains the time-out interval, in seconds, to be applied to input from the terminal.

FCONTROL

Table 2-3. FCONTROL Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
5	L	Reposition file at its beginning: The next record, read or written, is the first record in the file. Not valid for files accessed as APPEND-ONLY. On a labeled magnetic tape file or serial disk file, the tape is positioned at the beginning of the opened file, and not necessarily at the beginning of the volume. Rejected for spooled DATA tapes. Any <i>item</i> identifier is ignored.
6	L	Write end-of-file: Marks the end-of-file (EOF) on disk or magnetic tape and is effective only for those devices: ■ If used for a disk file it writes a logical end-of-data marker where the file was last accessed. The disk file label is updated and written to disk. ■ If used for a file residing on unlabeled magnetic tape, a tape mark is written at that current position. ■ If used for interprocess communication (IPC), it verifies the state of the file by writing the file label and buffer area to disk; ensures the message file can survive system crashes. No EOF is written. Rejected for spooled DATA tapes and labeled magnetic tape files. Is any variable identifier. FCONTROL needs this parameter to satisfy the internal requirements of the intrinsic. It serves no other purpose and is not modified by the intrinsic.
7	L	Space forward to tape mark: Moves a magnetic tape forward until a tape mark is encountered. If used on labeled magnetic tapes, the tape is positioned at the beginning of user trailer labels, if any. Rejected for spooled DATA tapes. Any <i>item</i> variable identifier is ignored.
8	L	Space backward to tape mark: On unlabeled tapes, moves a magnetic tape backward until a tape mark is encountered. If used on labeled tapes, the tape is positioned at the beginning of user header labels, if any. Rejected for spooled DATA tapes. Any <i>item</i> variable identifier is ignored.
9	L	Rewind and unload tape: Repositions a magnetic tape file at its beginning and places the tape offline. Not allowed for labeled tapes. Rejected for spooled DATA tapes and labeled magnetic tape files. Is any variable identifier. FCONTROL needs this parameter to satisfy the internal requirements of the intrinsic. It serves no other purpose and is not modified by the intrinsic.

Table 2-3. FCONTROL Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
10	L	Change terminal input speed: Not reset in BREAK mode; not reset after FCLOSE.
11	L	Change terminal output speed: Not reset in BREAK mode; not reset after FCLOSE. (<i>item</i> is ignored)
12	L	Turn ECHO facility on: Not reset in BREAK mode; not reset after FCLOSE. (Default) (<i>item</i> is ignored)
13	L	Turn ECHO facility off: Not reset in BREAK mode; not reset after FCLOSE. (<i>item</i> is ignored)
14	L	Disable the system BREAK function: Reset after FCLOSE. (<i>item</i> is ignored)
15	L	Enable the system BREAK function: Reset after FCLOSE. (Default) (<i>item</i> is ignored)
16	L	Disable the subsystem BREAK function: Reset in BREAK mode; reset after FCLOSE. (Default) (<i>item</i> is ignored)
17	L	Enable the subsystem BREAK function: Reset in BREAK mode; reset after FCLOSE. (<i>item</i> is ignored)
18	L	Disable tape option (Default) (<i>item</i> is ignored)
19	L	Enable tape option (<i>item</i> is ignored)
20	L	Disable the terminal input timer: Reset in BREAK mode; reset after FCLOSE. (Default) (<i>item</i> is ignored)
21	L	Enable the terminal input timer: Reset in BREAK mode; reset after FCLOSE. (<i>item</i> is ignored)
22	L	Read the terminal input timer (<i>item</i> is ignored)
23	L	Disable parity checking (Default) (<i>item</i> is ignored)
24	L	Enable parity checking (<i>item</i> is ignored)
25	L	Define line termination characters for terminal input: Reset in BREAK mode. (<i>item</i> is ignored)
26	L	Disable binary transfers (Default) (<i>item</i> is ignored)
27	L	Enable binary transfers (<i>item</i> is ignored)
28	L	Disable user block mode transfers: Not reset in BREAK mode; not reset after FCLOSE. (Default) (<i>item</i> is ignored)
29	L	Enable user block mode transfers: Not reset in BREAK mode; not reset after FCLOSE. (<i>item</i> is ignored)
34	L	Enable line deletion echo suppression: Not reset in BREAK mode; reset after FCLOSE. (Default) (<i>item</i> is ignored)
35	L	Disable line deletion echo suppression: Not reset in BREAK mode; reset after FCLOSE. (<i>item</i> is ignored)
36	L	Set parity (<i>item</i> is ignored)

FCONTROL

Table 2-3. FCONTROL Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
37	L	Allocate a terminal (<i>item</i> is ignored)
38	L	Set terminal type (<i>item</i> is ignored)
39	L	Obtain terminal type information (<i>item</i> is ignored)
40	L	Obtain terminal output speed (<i>item</i> is ignored)
41	L	Set unedited terminal mode: Reset in BREAK mode. (<i>item</i> is ignored)
43	L	Aborts pending NOWAIT I/O request: For interprocess communication, CCG is returned when an outstanding I/O operation has completed. An IOWAIT call must be issued to finish the request.
45	L	Enable/disable extended WAIT: Applicable only if <i>flenum</i> specifies a message file. For interprocess communication, an <i>item</i> value of 1 enables 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 remains in effect until an FCONTROL call is issued with an <i>itemnum</i> 45 and an <i>item</i> value of 0. An <i>item</i> value of 0 specifies that if a second FREAD call is issued and it encounters an empty file that has no reader, it returns an end-of-file condition. Default: 0

Table 2-3. FCONTROL Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
46	L	Enable/disable reading writer's ID: Applicable only if <i>filenum</i> specifies a message file. For interprocess communication, an <i>item</i> value of 1 enables reading the writer's ID. Each record read has a two-word header. The first word indicates the type of record: 0 Data record 1 Open record 2 Close record The second word contains the writer's ID number. If the record is a data record, the data follows the header. Open and close records contain no more information. If the <i>item</i> value is 0, reading the writer's ID is disabled. 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: 0
47	L	Nondestructive read: Applicable only if <i>filenum</i> specifies a message file. If the <i>item</i> value is 1, the next FREAD by this reader does not delete the record. Subsequent FREAD calls are unaffected. When <i>item</i> is specified as 0, the next FREAD deletes the record. Default: 0
48	U32	Arm/disarm software interrupts: Passes a value that arms/disarms the software interrupt mechanism. (Valid only if <i>filenum</i> specifies a message file). To arm the files software interrupt, pass the external-type label (<i>plabel</i>) of a user-written interrupt procedure. To disarm the software interrupt pass a value of zero.

FCONTROL

Note



Itemnums 10 through 41 are used in changing terminal characteristics. Included with the definition of the code is an indication, where applicable, of whether the characteristic is reset in BREAK mode or after FCLOSE. Characteristics that are reset in BREAK mode are restored when there is a RESUME from BREAK mode.

Note



All descriptions pertaining to magnetic tape are also valid for serial disk.

Operation Notes

The FCONTROL intrinsic applies to files on disk, tape, terminal, or line printers. There are some special conditions that exist when using FCONTROL with files on labeled magnetic tape. Some FCONTROL functions cannot be used with labeled tapes, and other functions can produce unexpected results. (Refer to *itemnums* 5, 6, 7, 8, and 9.)

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. An error occurred.

Related Information

Manuals	<i>MPE File System Reference Manual</i> (30000=90236) and <i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250).
Special Considerations	Split-stack calls permitted.

FDELETE

Deactivates a specified logical record in an RIO record. (No intrinsic number assigned)

Syntax

O-V	IV	DV
FDELETE(<i>filenum</i> , <i>recnum</i>);		

Parameters

<i>filenum</i>	<i>integer by value (required)</i> Passes the file number of the file to be modified.
<i>recnum</i>	<i>double by value (optional)</i> Passes the relative logical record to be modified.

Operation Notes

If a record is not specified or the *recnum* is negative, the next logical record becomes inactive. If the selected record has already been deactivated, CCE is returned. Check to see if an RIO record is inactive by calling the FCHECK intrinsic.

Only applicable for RIO files. Split-stack calls permitted.

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 occurred.

Related Information

Manual	<i>MPE File System Reference Manual</i> (30000-90236).
--------	--

FDEVICECONTROL

Provides control operations to a printer, terminal, or a spooled devicefile and is used to:

- Download character sets, forms, and internal or control tables used in printing.
- Control the page size, pen positioning, form and use of character sets, the number of copies to be printed, and all other printing environment characteristics.

Syntax

	IV	LA	IV	LV					
FDEVICECONTROL	(	<i>filenum</i>	,	<i>buffer</i>	,	<i>length</i>	,	<i>controlcode</i>	
		LV		LV				I	
				<i>parm1</i>	,	<i>parm2</i>	,	<i>fserrorcode</i>	) ;

Parameters	<i>filenum</i>	integer by value (required) Passes the file number of the devicefile.
	<i>buffer</i>	logical array (required) Passes and returns device information: ■ Passes information to the device, such as character sets, forms, or vertical format control (VFC) information. ■ Returns device driver information.
	<i>length</i>	integer by value (required) Passes the length of <i>buffer</i> : ■ If positive, it indicates the length in words. ■ If negative, it indicates the length in bytes.
	<i>controlcode</i>	logical by value (required) The item value to be returned. (Refer to Table 2-4.)
	<i>parm1</i> , <i>parm2</i>	integer by value (required) The value specified in the corresponding <i>controlcode</i> ; there are several values for <i>parm1</i> and <i>parm2</i> that define the operation in more detail. (Refer to Table 2-4.)

fserrorcode integer by reference (required)

Returns a file system error Code number if an error occurs. If no error occurs, *fserrorcode* is set to zero. If FDEVICECONTROL detects a bounds violation for *fserrorcode* (that is, an address outside your stack area), *fserrorcode* is returned unchanged.

Table 2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values

Controlcode	Mnemonic	Parm1,Parm2 Description
128	IV	Character set selection: <i>parm1</i> (8:8) Primary character set identification <i>parm2</i> (8:8) Secondary character set identification The HP 268x page printer can contain up to 32 character sets, allowing the use of a variety of fonts, styles, print rotations, and languages. <i>Controlcode</i>=134 is used to download character sets to the printer or to select any two downloaded character sets to be the current primary and secondary character sets. To change to the secondary character set (one character at a time), set the eighth bit of the byte coding for the desired ASCII character. The HP 268x strips out this bit and prints (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 CTRL N (shift-in character) in the data. Insert a CTRL O (shift-out character) where the primary character set is to be reactivated.
129	IV	Logical page activation/deactivation request: <i>parm1</i> (0:1) 0 = Ignore the left byte of <i>parm2</i>. 1 = Deactivate the logical page table entry identified in the left byte of <i>parm2</i>. <i>parm1</i> (1:1) 0 = Ignore the right byte of <i>parm2</i>. 1 = Activate the logical page table entry identified in the right byte of <i>parm2</i>. <i>parm2</i> (0:8) Logical page table entry (0 to 31) to be deactivated. Ignored if <i>parm1</i> (0:1)=0. <i>parm2</i> (0:8) Logical page table entry (0 to 31) to be activated. Ignored if <i>parm1</i> (1:1)=0. Logical pages being printed can be canceled or enabled through the activation or deactivation of those pages. Every physical page is composed of one or more logical pages. When the HP 268x begins to print each physical page, it scans the logical page table (LPT) 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 HP 268x performs a physical page eject and starts the sequence again. There must be at least one active LPT entry while the HP 268x is printing.

FDEVICECONTROL

Table 2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values (continued)

Controlcode	Mnemonic	Parm1,Parm2 Description
130	IV	Relative pen displacement: <i>parm1</i> A integer containing the desired X-axis displacement, in dots, of the pen from its current position. <i>parm2</i> A integer containing the desired Y-axis displacement, in dots, of the pen from its current position. No pen movement results from requests to move the pen off the logical page. Because the coordinate system is based upon the current logical page itself and not upon the page's orientation with respect to the printer, do not consider how the page has been rotated when assigning displacement values to <i>parm1</i> and <i>parm2</i>. Since the dot density for the HP 2680 (180 dots per inch) differs from the HP 2688 (300 dots per inch), the effects of <i>parm1</i> and <i>parm2</i> are different.
131	IV	Absolute pen move: <i>parm1</i> An integer containing the X-coordinate, in dots, of the point to move the pen. <i>parm2</i> An integer containing the Y-coordinate, in dots, of the point to move the pen. The values in <i>parm1</i> and <i>parm2</i> are measured from the upper left corner of the logical page. Do not take page rotation into account when assigning coordinates as the printer does not move the pen if the location specified is off the logical page. Since the dot density for the HP 2680 (180 dots per inch) differs from the HP 2688 (300 dots per inch), the effects of <i>parm1</i> and <i>parm2</i> are different.
132	IV	Define job characteristics: <i>parm1</i> (0:1) 0 = Ignored 1 = Printer does not print job separation marks until next open job <i>parm1</i> (1:1) 0 = Ignored 1 = Parm2 contains maximum allowable number of copies <i>parm2</i> Significant only if <i>parm1</i> (1:1)=1 (Specifies the maximum number of copies the printer makes, for any one page, for the current job. The default maximum is 32,767.)

Table 2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values (continued)

Controlcode	Mnemonic	Parm1,Parm2 Description
133	IV	Define physical page. The following bits are ignored if set to zero: <i>parm1</i> (0:1) 1 = Multicopy form overlay feature on <i>parm1</i> (1:1) 1 = Multicopy form overlay feature off <i>parm1</i> (2:1) 1 = Reserved for MPE <i>parm1</i> (3:1) 1 = Redefine physical page length <i>parm1</i> (4:1) 1 = Redefine number of copies per page <i>parm1</i> (5:1) 1 = Reserved for MPE <i>parm1</i> (6:1) 1 = Reserved for MPE <i>parm1</i> (7:1) 1 = Reserved for MPE <i>parm1</i> (8:8) New physical page length in units of 0.25 inches; cannot be less than 3.0 inches (a value of 12) or greater than 17.0 inches (a value of 68) <i>parm2</i> Number of copies to print; if number exceeds the maximum defined in <i>parm2</i> of controlcode 132, only the maximum number of copies is printed Although FDEVICECONTROL accepts page length values that are multiples of 0.25 inches, the HP 268x printer is able to produce only pages that are multiples of 0.5 inches. For this reason, use only even values in <i>parm1</i> (8:8). In other words, bit (15:1)=0.
134	IV	Download/delete character set: <i>parm1</i> (0:1) 0 = Download character set identified in right-hand byte of <i>parm2</i>, HP 268x 1 = Purge character set identified in right-hand byte of <i>parm2</i>, HP 268x <i>parm2</i> (0:1) 0 = First record of a load 1 = Continuation of previous record <i>parm2</i> (8:8) Character set identifier; integer from 0 to 31 If there is an attempt to download a character set having the same identifier as one existing in the printer, then the HP 268x purges the existing character set and repacks the user area before loading the new font. However, before the modification of the user area, the HP 268x prints all data currently in its buffer, as it does when a character set, form, or vertical format control set (VFC) is loaded, overlayed, or deleted.

FDEVICECONTROL

Table 2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values (continued)

Controlcode	Mnemonic	Parm1,Parm2 Description
135	IV	Download/delete form: <i>parm1</i> (0:1) 0 = Load form set identified in the right-hand byte of <i>parm2</i> 1 = Purge identified form set from the HP 268x printer's memory <i>parm2</i> (0:1) 0 = First record of a load 1 = Continuation of previous record <i>parm2</i> (8:8) Form set identifier; integer from 0 to 31 If attempting to download a form set having the same identifier as one existing in the printer, then the HP 268x purges the existing form set and repacks the user area before loading the new form. However, before the modification of the user area, the HP 268x 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 (VFC).
136	IV	Download logical page table: <i>parm1</i> Not used <i>parm2</i> (0:1) 0 = First record of a load 1 = Continuation of previous record A logical page is a page of data that may take up an entire sheet of paper. It is possible to print up to eight 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 eight logical pages printed on any one physical page can be varied.
137	IV	Download multicopy form overlay table: <i>parm1</i> Not used <i>parm2</i> Not used Prints up to eight 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 eight possible copies to be overlaid with a form. The format of each word of the table is: Bit (0:1) 0 = Ignored 1 = Overlay form1 on physical page Bit (1:1) 0 = Ignored 1 = Overlay form2 on physical page Bits (2:4) Reserved for MPE Bits (6:5) Form1 identifier; integer from 0 to 31 Bits (11:5) Form2 identifier; integer from 0 to 31

Table 2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values (continued)

Controlcode	Mnemonic	Parm1,Parm2 Description
138	IV	Download/delete vertical format control (VFC); <i>parm1</i> (0:1) 0 = Load VFC 1 = Delete VFC <i>parm2</i> (0:1) 0 = First record of a load 1 = Logical continuation of the previous record <i>parm2</i> (8:8) VFC set identifier; integer from 0 to 31 The VFC table is an ASCII file downloaded to the HP 268x printer in order to give specific instructions on the print density, location of the top of the page, the bottom of the page, and other specifications of the printed page. The HP 268x 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, use the same or different VFC tables for logical pages of different rotations.
139	IV	Download/delete a picture: <i>parm1</i> (0:1) 0 = Load picture 1 = Delete picture <i>parm2</i> (0:1) 0 = First record of a load 1 = Logical continuation of the previous record <i>parm2</i> (8:8) Picture identifier; integer from 0 to 31 When the picture is downloaded to the HP 268x, it changes every pointer to reflect where the dot per bit symbol actually is in memory. If a picture is downloaded and one is already present with the same identifier (0-31), then the original one is overwritten in the picture descriptor block (PDB). The area taken up by the deleted picture is freed as soon as the page has been transferred to paper.
140	IV	Page control: <i>parm1</i> (15:1) 0 = Do not eject physical page 1 = Eject physical page before going to logical page (no effect if the first record since an environment load, FOPEN/FCLOSE) <i>parm1</i> (13:2) Auto-eject mode: 00 = Use auto-eject flag of last data record (default at start; auto-eject enabled) 01 = Enable auto-eject (select VFC channel 1 on new page) 11 = Disable auto-eject (position pen at top of page) <i>parm2</i> (8:8) Logical page number; integer from 0 to 31 The logical page identified in <i>parm2</i> becomes the current logical page even if other logical pages have entries that precede it in the logical page table. FDEVICECONTROL activates the specified page if it is inactive, and the HP 268x performs a physical page eject if <i>parm1</i> (15:1)=1.

FDEVICECONTROL

Table 2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values (continued)

Controlcode	Mnemonic	Parm1,Parm2 Description
141	IV	Clear environment: When set to zero, the following bits are ignored. <i>parm1</i> (0:1) 1 = Clear all character sets <i>parm1</i> (1:1) 1 = Clear all forms <i>parm1</i> (2:1) 1 = Clear all vertical format controls (VFCs) <i>parm1</i> (3:1) 1 = Clear all pictures <i>parm2</i> Not used The printer flushes all data currently in its buffers and then performs the indicated clears, if any.
142		Reserved for MPE
143	IV	Load default environment: <i>parm1</i> Not used <i>parm2</i> Not used The HP 268x 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).
144	IV	Print picture: <i>parm1</i> (0:1) 0 = Temporary picture 1 = Addressable picture <i>parm1</i> (1:1) 0 = X and Y relative to the current pen position 1 = X and Y are absolute pen position to the logical page <i>parm1</i> (2:14) X coordinate for picture placement (radixed integer) <i>parm2</i> (0:1) 0 = First temporary picture load 1 = Continuation record for load <i>parm2</i> (1:1) Not used
145	IV	End of job: <i>parm1</i> Not used <i>parm2</i> Not used

Table 2-4. FDEVICECONTROL Controlcode/Parm1,Parm2 Values (continued)

Controlcode	Mnemonic	Parm1,Parm2 Description
146	IV	Device extended capability mode: <i>parm1</i> 0 = Clear 1 = Set <i>parm2</i> Not used
192	DV	Access Workstation Configurator terminal configuration file. <i>parm1</i> Item number of function. Refer to the <i>Workstation Configurator Reference Manual (30239-90001)</i> for values. <i>parm2</i> Access code ■ 1 = Return current value in <i>buffer</i>. ■ 2 = Set item to value in <i>buffer</i>. ■ 3 = Set item to value in <i>buffer</i>, and return current value in <i>buffer</i>
193	IV	Record processing information for NRJE spoolfiles: <i>parm1</i> Indicates the character code (for example, ASCII) of spoolfile data. Refer to <i>SNA NRJE Network Remote Job Entry User/Programmer Reference Manual (30245-90001)</i>. <i>parm2</i> (14:1) 0 = Data not compacted 1 = Data compacted <i>parm2</i> (15:1) 0 = Data not compressed 1 = Data compressed

Condition Codes	CCE	Request granted.
	CCG	Not returned.
	CCL	Request denied. An error occurred.

Related Information	Manuals	<i>Point-to-Point Workstation I/O Reference (30000-90250)</i>
----------------------------	---------	---

FERRMSG

Returns a message corresponding to an FCHECK error number and enables error messages to be displayed from a program. (Intrinsic number 305)

Syntax

0-V	I	LA	I
FERRMSG(<i>ferrorcode</i> , <i>msgbuffer</i> , <i>msglength</i>);			

Parameters	<i>ferrorcode</i>	integer by reference (required) Passes an error code returned by the FCHECK intrinsic indicating which message to return in <i>msgbuffer</i> .
	<i>msgbuffer</i>	logical array (required) Returns the error message identified with <i>ferrorcode</i> . To contain the longest possible message, <i>msgbuffer</i> must be ≥ 72 bytes long.
	<i>msglength</i>	integer by reference (required) Returns the length of the error message in <i>msgbuffer</i> . The length is returned in positive bytes.
Condition Codes	CCE	Request granted.
	CCG	Request denied. No error message exists for this <i>ferrorcode</i> .
	CCL	Request denied. The <i>msgbuffer</i> address was out of bounds, <i>msgbuffer</i> was not large enough, or <i>msglength</i> was out of bounds.
Related Information	Additional Discussion	“Using FERRMSG” in Chapter 4.

FFILEINFO

Returns information about a file. (No intrinsic number assigned)

Syntax

0-V	IV	IV	*
FFILEINFO(<i>filenum</i> [, <i>itemnum</i> , <i>item</i>] [...]);			

NoteUp to five *itemnum/item* pairs can be specified.**Parameters***filenum***integer by value (required)**

The file number of the file that information is requested.

*itemnum***integer by value (optional)**Specifies which *item* value is to be returned. (Refer to Table 2-5.)*item*

type varies

Returns the value of the item specified in the corresponding *itemnum*. (Refer to Table 2-5.)

FFILEINFO

Table 2-5. FFILEINFO Itemnum/Item Values

Itemnum	Item Type	Item Description
1	BA	File designator (28 bytes): Returns the file designator of the file being referenced in the format: <i>filename.groupname.accountname</i> Must be ≥ 28 bytes in length. Unused bytes are filled with right-justified blanks and a names file will return an empty string.
2	L	File options: Returns file access characteristics (refer to the <i>foption</i> bit summary).
3	L	Access options: Returns file access information (refer to the <i>FFaoption</i> figure).
4	I	Record size: Returns the logical record size associated with the file: ■ If the file was created as a binary file, this value is positive and is in words. ■ If the file was created as an ASCII file, this value is negative and is in bytes. For message files, when there is call to FCONTROL with <i>controlcode</i>=46, the value returned is the size of the data records, including the 4-byte header.
5	I	Device type/subtype: <i>integer</i> (<i>optional</i>) A word to which the type and subtype of the device being used for the file is returned, where bits (0:8) indicate device subtype, and bits (8:8) indicated device type. If the file is not spooled, which can be determined from <i>hdaddr</i> (0:8), the returned <i>devtype</i> is actual. The same is true if the file is spooled and was opened via the logical device number. However, if an output file is spooled and was opened by device class name, <i>devtype</i> contains the type and subtype of the first device in its class, which may be different from the device actually used. If you have opened a device in a serial disk class, the type returned in bits (8:8) is 31 (%37) even though the real device type is as specified in Table 4-2, Classification of Devices, in the MPE V System Operation and Resource Management Reference Manual (32033-90005). Device type 7 (%07) is returned by FGETINFO for devices opened in a foreign disk class.

Table 2-5. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
6	L	Logical device number: Returns the logical device number of the device where the file resides. ■ If the file is a disk file, the ldev is the location of the first extent. ■ If the file is spooled, the ldev is a virtual device number that does not correspond to the system configuration I/O device list. ■ If the file is located on a remote computer, linked by a DS (point-to-point or X.25) link, the left eight bits (0:8) are the ldev of the distributed system (DS) device. ■ If the file is located on a remote computer, linked by NS 3000, the left eight bits (0:8) are the remote environment of the connection. The right eight bits (8:8) are the ldev of the device on the remote computer where the file resides. ■ For version G.02.00 or later, if either the DS device is returned, or the ldev is zero, then the entire word is zero. ■ For version G.00.00 or later, if ldev is greater than 255, zero is returned. Note: If a zero is returned, use item 50.
7	L	Hardware device address: A word to which the hardware address of the device is returned, where bits (0:8) are the Device Reference Table (DRT) number, and bits (8:8) are the unit number. The limitations for the Device Reference Table number are discussed in "Special Considerations". if the device is spooled, the DRT number will be zero and the unit number is undefined.
8	I	File code: Returns the file code of a disk file

FFILEINFO**Table 2-5. FFILEINFO Itemnum/Item Values (continued)**

Itemnum	Item Type	Item Description
9	I	Current logical record pointer: A double-word to which a double integer representing the current logical record pointer setting is returned. 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.
10	I	EOF: A double-word to which a double positive integer equal to the number of logical records currently in the file is returned. If the file does not reside on disk, this value will be zero. For Interprocess Communication (IPC) when a call to FCONTROL with a <i>controlcode</i> of 46 is in effect, the number of records returned in <i>EOF</i> will include open, close, and data records.
11	I	File Limit: A double-word to which a double positive integer representing the number of the last logical record that could ever exist in a file, because of the physical limits of the file, is returned. If the file does not reside on disk, this value will be zero.
12	I	Log Count: A double-word to which a double positive integer representing the total number of logical records passed to and from the user during the current access of the file is returned.
13	I	Physical Count: A double-word to which a double positive integer is returned. This value represents the total number of physical input/output operations performed within this process against the file since the last FOPEN call.
14	I	Block size: Returns the file block size: ■ If the file is binary, the value is positive and the size is in words. ■ If the file is ASCII, the value is negative and the size is in bytes.

Table 2-5. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
15	L	Extent size: A word to which the disk extent size associated with the file (in sectors) is returned.
16	I	Maximum Number of Extents: A word to which the maximum number of disk extents allowable for the file is returned.
17	I	User Labels: Returns the number of user labels defined for the file during creation. If the file is not a disk file, this number is zero. When an old file is opened for overwrite output, the value is not reset and the old user label is not destroyed.
18	BA	Creator: Returns the name of the file creator (8-bytes). If the file does not reside on disk, blanks are returned.
19	I	Label Address: A double-word to which the sector address of the file label is returned. The high-order eight bits show the logical device number. The remaining 24 bits show the absolute disk address. If the file is not on disk, zero is returned.
20	I	Blocking factor
21	I	Physical block size; indicates words
22	I	Data block size; indicates words
23	I	Offset to data in blocks; indicates words
24	I	Offset of active record table for RIO files; indicates words
25	I	Size of active record table within the block; indicates words
26	BA	Volume ID (label tape)
27	BA	Volume set ID (label tape)

Table 2-5. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
28	L	Expiration date (Julian)
29	I	File sequence number
30	I	Reel number
31	I	Sequence type
32	I	Creation date (Julian)
33	I	Label type
34	I	Current number of writers
35	I	Current number of readers
36	L	File allocation date, when the file was last restored (CALENDAR format)
37	D	File allocation time, when the file was last restored (CLOCK format)
38	L	Spoolfile devicefile number: Bits (1:15) = Devicefile number Bit (0:1) = 1 Output spoolfile Bit (0:1) = 0 Input spoolfile
40	D	disk or diskette device status
41	I	Device type
42	I	Device subtype
43	BA	Environment file name
44	I	Last disk extent allocated
45	BA	File name from labeled tape HDR1 record
46	I	Tape density * The value returned for item 46 is: ■ 0, when the tape drive is single density ■ The density you specified at FOPEN when the tape is variable density, positioned at the beginning of the tape, and you have any kind of write access. ■ The actual tape density in all other cases. The tape density is measured in bits per inch (BPI). Legal densities are 800, 1600, and 6250.

Table 2-5. FFILEINFO Itemnum/Item Values (continued)

Itemnum	Item Type	Item Description
47	I	DRT number
48	I	Device unit number
49	I	Software interrupt for message files
50	I	Real device number of the file
51	I	Remote environment number Note: If using NS/3000 RFA (remote file access), specify <i>DSDEVICE ldev#</i> when you are using a DS (point-to-point or X.25) link.
52	D	Last modification time (CLOCK format)
53	L	Last modification date (CALENDAR format)
54	L	File creation date (CALENDAR format)
55	L	Last access date (CALENDAR format)
56	I	Number of data blocks in a variable length file
57	I	Number of user labels written to the file
58	I	Number of users having output access (WRITE) for a particular file (# of opens)
59	I	Number of users having input access (READ/UPDATE) for a particular file (# of opens)
60	I	Terminal type: 0 File's associated device not a terminal 1 Standard hardwire or multipoint terminal 2 Terminal connected through phone-modem 3 DS pseudo terminal 4 X.25 Packet Switching Network PAD (packet assembler/disassembler) terminal 5 NS virtual terminal
61	BA	NS 3000 remote environment ID name Note: If using NS 3000 RFA (remote file access), specify <i>DSDEVICE ldev#</i> when using a DS (point-to-point or X.25) link. A buffer must be provided for the node name (or <i>envid</i>) with the required space of 52 bytes; otherwise, data corruption may occur on variables following <i>itemnum=61</i> or an FSERR 73, BOUNDS VIOLATION may be returned.

Table 2-6. FFILEINFO File codes

Integer	Mnemonic	Description
0	Default (unreserved)	
1024	USL	User subprogram library
1025	BASD	Basic data
1026	BASP	Basic program
1027	BASFP	Basic fast program
1028	RL	Relocatable library
1029	PROG	Program file
1031	SL	Segmented library
1035	VFORM	VPLUS forms file
1036	VFAST	VPLUS fast forms file
1037	VREF	VPLUS reformat file
1040	SAV	Cross loader ASCII file (SAVE)
1041	BIN	Cross loader relocated binary file
1042	DSP	Cross loader ASCII file (DISPLAY)
1050	EDITQ	Edit quick file
1051	EDTCQ	Edit KEEPQ file (COBOL)
1052	EDTCT	Edit TEXT file (COBOL)
1054	TDPDT	TDP diary file
1055	TDPQM	TDP proof marked QMARKED
1056	TDPP	TDP proof marked non-COBOL file
1057	TDPCP	TDP proof marked COBOL file
1058	TDPQ	TDP work file
1059	TDPXQ	TDP work file (COBOL)
1060	RJEPN	RJE punch file
1070	QPROC	QUERY procedure file
1080	KSAMK	KSAM key file
1083	GRAPH	GRAPH specification file
1084	SD	Self-describing file

Table 2-6. FFILEINFO File codes (continued)

Integer	Mnemonic	Description
1090	LOG	User logging log file
1100	WDOC	HPWORD document
1101	WDICT	HPWORD hyphenation dictionary
1102	WCONF	HPWORD configuration file
1103	W2601	HPWORD attended printer environment
1110	PCCELL	IFS 3000 character cell file
1111	PFORM	IFS 3000 form file
1112	PENV	IFS 3000S environment file
1113	PCCMP	IFS 3000 compiled character cell file
1114	RASTR	Graphics image in RASTR format
1130	OPTLF	OPT/3000 log file
1131	TEPES	TEPE/3000 script file
1132	TEPEL	TEPE/3000 log file
1133	SAMPL	APS/3000 log file
1139	MPEDL	MPEDCP/DRP log file
1140	TSR	HPToolset root file
1141	TSD	HPToolset data file
1145	DRAW	Drawing file for HPDRAW
1146	FIG	Figure file for HPDRAW
1147	FONT	Reserved
1148	COLOR	Reserved
1149	D48	Reserved
1152	SLATE	Compressed SLATE file
1153	SLATW	Expanded SLATE work file
1156	DSTOR	RAPID/3000 DICTDBU utility store file
1157	TCODE	Code file for Transact compiler
1158	RCODE	Code file for Report/3000 compiler
1159	ICODE	Code file for Inform/3000 compiler
1166	MDIST	HPDesk distribution list
1167	MTEXT	HPDesk text

Table 2-6. FFILEINFO File codes (continued)

Integer	Mnemonic	Description
1168	MARPA	ARPA messages file
1169	MARPD	ARPA distribution list
1170	MCMND	HPDesk abbreviated commands file
1171	MFRTM	HPDesk diary free time list
1172	None	Reserved
1173	MEFT	HPDesk external file transfer messages file
1174	MCRPT	HPDesk encrypted item
1175	MSERL	HPDesk serialized (composite) item
1176	VCSF	Reserved
1177	TTYE	Terminal type file
1178	TVFC	Terminal vertical format control file
1192	NCONF	Network configuration file
1193	NTRAC	Network trace file
1194	NLOG	Network log file
1195	MIDAS	Reserved
1211	ANODE	Reserved
1212	INODE	Reserved
1213	INVRT	Reserved
1214	EXCEP	Reserved
1215	TAXON	Reserved
1216	QUERF	Reserved
1217	DOCDR	Reserved
1226	VC	VC file
1227	DIF	DIF file
1228	LANGD	Language definition file
1229	CHARD	Character set definition file
1230	MGCAT	Formatted application file
1236	BMAP	Base map specification file
1242	BDATA	BASIC data file
1243	BFORM	BASIC field order file for VPLUS

Table 2-6. FFILEINFO File codes (continued)

Integer	Mnemonic	Description
1244	BSAVE	BASIC saved program file
1245	BCNFG	Configuration file for default option BASIC program
1258	PFSTA	Pathflow static file
1259	PFDYN	Pathflow dynamic file
1270	RFDCA	Revisable form DCA data stream
1271	FFDCA	Final form DCA data stream
1272	DIU	Document interchange unit file
1273	PDOC	HPWORD/150 document
1401	CWPTX	Reserved
1421	MAP	HPMAP/3000 map specification file
1422	GAL	Reserved
1425	TTX	Reserved

Table A-2. FOPTIONs for Use with FOPEN

BITS	(0:2)	(2:3)	(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	Reserved	File Type	Disallow :FILE	MPE Tape Labels	Carriage Control	Record Format	Default Designator	ASCII/ Binary	Domain
MEANING		00 0=STD 00 1=KSAM 01 0=RIO 10 0=CIR 11 0= MSG	0=Allow :FILE 1=No :FILE	0=NON LABELED TAPE 1=LABELED TAPE	0=NOCCTL 1= CCTL	00=Fixed 01=Variable 10=Undefined	000=Filename 001=\$STD LST 010=\$NEWPASS 011=\$OLDPASS 100=\$STD IN 101=\$STD INX 110=\$NULL	0=Binary 1=ASCII	00=New File 01=Old Permanent File 10= Old Temporary File 11=Old Perm. or Temp. File

Figure 2-3. Foption Bit Summary

Table A-3. AOPTIONS for Use with FOPEN

BITS	(0:3)	(3:1)	(4:1)	(5:2)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)
FIELD	Reserved	File Copy	No-Wait I/O	Multi Access	Inhibit Buffering	Exclusive Access	Dynamic Locking	Multi-record Access	Access Type
MEANING		0=Access in file's native mode 1=Access as standard sequential file	1=No Wait 2=Non No-Wait	00=Non-multi-access 01=Only Intra-job multi-access 10=Inter-job multi-access allowed	0=BUF 1=NOBUF	00=Default 01=Exclusive 10=Exclusive Access Read 11=Share	0=No FLOCK Allowed 1=FLOCK Allowed	0=No Multi-Record 1=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 2-4. Aoption Bit Summary

Condition Codes

CCE

Request granted.

CCG

Not returned.

CCL

Request denied. Access or calling sequence error.

Related Information

Manuals

MPE File System Reference Manual(30000-90236).

FGETINFO

Returns access and status information about a file that is currently open. (Intrinsic number 11)

Syntax

```

      0-V      IV      BA      L      L
FGETINFO(filenum,formaldesig,foption,aoption,
        I      I      L      L      I
        recsize, devtype,ldevnum,hdaddr,filecode,
        D      D      D      D      D
        recpt, EOF, filelimit, logcount, physcount,
        I      L      I      I
        blksize,extsize, numextents, userlabels,
        BA      D
        creatorid, labaddr);

```

Parameters

filenum

integer by value (required)

The file number of the file for which information is requested.

formaldesig

byte array (optional)

Returns the actual designator of the file being referenced, in the following format:

filename.groupname.accountname

The *formaldesig* array must be at least 28 bytes in length. When the actual designator is returned, unused bytes in the array are filled with blanks on the right. A nameless file returns an empty string.

foption

logical by reference (optional)

Returns seven file access characteristics by setting bit groupings (refer to Figure 2-5) as follows:

Bits	Value/Meaning
14:2	Domain:
	The file domain searched to locate the file:
00	New file
01	Old permanent file
10	Old temporary file
11	Old temporary or permanent file
13:1	ASCII/Binary:
0	Binary
1	ASCII

Bits	Value/Meaning
10:3	Default file designator: 000 Actual file designator (same as formal file designator) 001 Actual file designator \$STDLIST 010 Actual file designator \$NEWPASS 011 Actual file designator \$OLDPASS 100 Actual file designator \$STDIN 101 Actual file designator \$STDINX 110 Actual file designator \$NULL
8:2	Record format <i>foption</i> : 00 Fixed-length records 01 Variable-length records 10 Undefined-length records 11 Spoolfile format records The format the records in the file are recorded:
7:1	Carriage control: 0 No carriage-control character expected 1 Carriage-control character expected
6:1	MPE tape label: 0 Nonlabeled tape 1 Labeled tape
5:1	Disallow FILE equation <i>foption</i> : This option ignores any corresponding FILE command, so the specifications in the call to FOPEN take effect (unless overridden by those in the file label). The following bit settings apply: 1 Disallow FILE command 0 Allow FILE command



Bits	Value/Meaning
2:3	File type:
000	Standard file (STD)
001	Keyed sequential access method file (KSAM)
010	Relative I/O file (RIO)
100	Circular file (CIR)
110	Message file (MSG)
0:2	Reserved for MPE. Should be set to zero.

aoption

logical by reference (optional)

Returns file access information by setting bit groupings (refer to Figure 2-6), in the following manner:

Bits	Value/Meaning
12:4	Access type <i>aoption</i> :
	The type of access allowed users of this file:
0000	READ only
0001	WRITE only
0010	WRITE only (previous data in file is not deleted)
0011	APPEND only
0100	READ/WRITE
0101	UPDATE
0110	EXECUTE
11:1	Multirecord:
0	Nonmultirecord mode
1	Multirecord mode
10:1	Dynamic locking:
0	Disallow dynamic locking/unlocking
1	Allow dynamic locking/unlocking

FGETINFO

Bits	Value/Meaning
8:2	Exclusive: Specifies the access, from the time it is opened to the time it is closed: 00 Default 01 EXCLUSIVE access 10 SEMI-EXCLUSIVE access 11 SHARED access
7:1	Inhibit buffering: Inhibits automatic buffering by MPE and allows I/O to take place directly between the stack and the applicable hardware device: 0 Normal buffering (BUF) 1 Inhibit buffering (NOBUF)
5:2	Multiaccess mode: Provides the user with shared access to the file: 00 Nonmultiaccess 01 Intrajob multiaccess (MULTI) 10 Interjob multiaccess (GMULTI)
4:1	NOWAIT I/O: Allows the user to initiate an I/O request and to have control returned before completion of the I/O. 0 NOWAIT I/O not in effect 1 NOWAIT I/O in effect
3:1	File copy access: 0 Access in file's type 1 Access as standard sequential file
0:3	Reserved for MPE

recsize

integer by reference (optional)

Returns the logical record size associated with the file:

- If the file was created as a binary file, this value is positive and expresses the size in words.
- If the file was created as an ASCII file, this value is negative and expresses the size in bytes.

For interprocess communication (IPC), when there is a call to **FCONTROL** with *controlcode*=46, the value returned in *lrecsize* is the size of the data records, including the 4-byte header.

devtype

integer by reference (optional)

A word to which the type and subtype of the device being used for the file is returned, where bits (0:8) indicate device subtype, and bits (8:8) indicate device type. If the file is not spooled, which can be determined from *hdaddr* (0:8), the returned *devtype* is actual. The same is true if the file is spooled and was opened via the 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. If you have opened a device in a serial disk class, the type returned in bits (8:8) is 31 (%37) even though the real device type is as specified in Table 4-2, Classification of Devices, in the MPE V System Operation and Resource Management Reference Manual (32033-90005). Device type 7 (%07) is returned by **FGETINFO** for devices opened in a foreign disk class.

ldevnum

logical by reference (optional)

A word to which the logical device number (*ldev*) associated with the device on which the file resides is returned.

If the file is a disk file, the logical device number will be that of the first extent. If the file is spooled, *ldevnum* 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, linked by a DS (point-to-point or X.25) link, the left eight bits (0:8) are the logical device number of the Distributed System (DS) device. If the remote computer is linked by NS/3000, the left eight bits are the remote environment of the connection. The right eight bits (8:8) are the logical device number of the file on the remote computer. For version G.00.00 or later, note that if *ldev* is greater than 255, zero is returned. For version G.02.00 or later, if either the DS device for the RFA or the *ldev* is zero, then the entire word is zero.

hdaddr

integer by reference (optional)

Returns the hardware address of the device, where bits (0:8) are the Device Reference Table (DRT) number, and bits (8:8) are the unit number. The

FGETINFO

	limitations for the Device Reference Table number are discussed in “Special Considerations”. If the device is spooled, the DRT number will be zero and the unit number is undefined.
<i>filecode</i>	integer by reference (optional) Returns the file code of a disk file.
<i>lrecptr</i>	double by reference (optional) Returns the current logical record pointer setting. This value is the displacement in logical records from record number 0 in the file and identifies the record that would be accessed next by an FREAD or FWRITE call.
<i>eof</i>	double by reference (optional) Returns a value indicating the last logical record in the file (equivalent to the number of logical records currently in the file). If the file does not reside on disk, this value is zero. For interprocess communication (IPC), when a call to FCONTROL with <i>itemnum</i> =46 is in effect, the number of records returned in <i>eof</i> includes open, close, and data records.
<i>filelimit</i>	double by reference (optional) Returns a number representing the last logical record that could exist in the file (the physical limits of the file). If the file does not reside on disk, this value is zero.
<i>logcount</i>	double by reference (optional) Returns the total number of logical records passed to and from the program during the current file access.
<i>physcount</i>	double by reference (optional) A double-word to which a double positive integer is returned. This value represents the total number of physical input/output operations performed within this process against the file since the last FOPEN call.
<i>blksize</i>	integer by reference (optional) Returns the block size of the referenced file. If the file is a binary file, this value is positive and expresses the size in words. If the file is ASCII, this value is negative and shows the size in bytes.
<i>extsize</i>	logical by reference (optional) A word to which the disk extent size associated with the file (in sectors) is returned.

<i>numextent</i>	integer by reference (optional) Returns the maximum number of disk extents allowable for the file.
<i>userlabels</i>	integer by reference (optional) Returns the number of user labels defined for the file during creation. If the file is not a disk file, this number is zero. When an old file is opened for overwrite output, the value of <i>userlabels</i> is not reset, and old user labels are not destroyed.
<i>creatorid</i>	byte array (optional) Returns the name of the file creator (8-byte array). If the file is not a disk file, blanks are returned.
<i>labaddr</i>	double by reference (optional) Returns the sector address of the file label. The high-order eight bits show the logical device number. The remaining 24 bits show the absolute disk address. If the file is not on disk, zero is returned.

Table A-2. FOPTIONs for Use with FOPEN

BITS	(0:2)	(2:3)	(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	Reserved	File Type	Disallow :FILE	MPE Tape Labels	Carriage Control	Record Format	Default Designator	ASCII/ Binary	Domain
MEANING		00 0=STD 00 1=KSAM 01 0=RIO 10 0=CIR 11 0= MSG	0=Allow :FILE 1=No :FILE	0=NON LABELED TAPE 1=LABELED TAPE	0=NOCCTL 1= CCTL	00=Fixed 01=Variable 10=Undefined	000=Filename 001=\$STD LST 010=\$NEWPASS 011=\$OLDPASS 100=\$STDIN 101=\$STDINX 110=\$NULL	0=Binary 1=ASCII	00=New File 01=Old Permanent File 10= Old Temporary File 11=Old Perm. or Temp. File

Figure 2-5. Foption Bit Summary

Table A-3. AOPTIONs for Use with FOPEN

BITS	(0:3)	(3:1)	(4:1)	(5:2)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)
FIELD	Reserved	File Copy	No-Wait I/O	Multi Access	Inhibit Buffering	Exclusive Access	Dynamic Locking	Multi- record Access	Access Type
MEANING		0=Access in file's native mode 1=Access as standard sequential file	1=No Wait 2=Non No- Wait	00=Non-multi- access 01=Only Intra- job multi- access 10=inter-job multi- access allowed	0=BUF 1=NOBUF	00=Default 01=Exclusive 10=Exclusive Access Read 11=Share	0=No FLOCK Allowed 1=FLOCK Allowed	0=No Multi- Record 1=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 2-6. Aoption Bit Summary

Operation Notes Returns access and status information about a file located on any device. The file must be opened by the calling process at the time of the FGETINFO call.

Condition Codes	CCE	Request granted.
	CCG	Not returned.
	CCL	Request denied. An error occurred.

Related Information	Manuals	<i>MPE File System Reference Manual(30000-90236).</i>
----------------------------	---------	---

FINDJCW

Searches the job control word table for a specified job control word (JCW) and returns its value. (Intrinsic number 86)

Syntax

BA	L	I
FINDJCW(<i>jcwname</i> , <i>jcwvalue</i> , <i>jcwstatus</i>) ;		

Parameters

jcwname

byte array (required)

Passes the name of the job control word (JCW) to be found. This name can contain up to 255 alphanumeric characters, starting with a letter and delimited by a nonalphanumeric character, such as a blank.

jcwvalue

integer by reference (required)

Returns the value of *jcwname*, if *jcwname* is found. If *jcwname* is not found, *jcwvalue* is unchanged.

jcwstatus

integer by reference (required)

Returns the execution status of the intrinsic, as follows:

Value	Meaning
0	Successful
1	<i>Jcwname</i> > 255 characters
2	<i>Jcwname</i> does not start with an alpha character
3	<i>Jcwname</i> not found

Operation Notes

There are three types of JCWs in the system:

- User-defined
- System-defined
- System-reserved

FINDJCW can return the value of any type of JCW. The system-reserved JCWs are:

- HPDATE
- HPDAY
- HPHOUR
- HPMINUTE
- HPMONTH
- HPYEAR

Related Information

Additional Discussion	“Interprocess Communications” in Chapter 5.
Intrinsics	GETJCW, PUTJCW, SETJCW
Commands	SETJCW, SHOWJCW

FINDPMAPADDR

Searches the FPMAP in a program or SL file for a specific address in a particular segment. An external FPMAP record corresponding to the nearest entry point at or preceding *address* is returned. The caller must have MULTI-RECORD READ access to open the file. (No intrinsic number assigned)

Syntax

	IV	IV	IV	V
FINDPMAPADDR	(<i>progfnum</i> ,	<i>segnum</i> ,	<i>address</i> ,	<i>xpmaprec</i> ,
	IV	I		
	<i>xpmapreclen</i> ,	<i>status</i>)	;	

Parameters

<i>progfnum</i>	integer by value	Passes the number of the program or SL file of the FPMAP to be searched, as returned from the FOPEN intrinsic. Caller is responsible for opening the file with MULTI-RECORD READ access.
<i>segnum</i>	integer by value	Passes the number of the logical segment in which the address is to be found.
<i>address</i>	integer by value	Passes the PB-relative address in segment <i>segnum</i> with which the external FPMAP record returned is associated.
<i>xpmaprec</i>	value	Returns the external FPMAP record corresponding to the first entry point at or preceding <i>address</i> . If the entry point located has more than one name, the first one encountered in the FPMAP will be returned.
<i>xpmapreclen</i>	integer by value	Returns the number of words of the external FPMAP record returned in <i>xpmaprec</i> . If the call requests more words than are currently defined for an external FPMAP record, zeros are returned in the extra words.

status***integer***

Returns a value specifying whether the call was successful

Value:

0	Successful
2	Address out of range
3	Logical segment <i>segnum</i> did not exist
10	File <i>progfnun</i> did not contain FPMAPdata
11	File <i>progfnun</i> was not a program of the SL file.
12	A file system I/O error occurred on file <i>progfnun</i>

Related Information

Manual

MPE Segmenter Reference Manual (30000-90011).

FINDPMAPNAME

Searches the FPMAP in a program or SL file for a segment name or an entry point name. (No intrinsic number assigned)

Syntax

IVBABAIA

FINDPMAPNAME(*prognum*,*segname*,*entname*,*xpmaprec*,

VI

xmapreclen,*status*);

Functional Return

This intrinsic returns the external FPMAP record specified. If unable to find the segment or entry point specified, the intrinsic returns a status code that indicates why the search failed.

Parameters

<i>prognum</i>	integer by value The number of the program or SL file whose FPMAP is searched, as returned from the FOPEN intrinsic. The caller must have MULTI-RECORD and READ access to open the file.
<i>segname</i>	byte array Optional. The name of the segment to be found. Terminate with a blank. If the first character of <i>segname</i> is blank, <i>segname</i> is omitted from the calling sequence. A maximum of 15 characters can be specified for the search. All characters will be upshifted.
<i>entname</i>	byte array Optional. The name of the procedure or secondary entry point to be found. Terminate with a blank. If the first character of <i>entname</i> is blank, <i>entname</i> is omitted from the calling sequence. A maximum of 15 characters can be specified for the search. All characters will be upshifted.
<i>xpmaprec</i>	integer array The external FPMAP record corresponding to the entry point passed in <i>entname</i> . The format of the record is defined in Figure 3-1. The record returned may be a segment, a procedure, or a secondary entry point (depending upon <i>segname</i> and <i>entname</i> specified).
<i>xmapreclen</i>	value

The number of words of the external FPMAP record returned in *xpmaprec*. If the call requests more words than are currently defined for an external FPMAP record, zeros are returned in the extra words.

status

integer

Returns a value specifying whether the call was successful:

Value

- | | |
|----|---|
| 0 | Successful. |
| 1 | Entry point name not located. |
| 2 | Address to be located outside the bounds of segment specified. |
| 3 | Program/SL file did not contain segment specified, or, the SL segment requested was marked as deleted. |
| 4 | External PMAP file was too small. |
| 9 | Option variable parameter list was illegal. |
| 10 | Program/SL file did not contain a PMAP. |
| 11 | Program/SL file code was not that of a program/SL file, or, SL loader ID was not compatible with this version of the PMAP intrinsics. |
| 12 | File system error on the program/SL file. |
| 13 | File system error on the external PMAP file. |
| 14 | Internal PMAP file improperly FOPENed. |
| 15 | Bad internal PMAP. |

Related Information

Manual

MPE Segmenter Reference Manual (30000-90011).

FINTEXTIT

Causes the return from your interrupt procedure. (Intrinsic number 23)

Syntax

O-V	LV
<code>FINTEXTIT(<i>interruptstate</i>);</code>	

Parameters *interruptstate* integer by value (optional)

A value indicating the state in which software interrupts should be left. Bit (15:1) controls this as:

Value	Meaning
1	Enable software interrupts (Default)
0	Disable software interrupts

Operation Notes

Software interrupts are set on return, according to *interruptstate*. If *interruptstate* is omitted, the software interrupts are enabled by default.

Related Information

Manuals *MPE File System Reference Manual*(30000-90236).

FINTSTATE

Enables/disables all software interrupts against the calling process (available only on version G.01.00 or later). (Intrinsic number 24)

Syntax

L	LV
<i>oldstate</i> :=FINTSTATE(<i>interruptstate</i>);	

Functional Return

oldstate

integer (assigned functional return)

Returns the previous state (enabled or disabled) of the software interrupts.

Parameters

interrupt-state

integer by value (required)

A value enabling/disabling software interrupts through bit (15:1):

Value	Meaning
0	Disable software interrupts
1	Enable software interrupts

Operation Notes

The software interrupt facility enables FREAD/FWRITE completion processing with an interrupt procedure. A call to FREAD or FWRITE is necessary to initiate the I/O request. Both of these intrinsics return to the process as soon as the request has been started. When the operation completes, the program is interrupted and control goes to the chosen interrupt procedure. This performs whatever processing is necessary and then resumes the original program.

Software interrupts are armed for a particular file by specifying the interrupt procedure's *plabel* in an FCONTROL call with *itemnum*=48. If *aoption* (4:1) is set to 0, selecting *itemnum*=48 resets it to 1. Use IOWAIT or IODONTWAIT if *itemnum*=48 is used.

FINTSTATE

Software interrupts are inhibited just before entering an interrupt procedure. This is done to stop unwanted nesting of the interrupt procedures. Each interrupt procedure should call **FINTEXIT** to reenable other interrupts just before it exits.

Software interrupts are automatically inhibited before a **CTRL Y** trap procedure. The trap procedure can allow software interrupts only by calling the **FINTSTATE** intrinsic. The **RESETCONTROL** intrinsic restores the interrupt state of the process to its pre-**CTRL Y** value (unless the trap procedure issues an **FINTSTATE** call; **RESETCONTROL** makes no change).

The first parameter of the software interrupt procedure is the file number of the file that caused the interrupt, at location **Q-4**.

It is necessary to issue a call to the **IODONTWAIT** intrinsic against the file in order to complete the request. When reading, the *buffer* parameter is ignored in the **FREAD** call. The data is moved to the array specified by the *buffer* parameter of **IODONTWAIT**.

An incomplete **FREAD/FWRITE** request can be aborted by issuing an **FCONTROL** call with an *itemnum* of 43 (abort **NOWAIT** I/O).

Limitations:

- Only message (MSG) files allow software interrupts.
- No more than one incomplete **FREAD/FWRITE** can be outstanding for a particular file.
- The interrupt is held off while executing within MPE , with the following exceptions:
 - **PAUSE** allows the interrupt.
 - **IOWAIT** allows the interrupt if the *filenum* parameter is set to 0.In these instances, the intrinsic is reinvoked after the interrupt procedure executes.
- Do not use **FINTSTATE** with remote files.

Related Information

Manuals *MPE File System Reference Manual*(30000-90236).

FLABELINFO

Returns information from the file label of a disk file. (Intrinsic number 25)

Syntax

```

      BA      IV      I
FLABELINFO(formaldesig,mode,fserrorcode,
      IA      BA      IA
      itemnum,item,itemerror);

```

Parameters *formaldesig* byte array (required)

The name of the file, terminated by a nonalphanumeric character other than a period (.) or a slash (/). The file name can include password, group, and account specifications. The file must be in the permanent file directory. The file name can backreference a file equation and optionally be preceded by an asterisk.

mode integer by value (required)

An option specifying the valid backreferencing to file equations for the file. Valid values are:

Value	Meaning
0	Use file equation (if one exists)
1	Must use file equation (error if one does not exist)
2	Ignore existing file equations

fserrorcode integer by reference (required)

Returns a value indicating whether an error or warning occurred when FLABELINFO attempted to return requested information:

- A value of zero indicates that no errors were encountered.
- A positive value is a file system error code and indicates that an error was encountered and no information was returned in *item*.
- A -1 indicates that an item error or warning has occurred. Check the *itemerror* parameter to determine which item(s) has an error/warning and what it is.

FLABELINFO

<i>itemnum</i>	integer array (required) Specifies which <i>item</i> value is to be returned. (Refer to Table 2-7.) To indicate the end of the list, place a zero in the element following the last <i>itemnum</i> .
<i>item</i>	byte array (required) The value of the item specified in the corresponding <i>itemnum</i> . (Refer to Table 2-7.) <i>itemnum/items</i> are paired such that the <i>n</i> th field of the <i>item</i> record corresponds to the <i>n</i> th element of the <i>itemnum</i> array.
<i>itemerror</i>	integer array (required) An error number corresponding to the items specified in the <i>itemnum</i> array. The <i>itemnum/item</i> and <i>itemerrors</i> parameters are paired such that the <i>n</i> th element of the <i>itemerror</i> array corresponds to the <i>n</i> th element of the <i>itemnum</i> array. If a value in the <i>itemerror</i> array is negative, a warning exists for the corresponding item. If the value is positive, an error was detected for the corresponding item. The absolute value of each value is a file system error number.

Table 2-7. FLABELINFO Itemnum/Item Values

Itemnum	Mnemonic	Item Description
1	BA	File name. (8 byte)
2	BA	Group name. (8 byte)
3	BA	Account name (8 byte)
4	BA	File creator name. Either creator (C),privileged (PM) or manager (AM if file is within account, otherwise SM) access required. (8 bytes)
5	L	Security matrix for access: Either creator (C), privileged (PM) or manager (AM if file is within account, otherwise SM) access required.
6	L	File creation date. The date in CALENDAR intrinsic format. Either creator (C) or manager (AM if file is within account, otherwise SM) access required.
7	L	Last access date. The date in CALENDAR intrinsic format. May not be up-to-date when the file is open.
8	L	Last modification date. The date in CALENDAR intrinsic format. May not be up-to-date when the file is open.
9	I	File code of disk file
10	I	Number of user labels written.May not be up-to-date when the file is open.
11	I	Number of user labels available. May not be up-to-date when the file is open.
12	D	Total number of logical records possible in the file.
13	L	File options. Refer to the <i>foption</i> bit summary.
14	I	Record size, in bytes (a negative value is returned to indicate bytes).
15	I	Block size in words.
16	I	Maximum number of extents minus one.

Table 2-7. FLABELINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
17	I	Last extent size. Indicates sectors. May not be up-to-date when the file is open.
18	I	Extent size. Indicates sectors.
19	D	Number of logical records in file. Equivalent to EOF. May not be up-to-date when the file is open.
20	D	File allocation time: The time when file was last restored in CLOCK intrinsic format.
21	L	File allocation date: The date when the file was last restored in CALENDAR intrinsic format.
22	I	Number of open/close records: MSG files only. May not be up-to-date when the file is open.
23	BA	Device name from file label (8 bytes)
24	L	Last modification time: CALENDAR intrinsic format. May not be up-to-date when the file is open.
25	LA	First user label (user label 0) (25 bytes) May not be up-to-date when the file is open. Manager (AM if file is within account, otherwise SM) or READ/WRITE (R/W) access required.
26-32		Defined but illegal. Used only on MPE-XL-based systems.
33	BA	File lockword (8 bytes) Returned if you are the file creator, account manager, or system manager.
34		Defined but illegal. Used only on MPE XL-based systems. 0 File is placed on the specified volume at creation 1 File can be placed on any volume containing the specified class at creation 2 File can be placed on any volume within the specified volume set at creation (Default)

Table A-2. FOPTIONs for Use with FOPEN

BITS	(0:2)	(2:3)	(5:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	Reserved	File Type	Disallow :FILE	MPE Tape Labels	Carriage Control	Record Format	Default Designator	ASCII/ Binary	Domain
MEANING		00	0=STD	0=Allow :FILE	0=NON LABELED TAPE	0=NOCCTL	00=Filename	0=Binary	00=New File
		00	1=KSAM	1=No :FILE	1= CCTL	01=Variable	001=\$STD LST	1=ASCII	01=Old Permanent File
		01	0=RIO			10=Undefined	010=\$NEWPASS		
		10	0=CIR				011=\$OLDPASS		10= Old Temporary File
		11	0= MSG				100=\$STDIN 101=\$STDINX 110=\$NULL		11=Old Perm. or Temp. File

Figure 2-7. Foption Bit Summary

Condition Codes	CCE	Request granted.
	CCG	Not returned.
	CCL	Request denied. An error occurred. Refer to the <i>fserrorcode</i> and <i>itemerror</i> parameters for more information.

FLOCK

Dynamically locks a file. If dynamically locking more than one file or combination of file and global RIN, multiple RIN (MR) capability is required. (Intrinsic number 15)

Syntax

IV	LV
FLOCK(<i>filenum</i>,<i>lockflag</i>);	

Parameters

filenum

integer by value (required)

The file number of the file whose file resource identification number (RIN) is to be locked.

lockflag

logical by value (required)

Specify either conditional or unconditional locking by setting bit (15:1) as follows:

Value	Meaning
-------	---------

0

Locking takes place only if the file's global RIN is not currently locked. If the RIN is locked, control returns immediately to the calling process, with condition code CCG.

1

Locking takes place unconditionally. If the file cannot be locked immediately, the calling process suspends until the file can be locked.

Condition Codes

The following condition codes are possible when ***lockflag*** bit (15:1)=1:

CCE

Request granted.

CCG

Not returned.

CCL

Request denied. This file was not opened with the dynamic locking ***aoption*** bit (10:1) specified in the **FOPEN** intrinsic, or there was an attempt to lock more than one file, or combination of file and RIN, and the calling process does not have Multiple RIN (MR) capability.

The following condition codes are possible when ***lockflag*** bit (15:1)=0:

CCE

Request granted.

CCG

Request denied. The file was locked by another process.

CCL

Request denied. This file was not opened with the dynamic locking ***aoption*** bit (10:1) specified in the

FOPEN intrinsic, or there was an attempt to lock more than one RIN and the calling process does not have multiple RIN (MR) capability.

Related Information

Additional
Discussion

The FOPEN intrinsic in this Chapter.

Manuals

MPE File System Reference Manual(30000-90236).

Special
Considerations

Split-stack calls permitted.

Standard capability if only one file is to be locked dynamically. If more than one file is to be locked dynamically, or combination of file and global RINs, the Multiple RIN (MR) capability is required.

FLUSHLOG

Flushes the contents of the user logging memory buffer to the user logging file. User logging (LG) or system supervisor (OP) capability is required. (No intrinsic number is assigned)

Syntax

D I
FLUSHLOG(<i>index</i> , <i>logstatus</i>);

Parameters

index

double by reference (required)

Passes the access to the user logging system. This is the value returned in the *index* parameter of the OPENLOG intrinsic.

logstatus

integer by reference (required)

Returns one of the following values, indicating the success/failure of the intrinsic call:



Value	Meaning
0	Successful
1	User requested NOWAIT 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 is not running.
4	Incorrect <i>index</i> parameter passed to a logging intrinsic
5	Incorrect <i>mode</i> parameter passed to a logging intrinsic
6	User request denied because logging process is suspended.
7	Illegal capability; user logging (LG) or system supervisor (OP) capability required
8	Incorrect password passed to a logging intrinsic.
9	Error occurred while writing
10	Invalid DST passed to a logging system intrinsic.
12	System is out of disk space, user logging cannot proceed
13	No more logging entries.
14	Invalid access
15	End-of-file encountered
16	Invalid logging identifier.

Operation Notes

The FLUSHLOG intrinsic writes the contents of the user logging memory buffer to the disk destination file. This helps to preserve the contents of the memory buffer in the event of a system failure. This intrinsic does not write special records.

Related Information

Additional Discussion	“User Logging” in Chapter 3.
Intrinsics	None
Commands	None
Manuals	None
Special Considerations	User Logging (LG) or System Supervisor (OP) capability required.

FMTCALENDAR

Passes any calendar date, in the same format as the **CALENDAR** intrinsic, and returns it in the following format:

FRI, JAN 27, 1989

(No intrinsic number assigned)

Syntax

LV BA
FMTCALENDAR(<i>date</i> , <i>formatdate</i>);

Parameters

date

logical by value (required)

The calendar date, in the same format as the **CALENDAR** intrinsic:

Bits	Value/Meaning
7:9	Day of year
0:7	Year of century

formatdate

byte array (required)

Returns the formatted calendar date in a 17-character array. If the day of the month is less than 10, a blank precedes it, for example:

FRI, JAN 6, 1989

Related Information

Additional
Discussion

“Formatting Calendar Data and Time Information”
in Chapter 5.

FMTCLOCK

Passes the time of day, in the same format as the CLOCK intrinsic, and returns it in the following format:

12:39 AM

(No intrinsic number assigned)

Syntax

DVBAFMTCLOCK(*time*,*formattime*);

Parameters

<i>time</i>	double by value (required)	
	The time of day, in the same format as the CLOCK intrinsic:	
	Bits	Value/Meaning
Word 1:	8:8	Tenths of seconds
	0:8	Seconds
Word 2:	8:8	Minute of hour
	0:8	Hour of day
<i>formattime</i>	byte array (required)	
	Returns the formatted time of day in an 8-character array. If the hour is a single digit (prior to 10:00) the array contains a leading blank as follows:	
	7:39 AM	

Related Information

Additional Discussion	“Formatting Calendar Date and Time Information” in Chapter 5.
-----------------------	---

FMTDATE

Passes in the calendar date and time of day, in the same format as the **CALENDAR** and **CLOCK** intrinsics, and returns it in the following format:

FRI, JAN 27, 1989, 12:39 AM

(No intrinsic number assigned)

Syntax

LV DV BA

FMTDATE(*date,time,dateandtime*) ;

Parameters	<i>date</i>	logical by value (required) The date, in the same format as the CALENDAR intrinsic: <table><tr><td>Bits</td><td>Value/Meaning</td></tr><tr><td>7:9</td><td>Day of year</td></tr><tr><td>0:7</td><td>Year of century</td></tr></table>	Bits	Value/Meaning	7:9	Day of year	0:7	Year of century				
	Bits	Value/Meaning										
	7:9	Day of year										
	0:7	Year of century										
	<i>time</i>	double by value (required) The time, in the same format as the CLOCK intrinsic: <table><tr><td>Bits</td><td>Value/Meaning</td></tr><tr><td>Word 1: 8:8</td><td>Tenths of seconds</td></tr><tr><td>0:8</td><td>Seconds</td></tr><tr><td>Word 2: 8:8</td><td>Minute of hour</td></tr><tr><td>0:8</td><td>Hour of day</td></tr></table>	Bits	Value/Meaning	Word 1: 8:8	Tenths of seconds	0:8	Seconds	Word 2: 8:8	Minute of hour	0:8	Hour of day
	Bits	Value/Meaning										
	Word 1: 8:8	Tenths of seconds										
	0:8	Seconds										
	Word 2: 8:8	Minute of hour										
	0:8	Hour of day										
<i>dateandtime</i>	byte array (required) Returns the formatted date and time in a 27-character array. If the day of the month is less than 10 or the time is one digit (before 10:00), a blank precedes it. For example, FRI, JAN 6, 1989, 7:39 AM											
Information	Additional Discussion	“Formatting Calendar Date and Time Information” in Chapter 5.										

FOPEN

Establishes access to a file and defines the physical characteristics of the file prior to access.

Syntax

```

      I      0-V      BA      LV      LV      IV
filenum:=FOPEN(formaldesig,foption,aoption,recsize,
      BA      BA      IV      IV      IV
device,formmsg,userlabels,blockfactor,numbuffers,
      DV      IV      IV      IV
filesize,numextents,initialloc,filecode);

```

Functional Return	<i>filenum</i>	integer (assigned functional return) Returns a unique file number identifying the opened file.
Parameters	<i>formaldesig</i>	byte array (optional) A formal file designator, following MPE file naming conventions. The file name must begin with a letter and contain alphanumeric characters, slashes, or periods. Terminate the string by placing a delimiter in the array element following the last valid character. The delimiter can be any nonalphanumeric character except a slash (/), period (.), colon (:), or exclamation point (!). If the file name is the name of a user-defined file, it can begin with an asterisk (*). If the file name is the name of a system-defined file, it can begin with a dollar sign (\$). When creating a KSAM file, you must specify a unique file name (one not currently existing in the permanent file directory). You can specify the remote location of a device as <i>filename:envid</i> . The file, lockword, group, and account names are each limited to eight characters in length. Default: A nameless file is assigned that can be read or written to, but not saved. (The domain option of a nameless file must specify a new file unless it is a devicefile.)

FOPEN

foption

logical by value (optional)

Specifies up to eight different file characteristics by setting corresponding bit groupings. Refer to Table 2-8 for their values.

Note



For old files, default conditions are specified in the file label.

Device characteristics may override some *foption*.

Table 2-8. Foption Parameter Descriptions

Bits	Value/Meaning
14:2	Domain option Indicates which file domain is searched to locate a file. A nameless disk file must always be a new file. The following bit settings are valid: 00 The file is new. No search is necessary. 01 The file is a permanent file. The system file domain (permanent file directory) is searched. 10 The file is a temporary file. The job file domain (temporary file directory) is searched. 11 The file is an old (permanent or temporary) file. The job file domain (temporary file directory) is searched. If not found, the system file domain is searched. Default: 00
13:1	ASCII/binary option Indicates which code, ASCII or binary, a new file is in when written to a device that supports both codes. For disk files, this also affects padding (zeros for a binary file, blank characters for an ASCII file) that can occur when you issue a direct-write intrinsic call (FWRITEDIR) to a record that lies beyond the current logical end-of-file indicator. By default, magnetic tape and files are treated as ASCII files. This option is applicable only at file creation. Foreign disks are treated as binary. The following bit settings are valid: 0 Binary file 1 ASCII file Default: 0
10:3	Designator option A value indicating a special file opening. Any of the following special files can be specified with the <i>formaldesig</i> parameter. For example, a file name of \$STDLIST opens standard list. The following bit settings are valid: 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. Default: 000

Table 2-8. Foption Parameter Descriptions (continued)

Bits	Value/Meaning
	For example, if you pass MYFILE in the <i>formaldesig</i> parameter, then using the designator option to equate it with \$STDIN is equivalent to allowing the file equation FILE MYFILE=\$STDIN. The designator option is not equated with the <i>formaldesig</i> parameter if both of the following conditions are true: ■ The disallow file equation option <i>foption</i> bit (5:1) allows file equations for this file opening. ■ An explicit or implicit FILE command equating the formal file designator to a different actual file designator occurs in the job/session. A leading * in a formal file designator passed by the <i>formaldesig</i> parameter overrides the disallow file equation option setting.
8:2	Record format option Bit settings indicate internal record structure for a file. This option is applicable only at file creation. The following bit settings are valid: 00 Fixed-length records. The file contains logical records of uniform length. Foreign disks always have fixed-length records. Fixed-length records are supported by disk and magnetic tape devices only. 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 block size is determined by multiplying the record size (specified or default) plus one by the blocking factor, and adding one word for the end-of-block indicator. For new files, this option is not allowed when NOBUF is specified. The record format would be changed by these bit settings internally to undefined-length records. If NOBUF is specified, then reads/writes are performed on the entire block, not just the record. Variable-length records are supported by disk and magnetic tape devices only. 10 Undefined-length records. The file contains records of varying length that were not written as variable-length files. By default, all files not on disk or magnetic tape are treated as containing undefined-length records. The file system makes no assumption about the amount of data that is useful. You must determine how much data is required. For undefined-length records, only the data supplied is written, with no information about its length. Undefined-length records are supported by all devices. Default: 00

Table 2-8. Foption Parameter Descriptions (continued)

Bits	Value/Meaning
7:1	Carriage control option Indicates whether or not a carriage control directive is supplied in the calling sequence of each FWRITE call that writes records onto the file. This option is applicable only at file creation. The following bit settings are valid: 0 No carriage-control directive is expected. 1 Carriage-control directives are expected. Default: 0 Carriage control is supplied only for ASCII files. This option and the ASCII/binary option (<i>foption</i> bit (13:1)) are mutually exclusive, and attempts to open new files with both binary and carriage control directives result in an access violation. A carriage-control character passed through the <i>controlcode</i> parameter of FWRITE is recognized as such and acted upon for files that have carriage-control specified in FOPEN. Embedded control characters are treated as data on files for which you specify no carriage-control, and they do not invoke spacing for such files. You can specify spacing action on files for which you have specified carriage-control either by embedding the control in the record, indicated with a <i>controlcode</i> parameter of 1 in the call to FWRITE, or by sending the control code directly through the <i>controlcode</i> parameter of FWRITE. If a carriage-control character is sent to a file where the control cannot be executed directly (for example, line-spacing characters sent to a disk or tape file), the control character is embedded as the first byte of the record. If a carriage-control character is sent to other types of files, the control is transmitted to the driver. 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 control codes %400 through %403 should contain only control information. A record written with control codes %400 through %403 and no data (count=0, or embedded control and count=1) does not cause physical I/O of any sort. To compute record size, the file system assumes carriage-control information to be part of the data record. Therefore, specifying the carriage-control option adds one byte to the record size when the file is originally created. For example, a specification of REC=-132,1,F,ASCII;CCTL results in a <i>recsize</i> of 133 bytes. In general, the entire record can be read (the size of which is returned in <i>itemnum</i>=6 of the FFILEINFO intrinsic or the <i>recsize</i> parameter of FGETINFO). However, on writes to files where carriage-control characters are specified, the data transferred is limited to <i>recsize</i>-1 unless a control of 1 is passed, indicating that the data record is prefixed with embedded carriage-control characters.
6:1	Labeled tape option 0 No labeled tapes 1 Labeled tapes Default: 0

Table 2-8. Foption Parameter Descriptions (continued)

Bits	Value/Meaning
5:1	Disallow file equation option Indicates whether or not to allow file equations. A leading * in a formal file designator can override the disallow file equation option. The following bit settings are valid: 0 Allow FILE equations to override programmatic or system-defined file specifications. 1 Disallow FILE equations from overriding programmatic or system-defined file specifications. Default: 0
2:3	File type option Indicates internal record structure used to access records in a file. If the file is old, this option is ignored. Specifying a designator option (<i>foption</i> bits (10:3)) value other than zero overrides this option. This option is applicable only at file creation. The following bit settings are valid: 000 Standard (STD) file 001 Keyed sequential access method (KSAM) file 010 Relative I/O (RIO) file 100 Circular (CIR) file 110 Message (MSG) file Default: 000
0:2	Reserved for MPE

aoption**logical by value (optional)**

Specifies up to eight different file access options by setting corresponding bit groupings. Refer to Table 2-9 for their values

Table 2-9. Aoption Parameter Descriptions

Bits	Value/Meaning
12:4	Access type option Indicates the type of access intended for the file. This option restricts/allows minimal use of file system intrinsics. The following bit settings are valid: 0000 Allows READ access only if the file's security provisions specify READ access. FWRITE, FUPDATE, and FWRITEDIR intrinsic calls cannot reference this file. The end-of-file (EOF) is not changed; the record pointer starts at 0. 0001 Allows WRITE access only if the file's security provisions allow WRITE access. Any data written in the file prior to the current FOPEN request is deleted. FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file. The EOF is set to 0; the record pointer starts at 0. On magnetic tape an EOF is written when the file is closed even if no data is written. 0010 Allows WRITE-SAVE access only if the file's security provisions allow WRITE access. Previous data in the file is not deleted. FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file. The EOF is not changed; the record pointer starts at 0. Therefore, data is overwritten if a call to FWRITE is made. The system changes this value to APPEND for message files. 0011 Allows APPEND access only if the file's security provisions allow either APPEND or WRITE access. FREAD, FREADDIR, FREADSEEK, FUPDATE, FSPACE, FPOINT, and FWRITEDIR intrinsic calls cannot reference this file. The EOF sets the record pointer prior to each FWRITE call. For disk files, the EOF is updated after each FWRITE call. Therefore, data cannot be overwritten. 0100 Allows READ/WRITE (I/O) access only if the file's security provisions allows both READ and WRITE access. If both READ and WRITE access are not allowed, the access type specified in the security provisions (either READ or WRITE) is allowed. Any file intrinsic except FUPDATE can be called for this file. The EOF is not changed; the record pointer starts at 0. This option is not valid for message files. 0101 Allows UPDATE access only if the file's security provisions allows both READ and WRITE access. If both READ and WRITE access are not allowed, the access type specified in the security provisions (either READ or WRITE) is allowed. All file intrinsics, including FUPDATE, can be called for this file. The EOF is not changed; the record pointer starts at 0. This option is not valid for message files.

Table 2-9. Aoption Parameter Descriptions (continued)

Bits	Value/Meaning
	0110 Allows EXECUTE access only if the file's security provisions allow EXECUTE access. This access allows READ/WRITE access to any loaded file. The program must be running in PM to specify EXECUTE access. This option is not valid for message files. Default: 0000
11:1	Multirecord option Indicates whether or not individual read or write requests are confined to record boundaries. The following bit settings are valid: 0 Nonmultirecord mode (NOMULTI) 1 Multirecord mode (MULTI) Default: 0 If the number of words or bytes to be transferred (specified in the <i>length</i> parameter of the read or write request) exceeds the size of the physical record that is referenced, the remaining words or bytes are taken from subsequent successive records until the number specified by <i>length</i> has been transferred. For message (MSG) files not accessed with the copy mode option (<i>aoption</i> bit (3:1)) enabled, the file system sets this option to zero. This option is available only if the inhibit buffering option (<i>aoption</i> bit (7:1)) is set to 1 (NOBUF).
10:1	Dynamic locking option Enables/disables file locking for the file. When this option is specified, the FLOCK and FUNLOCK intrinsics can be used to dynamically permit or restrict concurrent access to a disk file by other processes at specified times. The following bit settings are valid: 0 Disallow dynamic locking/unlocking 1 Allow dynamic locking/unlocking Default: 0 The process can continue this temporary locking/unlocking until it closes the file. If several users are sharing the file, they must all specify, or not specify, this option. For example, if a file is opened with the dynamic locking option enabled, and a subsequent user tries to open the file with dynamic locking disabled, the subsequent attempt to open fails. Dynamic locking/unlocking is made possible through the equivalent of a global resource identification number (RIN) assigned to the file and temporarily acquired by FOPEN. Cooperating users that have opened a file with the dynamic locking option enabled must access the file using either the FLOCK and FUNLOCK intrinsics to ensure exclusive use of the file. These users are allowed concurrent access even when not using FLOCK and FUNLOCK, but exclusiveness is not guaranteed.

Table 2-9. Aoption Parameter Descriptions (continued)

Bits	Value/Meaning
	LOCK access must be at account, group, and file levels for FOPEN to grant the dynamic locking option. (LOCK access is available if LOCK, EXECUTE, APPEND, or WRITE access is set at these levels.) This option is ignored for files not residing on disk.
8:2	Exclusive option Indicates continuous exclusive access to this file, from open to close. Use this option when performing a critical operation (for example, updating the file). The following bit settings are valid: 00 If access type option (<i>aoption</i> bits (12:4)) specifies READ only access, then READ-SHARE access takes effect. Otherwise, EXCLUSIVE access takes effect. Regardless of which access option was selected, FFILEINFO reports zero. 01 EXCLUSIVE access. After the file is opened, any additional FOPEN requests for this file, whether issued by this process or another process, are prohibited until this process issues the FCLOSE request or terminates. If any process already is accessing this file when an FOPEN call is issued with EXCLUSIVE access specified, an error status is returned to the process. If another FOPEN call is issued for this file while EXCLUSIVE access is in effect, an error code is returned to the process that issued that FOPEN call. Request EXCLUSIVE access only if the LOCK access mode is allowed by the security provisions for the file. For message files, specifying this value indicates there can be only one reader and one writer. 10 READ-SHARE access (SEMI-EXCLUSIVE access). After the file is opened, concurrent WRITE access to this file through another FOPEN request is prohibited, whether issued by this process or another process, until this process issues the FCLOSE request or terminates. A subsequent request for the READ/WRITE or UPDATE access type option (<i>aoption</i> bits (12:4)) obtains READ access. However, other types of READ access are allowed. If a process already has WRITE access to the file when this call is issued, an error code is returned to the calling process. If another FOPEN call that violates the READ only restriction is issued while READ-SHARE access is in effect, that call fails and an error code is returned to the calling process. Request READ-SHARE access only if the LOCK access mode is allowed by the security provisions for the file. For message files, this value specifies there can be multiple readers, but only one writer.



Table 2-9. Aoption Parameter Descriptions (continued)

Bits	Value/Meaning
11	SHARE access. After the file is opened, concurrent access to this file by any process is permitted, in any access mode, subject to other security provisions in effect. For message files, this value specifies that there can be multiple readers and multiple writers. Default: 00
7:1	Inhibit buffering option Enables/disables automatic buffering by MPE . If NOBUF is specified, I/O is allowed directly between the data area and the applicable hardware device. The following bit settings are valid: 0 Allow normal buffering (BUF) 1 Inhibit buffering (NOBUF) Default: 0 NOBUF access is for block transfer; not logical record transfer. If NOBUF is specified with this option as well as a variable-length record structure in record format option (<i>foption</i> bits (8:2)) and the file does not have a variable-length record format, then the format is changed internally to an undefined-length record format. Therefore, you are responsible for buffer management. When performing an FWRITE, the variable structure must be set up. NOBUF access assumes responsibility for blocking and deblocking of records in the file. To be consistent with files built using buffered I/O, records should begin on word boundaries. When the information content of the record is less than the defined record length, pad the record with blanks if the file is ASCII, or with zeros if the file is binary. The record size and block size for files with NOBUF specified follow the same rules as those files that are created using buffering. The default blocking factor for a file created under NOBUF is 1. If a file is opened NOBUF without multirecord mode specified in multirecord option (<i>aoption</i> bit (11:1)), a maximum of only one block of data per read or write can be transferred. The end-of-file (EOF) marker, 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 by the size of the transfer.

Table 2-9. Option Parameter Descriptions (continued)

Bits	Value/Meaning
	Transfers always begin on a block boundary. Those transfers that do not transfer whole blocks leave the next record pointer set to the first record in the next block. The EOF marker always points at the last record in the file. For files opened NOBUF, the FREADDIR, FWRITEDIR, and FPOINT intrinsics treat the <i>recnum</i> parameter as a block number. Indicate non-RIO access to an RIO file by specifying the file NOBUF. Use the physical block size from FFILEINFO to determine the maximum transfer length. For message files, the file system normally resets the inhibit buffering option bit to zero. However, a message file can be opened with NOBUF if the copy mode option (<i>aoption</i> bit (3:1)) is set to 1; this determines whether access to the file is record-by-record or block-by-block. If you are reading a message file with the copy mode option enabled, the inhibit buffering option has the following meaning: 0 Read by logical record 1 Read by block If writing to a message file, open it NOBUF if the copy mode option is enabled; access the file block-by-block.
5:2	Multiaccess mode option Indicates how the file's record pointer is to be shared. This option is useful for sharing standard input devices where there is some natural sequence of access to the file. This option permits processes located in different jobs or sessions to open the same file and share that file's record pointer. The following bit settings are valid: 00 No MULTI access. A unique record pointer is created for this access to the file. For message files, the file system sets this option to bits (5:2)=10 if this option is not specified or is set to 00. 01 Intrajob MULTI access only allowed. A record pointer is shared with all other opened files of the same name in the same job/session who opened the file with this option set to either 01 or 10. 10 Interjob GMULTI access allowed. A record pointer is shared with all other opened files of the same name on the system. Default: 0

Table 2-9. Aoption Parameter Descriptions (continued)

Bits	Value/Meaning
4:1	NOWAIT I/O option Enables/disables NOWAIT I/O. Determines whether or not the user has control returned before the completion of an I/O request. The NOWAIT I/O implies the inhibit buffering option (<i>aoption</i> bit (7:1)); if NOBUF is not specified, the file system does it for you and multirecord access is not available. This option is not available if the file is located on a remote computer. When opening nonmessage files, the process must be running in privileged mode to specify this option. The following bit settings are valid: 0 NOWAIT I/O not in effect 1 NOWAIT I/O in effect Default: 0
3:1	Copy mode option Permits any file to be temporarily treated as a standard sequential file so that file transfers can occur by logical record or block (by passing restrictions on data transfer imposed by other file types). The following values are valid: 0 The file is accessed as its own file type. 1 The file is treated as a standard (STD) file, with variable length records. For message files, this allows nondestructive reading of an old message file at either the logical record or block level. Only block-level access is permitted if the file has message file format. This prevents incorrectly formatted data from being written to the message file while it is unprotected. To access a message file in COPY mode, a process must have EXCLUSIVE access to the file. Default: 0
0:3	Reserved for MPE.

*resize***integer by value (optional)**

The size, in words or bytes, of the logical records in the file. Positive values are words, negative values are bytes. The valid range is dependent on storage and record formats:

- For fixed-length and undefined-length ASCII files, the valid range is 1 ... 32,767 bytes.
- For variable-length ASCII files and fixed-length, variable-length, and undefined-length binary files, the range is 1 ... 32,766 bytes (1 ... 16,383 words). All odd values specified are rounded up to the next even value (the next word boundary).

Default: Device dependent.

*device***byte array (optional)**

Contains a string of ASCII characters terminating with any nonalphanumeric 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 files for the HP 268x page printer (**ENV=** parameter). This parameter may be specified in one of the following forms: *devclass* or *ldev*. 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 the *devclass* form is specified, the file is allocated to any available device in that class. To open a file which is to reside on a private volume, you must specify a device class which includes those disk drives upon which the home volume set is mounted; the file then is allocated to any of the home volume set's drives that fall within that device class.

The logical device number (*ldev*) consists of a three-byte numeric string specifying a particular device.

If you open a foreign disk file, *device* must be either a foreign disk class name or the *ldev* of a disk in a foreign disk class. If you specify *ldev*, the disk should be mounted on the drive prior to the FOPEN. Otherwise it may be assumed to be a serial disk by the system. Any of the forms may be used to reference files on a remote computer by preceding the device or volume specification with *dsdevice#*.

Effective with the G.02.00 release, the remote location of a device may be specified with *device* parameter as *nodename#ldev;VTERM*CR

Note

When opening a magnetic tape as shared for a second time, the device must be opened by *ldev* instead of *devclass*. This is to ensure that the system operator does not get confused by a second tape request. The *ldev* may be programmatically obtained through FGETINFO.

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* string is not specified but the *device* parameter is, then a semicolon must be the first character of the *device*

FOPEN

parameter. The entire *device* parameter string must be terminated by a carriage return.

The keyword DEN= is applicable only when writing to tape on a tape drive that supports more than one density. The keyword DEN= is ignored at all other times.

When reading from 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",%15
BYTE ARRAY DEVICE(0:9):=";DEN=6250",%15
```

```
NUM:=FOPEN(FILEX,%4,%4,,DEVICE);
```

If you are opening a HP 268x page printer file, 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, including the page size, the margin width, the character set, the orientation (horizontal or vertical), and the name of 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. Hewlett-Packard provides a number of prepared environment files, which reside in the HPENV (G.00.00 and later) or ENV2680A (earlier versions) group of the SYS account. For information on how to build your own printing environments, refer to the IFS/3000 Reference Manual (36580-90001).

To specify your own printer environment you must also assign the 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 HP 268x printer:

```
EQUATE CR=%15;
BYTE ARRAY DEVICE(0:16):="PP;ENV=MYENVFILE",CR;
```

```
NUM:=FOPEN(FILEX,%4,%4,,DEVICE);
```

Any environment you select remains active until it is replaced by a new environment or until you **FCLOSE** the printer. If the printer has been opened with the **MULTI** access *aoption* (for example, as **\$STDLIST**), a selected environment remains active until replaced or until the final **FCLOSE** of the printer.

For Interprocess Communication (IPC), this field is relevant only if this is a new message file. The *device* field must either be omitted or must specify a disk; specification of any device other than disk opens the device. When this occurs, the file is no longer a message file.

formmsg

byte array (optional)

A forms message that can be used for sending messages (for example, telling the system operator what type of paper to use in the line printer) The message 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.

Used for tape label information if bit (6:1) of the *foption* parameter is set. The tape label information is formatted as follows:

```
. [valid[, type[, exp[, seq]]]];
```

The period is required so that the tape label information is not mistaken for a forms message.

valid Consists of six or fewer printable characters that identify the volume. In a multivolume set, specify only the first *valid*.

type Three alphabetic characters that identify label type information. The options are:

ANS - ANSI standard labels (default)

IBM - IBM standard labels

FOPEN

exp Month/day/year of the expiration date of the file or the date when the information contained in the file is no longer valid. 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 Use one of the following methods to specify the position of the file in relation to other files on the tape:

- A zero, which causes a search of all volumes until the file is found.
- An unsigned integer (1-9999), which specifies the position of the file relative to the current file on the tape.
- ADDF which causes the tape to be positioned to add a new file on the end of the volume (or last volume in a multivolume set).
- NEXT which positions the tape at the next file on the tape. If this is not the first FOPEN for the file and a rewind occurred on the last close, then the position remains at the beginning of the previous file.

Default: NEXT

*userlabels***integer by value (optional)**

The number, in the range 0..254, of user-label records to be created for the file. Applicable to new disk files only. The maximum number of user labels allowed varies from file to file. It depends on the final *blockfactor* and *recsize* used, as well as whether you 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: 0

*blockfactor***integer by value (optional)**

The number of logical records in one block. This value is used to calculate the size for disk and magnetic tape files. The valid range for this option is 1..255. If a value greater than 255 is specified, FOPEN resets the value to 255. If a value less than 1 is specified, FOPEN resets the value to the default *blockfactor* setting.

For fixed-length records, *blockfactor* specifies the actual number of records in a block. For variable-length records, *blockfactor* is interpreted as a multiplier used to compute the block size (*recsize*blockfactor*). For undefined-length records, *blockfactor* is always one logical record per block.

Default: For files opened BUF, the default is calculated by dividing the specified *recsize* into the block size configured for the device. For files opened NOBUF, the default is 1.

FOPEN

numbuffers **integer by value (optional)**

The number of buffers, number of copies, and output priority indicated by setting the following bits:

Bits	Value/Meaning
11:5	Numbuffers Indicates the number of buffers to allocate to the file. The valid range for this option is the binary equivalent of 1..31. The valid range for message files is 2..31. This option must not specify a number of buffers whose combined size exceeds the physical capacity of the file. Default: 00010 (decimal 2)
4:7	Spooler copies Indicates the number of copies of the file to be produced by the spooling facility. The valid range is 0 . . . 127. This option is applicable to spooled devices only. Specify this option for a file already opened (for example, \$STDLIST), this is the highest value supplied before the last FCLOSE takes effect. The copies do not appear continuously if the system operator intervenes or if a file of higher output priority becomes READY before the last copy is complete. Default: 0000001 (decimal 1)

Bits	Value/Meaning
------	---------------

0:4	Output priority
-----	-----------------

Indicate the output priority to be attached to the file for spooled output. Applicable only to spooled devices. The valid range for this option is the binary equivalent of 1..13. The output priority must be a number between 1 (lowest priority) and 13 (highest priority), inclusive. If a value less than the current OUTFENCE set by the system operator is specified, file printing is deferred until the operator raises the output priority of the file or lowers the OUTFENCE. Specify this option for a file already opened (for example, \$STDLIST), the highest value supplied before the last FCLOSE takes effect.

Default: 1000 (decimal 8)

filesize

double by value (optional)

The maximum file capacity. For variable-length and undefined-length records, the capacity is expressed in blocks (block = *recordsize* * *blockfactor*). For fixed-length records, the capacity is expressed in logical records. Applicable only at file creation.

The maximum file size is 2 sectors. However, some other file options, such as the blocking factor, may prevent the maximum file size from being reached. In general, the *filesize* is determined by the extent size and the number of extents (maximum= 32).

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

For message files, 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. Because of spare tracks or remapped tracks, 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 contiguous disk 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 userlabels (if any), are added to the number of physical records required for data. To determine extent size, this number is divided by the request *numextents* to determine extent size. 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 disk in the device class specified in the *device* parameter when the file was created. If it is necessary to ensure that all extents of a file are on a particular disk, a single disk 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 disk space fails, the **FOPEN** intrinsic returns an error condition code to the calling program.

Default: 1 extent.

*filecode***integer by value (optional)**

A value that can be used as a file code to identify the type of file. This code is recorded in the file label and is accessible through the **FFILEINFO** or **FGETINFO** intrinsics. Applicable only at file creation (except when opening an old file that has a negative file code).

If the program is running in user mode, specify a file code in the range 0..32,767 to indicate the file type being created; programs running in user mode can access files with non-negative file codes. If the program is running in privileged mode, specify a file code in the range -32,768..32,767; programs running in privileged mode can access files with a file code in the range -32,768..32,767. If an old file with a negative file code is opened, the file code specified must match the file code in the file label.

Default: 0

Operation Notes

A file can be referenced by its formal file designator. When executed, a unique file number is returned to the process. This file number can be used, rather than the formal file designator, in other calls to this file.

Condition Codes

CCE

Request granted. The file is open

CCG

Not returned.

CCL

Request denied. Either another process already has **EXCLUSIVE** or **SEMI-EXCLUSIVE** access for this file, the privilege level of this file is not user (3), or an initial allocation of disk space cannot be made due to lack of disk space. If the file is not opened successfully, the file number value returned by **FOPEN** is zero. Call the **FCHECK** intrinsic for more details.

Related Information

Additional
Discussion

“File Device Relationships” and “How to Use Files” in Chapter 4.

FPARSE

Parses and validates file designators. (Available on version G.02.00 or later) (No intrinsic number assigned)

BA IA IA DA
FPARSE(<i>formaldesig</i> , <i>result</i> , <i>items</i> , <i>vector</i>);

Parameters	<i>formaldesig</i> byte array (required) Passes the file reference string to be parsed. Terminate the string with any nonalphanumeric character except a slash (/), period (.), or colon (:). If an error occurs, the first element of the <i>vector</i> array returns the byte offset of the invalid item in <i>formaldesig</i> . <i>result</i> integer array (required) Returns two elements; the first element is the error number, the second element is reserved and always returns 0. A positive value indicates the syntax of the specified file reference is correct and what type of file reference is being made. A negative value indicates a syntax error in the file reference. The possible error number values are:
-------------------	---

Value	Meaning
0	Regular file designator
1	Backreference (* is the first character in <i>formaldesig</i>)
2	System file (\$) is the first character in <i>formaldesig</i>)

The default designator numbers for system files, as defined for **FOPEN** *foption*, are:

- 0 File name
- 1 \$STDLIST
- 2 \$NEWPASS
- 3 \$OLDPASS
- 4 \$STDIN
- 5 \$STDINX
- 6 \$NULL



Value	Meaning
-1	Invalid item values
-2	Parameter bounds violation
-3	Illegal delimiter; misuse of ., /, or :
-4	<i>Items</i> specified without specifying <i>vector</i>
-5	Illegal value in <i>items</i>
-6	Item list not terminated by 0
-7	Undefined system file
-8	Lockword not allowed in backreferenced (*) file
-9	NS 3000 not present; <i>envidname</i> was specified
-101	First character of the file name not alphabetic
-102	File name expected in the string
-103	File name identifier too long
-104	First character of lockword not alphabetic
-105	Lockword expected in string
-106	Lockword identifier too long
-107	First character of group name not alphabetic
-108	Group name expected in string
-109	Group name identifier too long
-110	First character of the account name not alphabetic
-111	Account name expected in string
-112	Account name identifier too long
-113	First character of <i>envidname</i> not alphabetic
-114	Expected <i>envidname</i> in string
-115	Identifier for <i>envidname</i> too long

Note

Up to three identifiers can be specified, separated by a period (.), in an *envidname*; errors -113 through -115 can reference one of the three identifiers. The error pointer indicates the location.

If NS 3000 is not installed on the system, FPARSE returns a -9 for file designators with an *envidname* after the file name. This error indicates that everything up to *envidname* was valid, but a parsed *vector* array is not returned.

items **integer array (optional)**

Passes an item code, one value per element, indicating what the corresponding *vector* parameter points to:

Value	Meaning
0	End of item list
1	File name
2	Lockword
3	Group name
4	Account name
5	NS 3000 <i>envidname</i>

vector **double array (optional)**

Returns vectors, one per element, for the requested *item*.

Note



A *vector* cannot be passed without a corresponding *item* value.

Information is returned as follows:

Bits	Value/Meaning
16:16	Return the length of the string specified by the corresponding <i>items</i> parameter. If zero is returned, the requested item in <i>formaldesig</i> was not specified.
0:16	Return the offset, in bytes, from the base of the <i>formaldesig</i> parameter to the first element of the string segment specified by the corresponding <i>item</i> parameter.

The last element returns the file designator length passed in *formaldesig* (bits (16:16)) and the default type (bits (0:16)) if the *result* parameter indicates a system file.

Related Information

Additional Discussion

“Parsing and Validating File Designators” in Chapter 4

Manuals

NS 3000 User/Programmer Reference Manual (36920-90001)

FPOINT

Sets the logical record pointer for a disk file (except serial disk) containing fixed-length or undefined-length records to any logical record. When the next **FREAD** or **FWRITE** file request is made, this record is read or written to. (Intrinsic number 6)

Syntax

IV DV
FPOINT(<i>filenum</i> , <i>lrecnum</i>);

Parameters

filenum

integer by value (required)

The file number of the file where the pointer is to be set.

lrecnum

double by value (required)

The relative logical record (or block number for a NOBUF file) where the logical record pointer is to be positioned. The number of the first logical record is zero.

On disk files, the EOF indicator is the file limit.

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:

- *Lrecnum* < 0
- Invalid *filenum* parameter
- I/O is pending on a NOWAIT request
- Spooled file
- Not a direct access disk file
- File does not contain fixed-length or undefined-length records
- Not allowed with APPEND access

Operation Notes

Not applicable to message files.

Related Information

Manual

MPE File System Reference Manual(30000-90236).

Special
Consideration

Split-stack calls are permitted

FREAD

Reads a logical record or portion of a record from a file to the stack.
(Intrinsic number 2)

Syntax

I	IV	LA	IV
<i>transfercount</i> := FREAD(<i>filenum</i> , <i>buffer</i> , <i>length</i>);			

Functional Return

transfercount integer (optional)

Returns the length of the data transferred to *buffer*:

- If a negative value is passed in the *length* parameter, the *transfercount* is a positive value indicating the number of bytes transferred.
- If a positive value is passed in the *length* parameter, the *transfercount* is a positive value indicating the number of words transferred.

Note



If the file was opened with the NOWAIT I/O option enabled, the *transfercount* is zero. The IOWAIT intrinsic can be used to determine the length of the data transferred.

Parameters

filenum integer by value (required)

The file number of the file to be read.

buffer logical array (required)

Returns the record that was read. This structure must be large enough to hold all of the information to be transferred.

length integer by value (required)

The length of the data to be transferred to *buffer*:

Value	Meaning
<0	Length in bytes
=0	No transfer occurs
>0	Length in words

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

Operation Notes

When the logical end-of-data is encountered, CCG is returned to the process:

- On magnetic tape, the end-of-data is a physical indicator such as a tape mark. When the program reads a file that spans more than one volume of labeled magnetic tape, the program is suspended until an operator mounts the next tape. CCG is not returned when end-of-tape is encountered.
- On disk, the end-of-data occurs when there is an attempt to read past the last logical record of the file. In this case, CCG is returned, and no record is read.
- If the file is embedded in an input source containing commands, an end-of-data occurs when an EOD command is encountered. End-of-data is indicated by a hardware end-of-file, including EOF (on \$STDIN), any record beginning with a colon (:), or on \$STDINX, by EOD.
- On a standard input device for a job (not a session) JOB, EOJ, or DATA indicate end-of-data.

When reading from an empty message file and another process has opened that file for WRITE access, the process waits. If the message file is empty and there are no writers, the process waits if there is an FCONTROL=45 in effect or if this is the first FREAD after the reader's FOPEN. Otherwise, CCG is returned. If an FREAD is issued against a message file and an FCONTROL=46 is in effect, the writer's ID and the record type code are appended to the beginning of the record.

FREAD

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

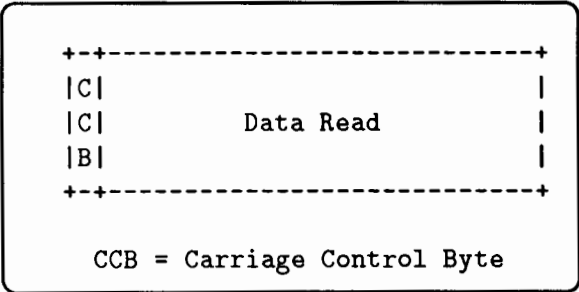


Figure 2-8. Carriage Control Byte

Condition Codes	CCE	Request granted. The information was read.
	CCG	Request denied. 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. CCG is returned at actual end-of-file, with a transmission log of 0 if there is an attempt to read past end-of-file.
	CCL	Request denied. The information was not read because 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. Check the condition codes both in normal I/O and in NOWAIT I/O.
Related Information	Additional Discussion	“Using FREAD and FWRITE with \$STDIN and \$STDLIST ” in Chapter 4.

FREADBACKWARD

Reads a logical record backward from the current record pointer. Data is presented as if read forward. Can be used to recover tape errors when handling I/O management and data recovery routines. (No intrinsic number assigned)

Syntax

I	IV	LA	IV
<i>transfercount</i> := FREADBACKWARD(<i>filenum</i> , <i>buffer</i> , <i>length</i>);			

Functional Return

transfercount integer (required)

Returns the length of the data transferred to *buffer*:

- If a negative value is passed in the *length* parameter, the *transfercount* is a positive value indicating the number of bytes transferred.
- If a positive value is passed in the *length* parameter, the *transfercount* is a positive value indicating the number of words transferred.

Note



If the file is opened with the NOWAIT I/O option enabled, the *transfercount* is zero. The IOWAIT intrinsic can be used to determine the length of the data transferred.

Parameters

filenum integer by value (required)

The file number of the file to be read.

buffer logical array (required)

The specified record. This structure must be large enough to hold all of the information to be returned.

length integer by value (required)

The length of the data to be transferred to *buffer*:

Value	Meaning
<0	Length in bytes
=0	No transfer occurs
>0	Length in words

If *length* is larger than the size of the logical record, and the multirecord *aoption* in FOPEN was not specified, transfer is limited to the length of the logical record. If the multirecord *aoption* in FOPEN is specified, transfer continues until either *length* is satisfied or end-of-data is encountered, and each

FREADBACKWARD

transfer begins at the start of the next physical record (block). Any data remaining in the last physical record read is inaccessible.

Operation Notes

Three restrictions apply when using **FREADBACKWARD**:

- Used only with a magnetic tape drive with read-reverse capability.
- The magnetic tape drive must be located on an HP-IB system.
- The magnetic tape NOBUF must be accessed.

Condition Codes

CCE	Request granted.
CCG	Request denied. The logical beginning-of-data was encountered. When reading a labeled magnetic tape file that spans more than one volume, CCG is not returned when beginning-of-tape (BOT) is encountered. 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	Request denied. The information was not read because a tape error occurred, or a tape error was recovered and the FSETMODE option was enabled. Check the condition code in both normal I/O and NOWAIT I/O.

Related Information

Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
Special Considerations	Split-stack calls permitted.

FREADDIR

Reads a specific logical record or portion of a record from a direct-access disk file to the data stack. (Intrinsic number 7)

Syntax

IV	LA	IV	DV
FREADDIR(<i>filenum</i> , <i>buffer</i> , <i>length</i> , <i>lrecnum</i>);			

Parameters

filenum

integer by value (required)

The file number of the file to be read.

buffer

logical array (required)

Returns the record that was read. This structure should be large enough to hold all of the information to be transferred.

length

integer by value (required)

The number of words or bytes to be transferred. If this value is positive, it signifies words; if negative, bytes. Zero signifies that no transfer occurs. If *length* is less than the size of the logical record, only the first *length* words or bytes are read from the record.

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

lrecnum

double by value (required)

The relative number, in the file, of the logical record or block number for the NOBUF files to be read. A value of zero indicates the first logical record.

FREADDIR

Operation Notes

This intrinsic is different than the **FREAD** intrinsic. The **FREAD** intrinsic reads only the record already pointed to by the logical record pointer. Issue the **FREADDIR** intrinsic only for disk files composed of fixed-length or undefined-length records. If RIO access is used, **FREADDIR** inputs the specified logical record. If the record is inactive, the contents of the inactive record are transmitted and a CCE is returned; there is no indication of the block containing some inactive records. (**FCHECK** returns a nonzero error number to distinguish active and inactive records.) If an RIO file is accessed using the non-RIO method (**NOBUF**), **FREADDIR** inputs the specified block.

After the **FREADDIR** intrinsic is executed, the logical record pointer is set to the beginning of the next logical record, or the 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 *length* parameter specified is greater than one logical record. For example, if all of record 11 and half of record 12 is read in a file, the logical record pointer is set to the beginning of record 13 after the **FREADDIR** intrinsic executes; the second half of record 12 is skipped.

Not applicable to message files.

Condition Codes

CCE	Request granted. The information was read.
CCG	Request denied. End-of-file was encountered.
CCL	Request denied. The information was not read; an error occurred.

Related Information

Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
Special Considerations	Split-stack calls permitted.
	Not applicable to message files.

FREADLABEL

Reads a user-defined label from a disk or magnetic tape file.
(Intrinsic number 19)

Syntax

O-V	IV	LA	IV	IV
FREADLABEL(<i>filenum</i> , <i>buffer</i> , <i>length</i> , <i>labelid</i>);				

Parameters

<i>filenum</i>	integer by value (required) The file number of the file whose label is to be read.
<i>buffer</i>	logical array (required) Returns the label that was read. This structure must be large enough to hold the number of words specified by <i>length</i> .
<i>length</i>	integer by value (optional) The number of words to be transferred from the label, <=128. Default: 128 words
<i>labelid</i>	integer by value (optional) The label number. For labeled tapes, <i>labelid</i> is ignored. The next sequential label is read. Default: Zero

Operation Notes

When a disk file is opened, user labels can be read from, or written to, in any order, at any time, regardless of access capabilities to the rest of the file. A disk file can have <=254 128-word user-defined labels.

A magnetic tape file (if labeled) must be labeled with an ANSI-standard or IBM-standard label. MPE automatically skips over any unread user-defined labels when the first FREAD intrinsic call for files is issued. To read a user-defined label, you should call the FREADLABEL intrinsic immediately after an FOPEN intrinsic has opened the file. The user-defined label must be 40 words in length to conform to the length of the ANSI-standard or IBM-standard label.

FREADLABEL

Condition Codes	CCE	Request granted. The label was read.
	CCG	Request denied. A label was referenced beyond the label written on the file.
	CCL	Request denied. The label was not read; an error occurred.
Related Information	Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
	Special	Split-stack calls permitted.
	Considerations	

FREADSEEK

Moves a record from a disk file to a buffer in anticipation of a FREADDIR intrinsic call. (Intrinsic number 12)

Syntax

IV	DV
FREADSEEK(<i>filenum</i> , <i>lrecnum</i>);	

Parameters*filenum***integer by value (required)**

The file number of the file to be read.

*lrecnum***double by value (required)**

The relative number of the logical record to be read.
A value of zero indicates the first logical record.

Operation Notes

The FREADSEEK intrinsic enhances direct access of disk files. When a certain record is required, use FREADSEEK before a call to FREADDIR to transfer it to the stack. The FREADSEEK intrinsic directs MPE to move the record from disk into a buffer in anticipation of the FREADDIR call, which moves the record directly to the stack.

Issue the FREADSEEK intrinsic call only for direct-access files where I/O buffering and fixed-length or undefined-length records are in effect.

Not applicable to message files.

Condition Codes

CCE

Request granted.

CCG

Request denied. A logical end-of-file indication was encountered.

CCL

Request denied. An error occurred.

Related Information

Manuals

MPE File System Reference Manual(30000-90236).

Special

Split-stack calls permitted.

Considerations

FREEDSEG

Releases an extra data segment assigned it by the GETDSEG intrinsic. Data segment management (DS) capability is required. (Intrinsic number 131)

Syntax

LV LV FREEDSEG(<i>index</i> , <i>id</i>);

Parameters

index

logical by value (required)

The index assigned to the data segment, obtained from a GETDSEG intrinsic call.

id

logical by value (required)

The identification assigned to the segment. Enter zero if none is assigned.

Operation Notes

If this is a private data segment or a sharable (nonprivate) segment not currently assigned to any other process in the job/session, the segment is deleted from the entire job/session. If it is a sharable segment that is currently assigned to another process, it is deleted from the calling process but continues to exist in the job/session.

If the process is running in privileged mode when the extra data segment is acquired, then the process must be in privileged mode when it invokes FREEDSEG.

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; 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 shared data segment.
Related Information	Additional Discussion	“Deleting an Extra Data Segment” in Chapter 3.
	Intrinsics	GETDSEG

FREELOCRIN

Frees all local resource identification numbers (RINs) from allocation to a job/session. (Intrinsic number 31)

Syntax

`FREELOCRIN;`

Operation Notes

The `FREELOCRIN` intrinsic must be called before a subsequent call to `GETLOCRIN`.

Condition Codes

CCE	Request granted.
CCG	Request denied. No RINs are currently reserved for the job.
CCL	Request denied. A minimum of one RIN is currently locked by a process.

Related Information

Additional Discussion	“Resource Management” in Chapter 3.
-----------------------	-------------------------------------

FRELATE

Determines whether a file pair (on any device) is interactive, duplicative, or both interactive and duplicative. (Intrinsic number 18)

Syntax

L

IV

IV

intordup:=FRELATE(*infilenum*,*listfilenum*);

Functional Return

intordup logical (optional)

Returns values indicating whether the two files referenced are interactive and/or duplicative:

Bits	Value/Meaning
15:1	0 Files specified by <i>infilenum</i> and <i>listfilenum</i> do not form an interactive pair.
	1 Files specified by <i>infilenum</i> and <i>listfilenum</i> form an interactive pair.
0:1	0 Files specified by <i>infilenum</i> and <i>listfilenum</i> do not form a duplicative pair.
	1 Files specified by <i>infilenum</i> and <i>listfilenum</i> form a duplicative pair.

Parameters

infilenum integer by value (required)
The file number of the input file.

listfilenum integer by value (required)
The file number of the list file.

Operation Notes

A device file is interactive if it requires human intervention for all input operations; this 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.

The interactive/duplicative attributes of a file pair do not change between the time the files are opened and closed.

Condition Codes

CCE	Request granted.
CCG	Request denied. Infilenum and/or listfilenum corresponds to \$NULL (a logical file that contains no data). No data can be read from this file, and all data written to it is discarded. Interactive or duplicative functions do not apply.
CCL	Request denied. An error occurred.

Related Information

Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
Special Considerations	Split-stack calls permitted.

FRENAME

Renames an open disk file (and its lockword, if applicable). The file being renamed must be either:

- A new file.
- An old file (permanent or temporary), opened for EXCLUSIVE access with the *exclusive* option of the FOPEN intrinsic, and with security provisions allowing WRITE access. If the file is permanent, you must be the creator. (Intrinsic number 17)

Syntax

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

Parameters

filenum integer by value (required)

The file number of the file to be renamed.

formaldesig byte array (required)

The new name of the file. The maximum number of characters allowed in the string is 36. The ASCII string contained in ***formaldesig*** must begin with a letter and can contain up to eight alphanumeric characters for each of the *filename*, *lockword*, *group*, and *account* fields. The string must end with a nonalphanumeric character, including a blank, but not a slash (/) or a period (.). The home volume set of ***formaldesig*** must be the same as the file being renamed. Volume sets cannot be spanned when renaming files. The format of ***formaldesig*** is:

filename/lockword.group.account

where:

filename Is the new file name for the file.
(Required in ***formaldesig***.)

lockword Is a lockword for the new file name.
(Optional portion of ***formaldesig***.)
To keep or add a lockword to the file, the *lockword* must be entered in the ASCII string. If this part of ***formaldesig*** is not specified, the new file named has no lockword associated with it.

FRENAME

<i>group</i>	Is the group where the file is to reside. (Optional portion of <i>formaldesig</i> .) If a group is not specified, the file resides in the group it was assigned before the FRENAME intrinsic call.
<i>account</i>	Is the account name where the file is to reside. (Optional portion of <i>formaldesig</i> .) If renaming a new or temporary file that was created, specify any account that shares the same volume set as the file being renamed. A permanent file cannot be renamed across account boundaries. If other than the current account name is specified for a permanent file, the CCL error condition is returned and the file retains its old name.

Condition Codes	CCE	Request granted.
	CCG	Not returned.
	CCL	Request denied. An error occurred.

FSETMODE

Controls the following access modes of files or devices:

- Issuing carriage return and line feed to terminal after a terminal read.
- Reporting tape automatic error recovery.
- Guaranteeing chronological order of user program write requests (serial writequeue).
- Blocking program execution until physical completion of write requests (BLOCKONWRITE).

(Intrinsic number 14)

Syntax

IV	LV
FSETMODE(<i>filenum</i> , <i>modeflags</i>);	

Parameters

filenum

integer by value (required)

The file number of the file whose access modes are to be set.

modeflags

logical by value (required)

The access mode option:

Bits	Value/Meaning
15:1	Controls chronological order of user program writes (serial write queue). This mode controls write requests to the file from all writers on the system. This bit is effective only on systems with disk caching enabled. With disk caching, it is possible for I/O's to be posted in a different order than sent.
0	Do not change the write queue where this file is being accessed
1	Access this file through the serial write queue

FSETMODE

Bits	Value/Meaning
14:1	Controls blocking of program execution until completion of physical (block) write operations. BLOCKONWRITE writes to <i>filenum</i> only from the calling process. This bit is only effective for files opened BUF. 0 Return control to the program from a write request without waiting for completion of the physical write operation 1 Force the program to wait until the physical (block) write operation is completed (when full data buffers are posted)
13:1	Controls issuing carriage return and line feed to a terminal after each terminal read. 0 Issue a carriage return and line feed at the completion of each read from a terminal. 1 Inhibit issuance of a carriage return and line feed at the completion of each read from a terminal.
12:1	Reporting tape device automatic error recovery. 0 Do not report automatic error recovery on a tape device. 1 Report tape device automatic error recovery by returning CCL to FREAD and FWRITE.
0:12	Reserved for MPE

Operation Notes

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

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

The FSETMODE intrinsic allows access to the MPE global Serial Write Queue and **BLOCKONWRITE** parameter on a file-by-file basis. By enabling the Serial Write Queue on a file, all write requests are guaranteed to be performed in the order issued. This preserves integrity while allowing the performance benefits of uninterrupted

process execution (BLOCKONWRITE=NO). This option may affect total system performance; contact your Hewlett-Packard SE before enabling it. The Serial Write Queue is globally disabled if the :CACHECONTROL BLOCKONWRITE=YES was specified, since this causes all processes on the system to wait for the physical disk I/O to complete.

BLOCKONWRITE management can also be controlled on a file-by-file basis with the FSETMODE intrinsic. If BLOCKONWRITE is locally enabled, the user's application will not be notified of a disk write completion until the physical write operation has completed. Therefore, the BLOCKONWRITE option guarantees commitment of the transaction to disk prior to completion notification to the user's application. The FSETMODE setting of an individual file will be overridden if the :CACHECONTROL BLOCKONWRITE=YES is specified.

A combination of these two options in FSETMODE provides complete integrity and transaction commitment notification with disk caching with minimal performance impact. Many disk writes can be performed with BLOCKONWRITE=NO, Serial Write Queue enable, and guarantee integrity. On the final disk write of a transaction, both localized BLOCKONWRITE=YES and Serial Write Queue enabled can be specified to provide both integrity and total transaction commitment notification.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. An error occurred.

Related Information

Additional Discussion	"Declaring Access-Mode Options" in Chapter 4.
Manuals	<i>MPE Files System Reference Manual</i> (30000-90236) <i>MPE V System Operation and Resource Management Reference Manual</i> (32033-90005). <i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250).
Special Considerations	Split-stack calls permitted.

FSPACE

Moves a record pointer forward or backward on a magnetic tape or disk file. (Intrinsic number 5)

Syntax

IV	IV
FSPACE(<i>filenum</i> , <i>displacement</i>);	

Parameters

filenum integer by value (required)

The file number of the file on which spacing is to be done.

displacement integer by value (required)

The number of logical records for buffered disk 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 32,767, the maximum negative value is -32,768.

If RIO access is used, *displacement* includes both active and inactive records. Attempts to backspace beyond the beginning of the file are ignored by the system. The logical record pointer points to record 0 (the first record), and no error codes are returned.

Operation Notes

The FSPACE intrinsic cannot be used with variable-length record files, message files, or with spooled files on disk. An attempt to use this intrinsic on such files results in CCL, and the logical record pointer is left at its current position.

Condition Codes	CCE	Request granted.
	CCG	Request denied. An end-of-file indicator was encountered during spacing. For disk files, this is the file limit, and the logical record pointer is not changed. For magnetic tape files, this is the end-of-file mark, and the logical record pointer points to the (logical) end-of-file. The magnetic tape 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. An error occurred; for example, the file resides on a device that prohibits spacing. Not allowed with APPEND access.
Related Information	Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
	Special	Split-stack calls permitted.
	Considerations	



FUNLOCK

Dynamically unlocks a file's global resource identification number (RIN) that was locked with the FLOCK intrinsic. (Intrinsic number 16)

Syntax

IV FUNLOCK(<i>filenum</i>);

Parameters

filenum

integer by value (required)

The file number of the file whose global RIN is to be unlocked.

Condition Codes

CCE

Request granted.

CCG

Request denied. The calling process did not lock the file.

CCL

Request denied. The file was not opened with the dynamic locking *aoption* of the FOPEN intrinsic, or the *filenum* parameter is invalid.

Related Information

Manuals

MPE File System Reference Manual(30000-90236).

Special
Consideration

Split-stack calls permitted.

FUPDATE

Updates (writes) a record in a disk file. (Intrinsic number 4)

Syntax

IV	LA	IV
FUPDATE(<i>filenum</i> , <i>buffer</i> , <i>length</i>);		

Parameters	<i>filenum</i>	integer by value (required) The file number of the file to be updated.
	<i>buffer</i>	logical array (required) Passes the record to be written in the update.
	<i>length</i>	integer by value (required) The number of words or bytes to be written to the file. A positive value indicates words; a negative value indicates bytes. For files opened BUF: ■ If <i>length</i> ≤ record size, the length is transferred in words or bytes and remaining portions of the record will be padded with fill characters. ■ If <i>length</i> = 0, no transfer will occur and the record address is overwritten with default fill characters (blanks for ASCII files; null characters for binary files). ■ If <i>length</i> > record size, CCL is returned and no transfer occurs. For files opened NOBUF: ■ If <i>length</i> ≤ block size, the length is transferred in words or bytes and remaining portions of the record will be padded with fill characters. ■ If <i>length</i> > block size, CCL is returned and no transfer occurs.
Condition Codes	CCE	Request granted.
	CCG	Request denied. An end-of-file condition was encountered during updating.
	CCL	Request denied. An error occurred; the file is not residing on disk, or <i>length</i> exceeds the size of the block when multirecord mode is not in effect.

FUPDATE

Operation Notes

This intrinsic affects the logical record (or block for NOBUF files) of the file last referenced by any intrinsic call, except an **FPOINT** call, which affects the record prior to the last record referenced. **FUPDATE** moves the specified information from the stack into this record. The file containing this record must be opened with the update *aoption* specified in the **FOPEN** call and the file cannot have variable-length records. If RIO access is used, the modified record is set to the **ACTIVE** state.

The **FUPDATE** intrinsic is not applicable to message files.

Related Information

Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
Special	Split-stack calls permitted.
Considerations	

FWRITE

Writes a record or portion of a record from the stack to a file on any device. (Intrinsic number 3)

Syntax

IV	LA	IV	LV
FWRITE(<i>filenum</i> , <i>buffer</i> , <i>length</i> , <i>controlcode</i>);			

Parameters	<i>filenum</i>	integer by value (required) The file number of the file to be written on.
	<i>buffer</i>	logical array (required) Passes the record to be written.
	<i>length</i>	integer by value (required) The number of words or bytes to be written to the record. If this value is positive, it indicates words; if negative, bytes. Zero indicates that no transfer occurs. If <i>length</i> is less than or equal to the <i>recsize</i> parameter associated with the record, only the first <i>length</i> (words or bytes) is written. If <i>length</i> is larger than the logical record size and NOBUF (<i>aoption</i> in FOPEN) is not specified, the FWRITE request is refused and CCL is returned. If NOBUF is specified, <i>length</i> cannot exceed the physical record size unless the multirecord <i>aoption</i> is specified also. If the multirecord <i>aoption</i> in FOPEN is specified, the excess words or bytes are written to succeeding physical records and if carriage-control is specified, the actual data transferred is limited to <i>recsize</i> minus one byte.

controlcode logical by value (required)

A carriage-control code, effective if the file is transferred to a line printer or terminal (including a spooled file whose destination is a line printer or a terminal). This parameter is effective only for files opened with carriage-control specified.

The options are:

Value	Meaning
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.

Use any octal code from Table 2-10 to determine space control and print the full record transferred. This results in a maximum of 132 characters per printed line.

If the ***controlcode*** parameter is not 0 or 1, and ***length*** is 0, only the space control is executed and no data is transferred.

Determine whether the carriage-control directive takes effect before printing (prespace movement) or after printing (postspace movement), with the FCONTROL intrinsic.

For spooled files, use FWRITE instead of FCONTROL to set the prespace/postspace control and the auto/no auto page eject control; use control codes %100 through %103 and %400 through %403 for this. If one of the above controls is specified with ***length***=0, no physical I/O occurs.

When a file is opened with CCTL, use the carriage-control codes in one of the following ways:

- As the value of the *controlcode* parameter.
- When *controlcode*=1, as the first byte of the *buffer* array.

The default carriage-control code is postspacing with automatic page eject. This applies to all Hewlett-Packard subsystems except FORTRAN 77 and COBOL, which have prespacing with automatic page eject.

Note

Channel assignments shown in Table 2-10 are the Hewlett-Packard standard defaults.

Table 2-10. Carriage Control Directives

Octal Code (ASCII)	Description of Carriage Action
%2 - %37 (" ")	Single space (with or without automatic page eject)
%40 - %52 (" ")	Single space (with or without automatic page eject)
%53 ("+")	No space, return (next printing at column 1), cannot be used more than once on the HP 2608A/S without losing data
%54 (" ")	Single space (with or without automatic page eject)
%55 ("-")	Triple space (with or without automatic page eject)
%56 - %57 (" ")	Single space (with or without automatic page eject)
%60 ("0")	Double space (with or without automatic page eject)
%61 ("1")	Conditional page eject (form feed) performed by the software; if the printer is not at top-of-form, a page eject is performed. Ignored if: Postspace mode The current request has a transfer count of 0 and the previous request was FOPEN, FCLOSE, or FWRITE specifying a carriage-control directive of %61. Prespace 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 FWRITE specifying a carriage-control of %61.
%62	Skip to one line before top of form (valid for HP 2608S and 2563A printers only)
%63	A conditional page eject form feed is performed by the printer; not at top-of-form, a page eject is performed (valid for HP 2608S and 2563A printers only)
%62 - %77 (" ")	Single space (with or without automatic page eject; for terminals)
%104 - %177 (" ")	Single space (with or without automatic page eject; for terminals)
%2nn	Space nn lines (no automatic page eject); nn is any octal number from 0 through 77
%300 - %313	Select VFC Channel 1 - 12 (HP 2613, 2617, 2618, 2619)
%300 - %317	Select VFC Channel 1 - 16 (HP 2608A/S)
%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

Table 2-10. Carriage Control Directives (continued)

Octal Code (ASCII)	Description of Carriage Action
%306	Skip to next 1/4 page
%307	Skip to next 1/6 page
%310	Skip to bottom of form
%311	User option (HP 2613/17/18/19), skip to one line before bottom of form (HP 2608A/S)
%312	User option (HP 2613/17/18/19), skip to one line before top of form (HP 2608A/S)
%313	User option (HP 2613/17/18/19), skip to top of form (HP 2608A)
%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
%310 - %317	(HP 2607)
%314 - %317	(HP 2613/17/18/19)
%320	No space, no return (next printing physically follows this)
%321-%377 (" ")	Single space (with or without automatic page eject)
%400 or %100	Sets postspace movement option (prints first, then spaces). If previous option was prespace movement, the driver outputs a line with a skip to VFC Channel 3 (automatic page eject in effect) or a one line advance (equivalent to an octal code of %201 without automatic page eject) to clear the buffer
%401 or %101	Sets prespace movement option (spaces first, 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)

Note

If octal codes %55 and %60 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 a code of %55 is replaced by %304. Therefore, the resulting skip can 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. For the HP 2608S and 2563A, if auto-eject and feature mode are in effect, a code of %60 is replaced by two codes of %302, and a code of %55 is replaced by three codes of %302. The resulting skip is double or triple space with auto-eject, respectively.

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

EFFECT ON DATA OUTPUT

Figure 2-9. Carriage Control Data

Operation Notes

If information is written to a fixed-length record and the NOBUF *option* in FOPEN is not specified, any unused portion of the record is padded with binary zeros or ASCII blanks.

When the FWRITE intrinsic is executed, the logical record pointer is set to the record immediately following the record just written, or to 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. However, this is noted in the file label only when the file is closed or when an extent is allocated. If the physical bounds of the file are reached, CCG is returned.

FWRITE

If a magnetic tape is unlabeled (as specified in the **FOPEN** intrinsic, or in the **FILE** command) and the program attempts to write over or beyond the physical or simulated end-of-tape (EOT) marker, the **FWRITE** intrinsic returns CCL. The actual data is written to the tape, and a call to **FCHECK** reveals an end-of-tape file error. After the EOT marker is crossed, all writes to the tape transfer the data successfully but return CCL 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 in the **FILE** command), CCL is not returned when the tape passes the EOT marker. Attempts to write to the tape after an EOT marker cause end-of-volume (EOV) markers to be written. A message is printed on the system console requesting another volume (reel of tape) be mounted.

For message files and circular files, this intrinsic logically appends the record to the end of the file. If a circular 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. If a message file is full and there are no readers, the process waits if there is an **FCONTROL=45** in effect or if this is the first **FWRITE** after an **FOPEN** call. Otherwise, CCG is returned.

FWRITE

Condition Codes	CCE	Request granted.
	CCG	Request denied. The physical bounds of the file prevented further writing.
	CCL	Request denied. An error occurred; <i>length</i> exceeded the size of the record in nonmultirecord mode, the FSETMODE option was enabled to signify recovered tape errors, or the end-of-tape marker was sensed. If the file is being written to a multivolume labeled 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.
Related Information	Manuals	<i>MPE File System Reference Manual</i> (30000-90236). <i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250).
	Special Considerations	Split-stack calls permitted.

FWRTEDIR

Writes a specific record from the stack to a disk file. (Intrinsic number 8)

Syntax

IV	LA	IV	DV
FWRTEDIR(<i>filenum</i> , <i>buffer</i> , <i>length</i> , <i>recnum</i>);			

Parameters

filenum

integer by value (required)

The file number of the file to be written to.

buffer

logical array (required)

Passes the record to be written. This structure should be large enough to hold all of the information to be transferred.

length

integer by value (required)

The number of words or bytes to be written to the file. A positive value indicates words; a negative value indicates bytes.

For files opened BUF:

- If *length* ≤ record size, the *length* is transferred. The remaining portion of the record is padded with fill characters.
- If *length* = 0, no transfer occurs; however the record addressed is overwritten with default fill characters (blanks for ASCII files; null characters for binary files).
- If *length* > record size, CCL is returned and no transfer occurs.

For files opened NOBUF and non-multirecord:

- If *length* ≤ blocksize, *length* words or bytes are transferred and any remaining portion of the block is padded with fill characters.
- If *length* > blocksize, CCL is returned and no transfer occurs.
- For files opened NOBUF and multirecord, *Length* words or bytes are transferred up to the file limit (ignoring all block boundaries) and remaining portions of the last block written to is padded with fill characters.

recnum

double by value (required)

FWRITEDIR

The relative number of the logical record, or block number for NOBUF files, to be written. Zero indicates the first record.

Operation Notes

This intrinsic differs from the **FWRITE** intrinsic, the **FWRITE** intrinsic writes only the record pointed to by the logical record pointer. Use the **FWRITEDIR** intrinsic only for disk files composed of fixed-length or undefined-length records.

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

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. This can result in the creation of dummy records to pad the records between the previous end-of-file and the newly written record. When the new record is in the same extent, these dummy records are filled with binary zeros or with ASCII blanks.

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 program.

Not applicable to message files.

Condition Codes

CCE	Request granted.
CCG	Request denied. The physical end-of-file was encountered.
CCL	Request denied. An error occurred.

Related Information

Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
Special Considerations	Split-stack calls permitted. Not applicable to message files.

FWRITELABEL

Writes a user-defined label onto a disk file or magnetic tape file that is labeled with an ANSI-standard or IBM-standard label. It also overwrites old user labels. (Intrinsic number 20)

Syntax

O-V	IV	LA	IV	IV
FWRITELABEL(<i>filenum</i> , <i>buffer</i> , <i>length</i> , <i>labelid</i>);				

Parameters	<i>filenum</i>	integer by value (required) The file number of the file to be labeled.
	<i>buffer</i>	logical array (required) Passes the label to be written. If the file is a labeled magnetic tape file, this label must be 40 words in length.
	<i>length</i>	integer by value (optional) The number of words or bytes passed by <i>buffer</i> . A positive value indicates words; a negative value indicates bytes. The default is 128 words (25 bytes).
	<i>labelid</i>	integer by value (optional) The number of the label to be written. The first label is zero. This parameter is ignored for labeled tapes; the next sequential tape label is written. The default is zero.

Operation Notes Once a disk file is opened, it is possible to read from or write to user-defined labels regardless of the 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-standard or IBM-standard label.

Condition Codes	CCE	Request granted.
	CCG	Request denied. The calling process attempted to write a label beyond the limit specified in the FOPEN intrinsic when the file was opened.
	CCL	Request denied. An error occurred.

Related Information	Manuals	<i>MPE File System Reference Manual</i> (30000-90236).
	Special Considerations	Split-stack calls permitted.

GENMESSAGE

Provides access to messages in catalogs that were formatted with the MAKECAT utility. (No intrinsic number assigned)

Syntax

```

      I          IV      IV      IV      BA
msglength:=GENMESSAGE(filenum,setnum,msgnum,buffer,
                      IV      IV      LV
                      buffersize,parammask,param1,
                      LV      LV      LV      LV
                      param2,param3,param4,param5,
                      IV          I
                      msgdestination,errornum);
```

Functional Return	<i>msglength</i>	integer (assigned functional return) Returns the length of the message (in bytes).
Parameters	<i>filenum</i>	integer by value (required) The file number that specifies the message catalog.
	<i>setnum</i>	Positive integer by value (required) The message set number within the catalog (>0,<63).
	<i>msgnum</i>	Positive integer by value (required) The message number within the message set (>0,<32767).
	<i>buffer</i>	byte array (optional) Returns the assembled message.
	<i>buffer</i> <i>size</i>	integer by value (optional) The length of <i>buffer</i> . If <i>buffer</i> is specified, the length is in bytes. If <i>buffer</i> is not specified, this parameter indicates the length (in bytes) of the records to be written to the destination file. Default: 72 bytes (For a single-line message, the combined length of the message number, a blank space separator, and message text must not exceed the default buffer size of 72 bytes.)

*paramask***logical by value (optional)**

A mask indicating parameter types for *param1* through *param5*. The bit settings are:

Bits	Value/Meaning
------	---------------

13:3	<i>param5</i> type:
------	---------------------

000	String, terminated by an ASCII null (0)
-----	---

001	integer
-----	---------

010	double by reference
-----	---------------------

011	Ignored
-----	---------

10:3	<i>param4</i> type:
------	---------------------

000	String, terminated by an ASCII null (0)
-----	---

001	integer
-----	---------

010	double by reference
-----	---------------------

011	Ignored
-----	---------

7:3	<i>param3</i> type:
-----	---------------------

000	String, terminated by an ASCII null (0)
-----	---

001	integer
-----	---------

010	double by reference
-----	---------------------

011	Ignored
-----	---------

4:3	<i>param2</i> type:
-----	---------------------

000	String, terminated by an ASCII null (0)
-----	---

001	integer
-----	---------

010	double by reference
-----	---------------------

011	Ignored
-----	---------

GENMESSAGE

Bits	Value/Meaning
------	---------------

1:3	<i>param1</i> type:
-----	---------------------

000	String, terminated by an ASCII null (0)
-----	---

001	integer
-----	---------

010	double by reference
-----	---------------------

011	Ignored
-----	---------

0:1	value:
-----	--------

1	Ignore rest of word and parameters <i>param1</i> through <i>param5</i>
---	---

0	Rest of word, in 3-bit groupings, specifies parameter types for <i>param1</i> through <i>param5</i>
---	---

Default: *param1* through *param5* are ignored.

param1
through
param5

logical by value (optional)

Passes the parameter to be inserted into the message. If *paramask* is 000 (string), pass the byte address of the array containing the string. If *paramask* is 001 (integer), pass an integer (a value from -32,768 through 32,767). If *paramask* is 010 (double by reference), pass the word address of the doubleword identifier that contains the value.

msgdestination **integer by value (optional)**

The file number of the message's destination file (0 = \$STDLIST, file number of the destination file).

Default: \$STDLIST if you do not specify *buffer*, no file if you do specify *buffer*.

errornum **integer (optional)**

Returns one of the following values, indicating the success or failure of the intrinsic call.

Value	Meaning
0	Successful
1	FREADLABEL call on catalog file failed
2	FREAD call on catalog file failed
3	Specified <i>setnum</i> not found in catalog
4	Specified <i>msgnum</i> not found in catalog
6	Message overflowed buffer (if <i>msgdestination</i> specified, message routed correctly)
7	Write to destination file failed
8	Catalog file opened with improper access options
11	<i>filenum</i> was not specified
12	<i>setnum</i> was not specified
13	<i>msgnum</i> was not specified
14	<i>setnum</i> < 0 was specified
15	<i>setnum</i> > 62 was specified
16	<i>msgnum</i> <= 0 was specified
17	<i>buffer</i> <= 0 was specified
18	<i>msgdestination</i> <= 0 was specified



Operation Notes

The calling program opens a message catalog, allowing the insertion parameters supplied by **GENMESSAGE**. Depending on how the *msgdestination* and *buffer* parameters are defined, **GENMESSAGE** then routes the message to its destination. To route the message to a file, specify *msgdestination*. To return the message to a calling program, specify *buffer*. To do both, specify both *msgdestination* and *buffer*.

Open the catalog file as a permanent ASCII file with the NOBUF and MULTI record access options enabled.

GENMESSAGE

Condition Codes

CCE	Request granted.
CCL	Request denied. File system error.
CCG	Request denied. Missing required parameter, invalid parameter, or invalid file number of catalog or destination file.

Related Information

Additional Discussion	“Using GENMESSAGE to Inset Parameters in Messages” in Chapter 4.
Manuals	<i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250)

GETDSEG

Creates or acquires an extra data segment for use by the process. Data segment management (DS) capability is required. (Intrinsic number 130)

Syntax

L I LVGETDSEG(*index*,*length*,*id*);

Parameters	<i>index</i>	logical by reference (required) The index of the extra data segment assigned by MPE . When GETDSEG is called in user mode, <i>index</i> is a logical index of the assigned data segment; if an error is found, <i>index</i> is set to %2000-%2004. When GETDSEG is called in privileged mode, <i>index</i> is the actual data segment table (DST) entry number assigned for the data segment.
	<i>length</i>	integer by reference (required) The user-specified maximum size, in words, of the newly created data segment. If the data segment already exists, <i>length</i> returns the maximum size, in words, of the data segment.
	<i>id</i>	logical by value (required) The identity that declares the extra data segment sharable between other processes in the job/session, or private to the calling process. For a sharable extra data segment, specify <i>id</i> as a nonzero value. If an extra data segment with the same identification already exists, it is made available to the calling process; otherwise, a new extra data segment, sharable within the job/session, is created with <i>id</i>. For a private extra data segment, specify an <i>id</i> of zero.

GETDSEG

Operation Notes

The number of extra data segments that can be requested, and the maximum size allowed these segments, is 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 process. This index number is assigned by MPE and allows the process to reference the extra data segment in later intrinsic calls.

Assign an identity to the extra data segment that either allows other processes in the job or session to share the extra data segment, or that declares it private to the calling process. If the extra data segment is sharable, other processes can obtain its index (through GETDSEG) and use this index to reference the extra data segment. Thus, the index is a local name that identifies the extra data segment throughout any process that obtained the index with the GETDSEG call. The index does not need to be the same value in all processes sharing the extra data segment. However, the identity is a job-wide or session-wide name that permits any process to determine the index of the extra data segment. If GETDSEG is called in user mode, the extra data segment is initially filled with zeros.

When GETDSEG is called in user mode, all subsequent calls to intrinsics that use *index* must be in user mode. When GETDSEG is called in privileged mode, all subsequent calls to intrinsics that use *index* must be in privileged mode.

Condition Codes

CCE	Request granted. A new extra data segment was created.
CCG	Request granted. An extra data segment with this identity exists already.
CCL	Request denied. The following values are returned in <i>index</i> :

Value	Meaning
%2000	An illegal <i>length</i> was specified.
%2001	The process requested more than the maximum allowable number of data segments.
%2002	Sufficient storage was not available for the data segment.
%2003	A stack expansion necessary to satisfy the request could not be done because the stack was frozen or the stack is already at its maximum size (stack expansion is usually not necessary to get an extra data segment).
%2004	There is not enough room in the job definition table to make an entry for the extra data segment.

Related Information

Additional
Discussion

“Creating an Extra Data Segment” in Chapter 3.

GETINFO

Returns user-supplied information that was passed to a process when it was created. (Intrinsic Number 87)

Syntax

0-V	BA	I	I
<i>result</i> :=GETINFO(<i>infostring</i> , <i>infolength</i> , <i>parm</i>);			

Functional Return

result

integer (assigned functional return)

Returns an execution value:

Value	Meaning
-------	---------

0	Successful execution.
---	-----------------------

1	Execution error, one of the following conditions exist:
---	---

- *Infostring* was specified, but *infolength* was not specified.
- *Infolength* was specified, but *infostring* was not specified.
- Either an invalid *infostring* or *infolength* value was specified.
- An invalid *parm* value was specified.

Parameters

infostring

byte array(optional)

Returns an information string (the contents of an array) passed to the calling process by the *info* parameter of the RUN command, or by the item associated with *itemnum* 11 of the CREATEPROCESS intrinsic.

If the information string was not supplied at process creation time, or if it is of length zero, *infostring* is returned unmodified.

If *infostring* is specified, *infolength* must pass the length, in bytes, of *infostring*.

infolength **integer by reference (optional)**

Passes and receives a value.

If *infostring* is specified, *infolength* must pass the length, in bytes, of *infostring*.

When returning from the call, *infolength* contains the smaller of either the length of the information string returned in *infostring* or the original value of *infolength*.

Note

Specify a value at least as large as the length of the information string supplied at process creation time. If *infolength* is too small, the information string is truncated upon return to *infostring*.

Infostring and *infolength* must be specified as a pair.

parm **integer by reference (optional)**

Returns a value passed to the calling process at process creation time by one of the following sources:

- The *parm* parameter of the RUN command.
- The item associated with *itemnum* 2 of the CREATEPROCESS intrinsic.
- The *param* parameter of the CREATE intrinsic.

A zero is returned if none of the above are used to pass a value.

Operation Notes

Returns user-supplied information to the calling process from one of the following sources:

- If the process was created with the RUN command, the contents of the *info* and *parm* parameters can be retrieved.
- If the process was created with the CREATEPROCESS intrinsic, retrieve the contents of the items associated with *itemnums* 11 and 2.
- If the process was created with the CREATE intrinsic, retrieve the contents of the *param* parameter.

Related Information

Intrinsics CREATEPROCESS

Manuals *MPE V Commands Manual*(32033-90006).

GETJCW

Returns the value of the system-defined job control word (JCW) to the calling process. (Intrinsic Number 73)

Syntax

L

JCW:=GETJCW;

Functional Return	<i>JCW</i>	logical (optional)
		Returns the JCW. This word is structured by the calling program through the SETJCW or PUTJCW intrinsics.
Related Information	Additional Discussion	“User-Defined Job Control Words” in Chapter 5.
	Intrinsics	FINDJCW, PUTJCW, SETJCW
	Commands	SETJCW, SHOWJCW

GETLOCRIN

Acquires local resource identification numbers (RINs) for a job/session. (Intrinsic Number 30)

Syntax

LV

GETLOCRIN(*rincount*);

Parameters***rincount****logical by value (required)*

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

Operation Notes

All local RINs required by a job/session must be acquired in a single call to GETLOCRIN. If more local RINs are required, all local RINs must be released with the FREELOCRIN intrinsic before GETLOCRIN can be called again.

Condition Codes

CCE

Request granted.

CCG

Request denied. RINs already are allocated to this job. Additional RINs cannot be allocated until these RINs are released.

CCL

Request denied. Not enough RINs are available to satisfy this call. None are allocated to this job.

Related Information

Additional
Discussion

“Resource Management” in Chapter 3.

GETORIGIN

Returns the source of the activation call for the calling process that has been previously suspended and subsequently reactivated. The source of the activation request can be the parent process, a child process, or another source (for example, an interrupt or the timer). Process handling (PH) capability is required. (Intrinsic Number 105)

Syntax

I

`source:=GETORIGIN;`

Functional Return

source integer by reference (optional)
Returns one of the following values:

Value	Meaning
0	Was not activated by either the parent or a child process
1	Activated by the parent process
2	Activated by a child process

Related Information

Additional Discussion “Determining Source of Activation” in Chapter 3.

GETPRIORITY

Changes the priority of a process. Process handling (PH) capability is required. (Intrinsic Number 120)

Syntax

0-V	IV	LV	IV
GETPRIORITY(<i>pin</i> , <i>priorityclass</i> , <i>rank</i>);			

Parameters*pin***integer by value (required)**

A value indicating the process whose priority is to be changed. To specify the calling process, set *pin* to zero. To specify a child process, set *pin* to the process identification number (PIN) of a child process.

*priorityclass***logical by value (required)**

Contains the priority class in which the new process is scheduled (AS, BS, CS, DS, ES), computed in the following manner:

$$(\text{ASCII of first character} * 256) + (\text{ASCII of second character})$$

The integer equivalents are:

AS = 16,723
 BS = 16,979
 CS = 17,235
 DS = 17,491
 ES = 17,747

Aborts the calling process if *priorityclass* exceeds the maximum allowable priority class of the rescheduled process or specifies an invalid priority class.

Specify any priority class except AS, if in user mode; this function is limited by the maximum priority assigned to the account by the system manager. If in privileged mode, processes can be scheduled into all subqueues including AS. Processes in the linear queues, AS or BS, do not give up the CPU voluntarily; it could loop indefinitely and prevent other processes from accessing the CPU.

GETPRIORITY

If in privileged mode, specify the *priorityclass* parameter as an absolute priority by *xA*, where *x* is an 8-bit priority number and *A* is the ASCII character *A*. For example, to request a priority class of 31 in the master queue, set *priorityclass* to %017501. An absolute priority must be specified in order to overcome the MAXPRI setting of an account.

Default: The priority class of the calling process

Note



Scheduling a process into the AS or BS priority class can result in the rescheduled process deadlocking the system or locking out system and user processes from execution.

rank

integer by value (optional)

Used for backward-compatibility with pre-MPE IV operating systems only.

Operation Notes

A process must be running in privileged mode to specify absolute priority.

A process can change its own priority or that of its child, but it cannot reschedule its parent.

Split-stack calls permitted.

Note



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 *priorityclass*.

Condition Codes

CCE	Request granted.
CCG	Request denied. The process specified does not exist.
CCL	Request denied. An illegal PIN was specified.

Related Information

Additional Discussion	“Rescheduling Processes” in Chapter 3.
Intrinsics	None
Commands	None
Manuals	None

GETPRIVMODE

Dynamically enters privileged mode. Privileged mode (PM) capability is required. (Intrinsic Number 200)

Syntax

O-P
GETPRIVMODE;

Operation Notes**Caution**

The normal checks and limitations that apply to the standard users in MPE are bypassed in privileged mode (PM). It is possible for a PM 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 PM code. This service, which is not provided under the standard service contract, is available on a time and materials billing basis. Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.

Condition Codes

CCE	Request granted. The program was in nonprivileged mode at the time of the intrinsic call.
CCG	Request granted. The program was already in privileged mode at the time of the intrinsic call.
CCL	Not returned.

Related Information

Additional Discussion	"Entering Privileged Mode" in Chapter 3.
-----------------------	--

GETPROCID

Returns the process identification number (PIN) of a child process. Process handling (PH) capability is required. (Intrinsic Number 112)

Syntax

IIV
pin:=GETPROCID(*numchild*);

Functional Return	<i>pin</i>	integer (assigned functional return) Returns the PIN of the specified child process.														
Parameters	<i>numchild</i>	integer by value (required) A number indicating which child's PIN GETPROCID returns. The value of <i>numchild</i> corresponds to the order of creation of the current children of the calling process, as follows: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td><=1</td><td>PIN of the oldest child process</td></tr><tr><td>2</td><td>PIN of second oldest child process</td></tr><tr><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td></tr><tr><td>.</td><td>.</td></tr><tr><td><i>n</i></td><td>PIN of the most recently created child process (where <i>n</i> is the number of children currently in existence)</td></tr></table>	Value	Meaning	<=1	PIN of the oldest child process	2	PIN of second oldest child process	.	.	.	.	.	.	<i>n</i>	PIN of the most recently created child process (where <i>n</i> is the number of children currently in existence)
Value	Meaning															
<=1	PIN of the oldest child process															
2	PIN of second oldest child process															
.	.															
.	.															
.	.															
<i>n</i>	PIN of the most recently created child process (where <i>n</i> is the number of children currently in existence)															
Related Information	Additional Discussion	"Determining Son Process" in Chapter 3.														

GETPROCINFO

Returns status information about the parent or a child process.
Process handling (PH) capability is required. (Intrinsic Number 110)

Syntax

D	IV
<i>processinfo</i> :=GETPROCINFO(<i>pin</i>);	

Functional Return

processinfo

double (optional)

Returns the process management information about a parent or child process:

Word 1:
Bits (8:8) - The priority number of the process in the master queue

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

Word 2:
Bit (15:1)-Activity state.\ This bit has the following settings:

=0 The process is suspended.

=1 The process is active.

Bits (13:2) - The following settings apply only if bit (15:1)=0:
Suspension condition.

=00 Is not used.

=01 The source of the expected activation is the father.

=10 The source of the expected activation is the son.

=11 The source of the expected activation is either the father or son.

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

Bits (7:2) - Origin of the last **ACTIVATE** intrinsic all denoted by the following bit settings:

=00 The process was activated by MPE.

=01 The process was activated by the father.

GETPROCINFO

=10	The process was activated by the son.
=11	Is not used.
Bits (4:3) - Queue characteristics.	The characteristics are defined by the following bit settings:
=001	DS or ES priority class
=010	CS priority class
=100	Linearly scheduled (AS, 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 PIN of the process that the returned message pertains to. If requesting a parent process, set *pin* to zero. If requesting a child process, set *pin* to the process identification number (PIN) of that process.

Operation Notes Split-stack calls permitted.

Condition Codes	CCE	Request granted.
	CCG	Request denied. The process is being terminated.
	CCL	Request denied. An illegal PIN was specified.

Related Information Additional Discussion “Determining Process Priority and State” in Chapter 3.

GETUSERMODE

Dynamically returns a program to nonprivileged mode. Privileged mode (PM) capability required. (Intrinsic Number 201)

Syntax

`GETUSERMODE;`

Condition Codes

CCE

Request granted. The program was in privileged mode at the time of the intrinsic call.

CCG

Request granted. The program was in nonprivileged mode at the time of the intrinsic call.

CCL

Not returned.

Related Information

Additional
Discussion

“Entering Non-Privileged Mode” in Chapter 3.



HPACDINFO

Lists security information from the access control definition (ACD) of a specified file or device. (No intrinsic number assigned)

Syntax

```
      0-V      D      IV      *  
HPACDINFO(status,itemnum,item  
          IV      *  
          [,itemnum,item] [, ... ] );
```

Note



A maximum of 5 *itemnum/item* pairs can be specified. The last specified *itemnum/item* pair takes precedence over all identical *itemnum/item* pairs specified. Any conflict will produce a warning message.

Parameters

status

double by reference (required) Returns the processing status of the HPACDINFO call. If no errors or warnings are encountered, *status* returns 32 bits of zero. If error or warnings are encountered *status* is interpreted as:

Bits	Value/Meaning
------	---------------

(0:16)	comprise <i>status info</i> . A negative value indicates an error condition and a positive value indicates a warning condition.
--------	---

(16:32)	comprise <i>status.subsys</i> . The value represented by these bits define the subsystem that sets the status information.
---------	--

Refer to **Table 2-14** for ACD status returns.

itemnum

integer by value (required)

Passes the number indicating the *item*, refer to Table 2-11.

item

Type varies (required)

Returns the variable information as specified in *itemnum*, refer to Table 2-11.

In Table 2-12, *itemnum*

1 - 6	Are defined for a file or device
20 - 22	Are defined for non-user dependent security information
30 - 35	Are defined for user-dependent security information

Table 2-11. HPACDINFO Itemnum/Item Values

Itemnum	Mnemonic	Item Description
The following constants have been defined for a file or device request:		
1	BA	Specifies the target file name. The <i>item</i> value specified must be a byte array of 1 to 35 characters, word aligned. Note, no wild card characters are allowed, a lockword can be included, the file name specified does not need to be fully qualified, and a filename < 35 characters must be terminated with a null character or a carriage return.
4	I	Specifies the file identifier. The <i>item</i> value specified must be a 16-bit integer value.
6	I	Specifies the device identifier (ldev). The <i>item</i> value specified must be a 16-bit integer value.
Security information that is not user-dependent can be requested using the following <i>itemnum/item</i> pairs:		
20	A	Returns a unique identifier representing the last version of the ACD. The <i>item</i> value returned is a 4-word, 16-bit integer, where: 1 Version Release Number 2 Date (MPE Format) 3 Time Stamp (H:M:S)
21	I	Returns the number of ACD entries located in the target file ACD. The <i>item</i> value returned is a 16-bit integer.
22	BA	Returns a unique identifier representing the first user specified in the ACD. The <i>item</i> value returned is an 18-character byte array.

Table 2-11. HPACDINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
Security information that is user-dependent can be requested using the following <i>itemnum/item</i> pairs:		
30	BA	Specifies the user whose ACD information is to be returned. The <i>item</i> value specified must be an 18-character byte array, word aligned. If it is <18 characters, it must be terminated with a null character or a carriage return. This entry is required for subsequent user-dependent requests.
31	BA	Returns ACD modes in ASCII format. The <i>item</i> value returned is a 25-character byte array.
32	I	Returns ACD modes as a bit mask. The <i>item</i> value returned is a 16-bit integer whose bits have the following meaning: r (0:1) - READ w (1:1) - WRITE x (2:1) - EXECUTE a (3:1) - APPEND l (4:1) - LOCK z (8:1) - Permission to read n (15:1) - No access
33	BA	Returns ACD modes in ASCII format as access is granted the particular user. The <i>item</i> value returned is a 25-character byte array. All wild cards will be matched from the user specification.
34	I	Returns ACD modes as a bit mask of all accesses granted the particular user. The <i>item</i> value returned is a 16-bit integer whose bits have the following meanings: r (0:1) - READ w (1:1) - WRITE x (2:1) - EXECUTE a (3:1) - APPEND l (4:1) - LOCK z (8:1) - Permission to read n (15:1) - No access All wild cards will be matched from the user specification.
35	BA	Returns the next user in the ACD. The <i>item</i> value returned is an 18-character byte array.

Operation Notes

The HPACDINFO intrinsic lists the requested security attributes of a specified file or device. Three types of *itemnum/item* pair descriptors can be used to define supplied input or to specify returned ACD information. Such descriptors identify the file or device and request non-user-dependent and user-dependent data.

Related Information

Manuals

MPE File System Reference Manual (30000-90236).

HPACDPUT

Manipulates security information in the Access Control Definition (ACD) of a specified file or device. (No intrinsic number assigned)

Syntax

```

      0-V      D      IV      *
HPACDDPUT(status,itemnum1,item1,
          IV      *
          itemnum2,item2);

```

Parameters

status

Double by reference (required)

Returns processing status of the HPACDPUT call. If no errors or warnings are encountered, a *status* returns 32-bits of zero. If errors or warnings are encountered *status* is interpreted as:

Bits	Value/Meaning
(0:16)	comprise <i>status.info</i> . A negative value indicates an error condition and a positive value indicates a warning condition.
(16:32)	comprise <i>status.subsys</i> . The value represented by these bits defines the subsystem that sets the status information.

The values that can be returned in *status* are listed in Table 2-13.

itemnum1

integer by value (required)

An identifier code defining the accompanying paired input or return field (*item1*). The values for *itemnum1* are described in the *item1* parameter description. Refer to Table 2-12.

item1

Type varies by reference (required)

Specifies the list characteristics to be returned from the intrinsic. The File or Device identifier must be specified using one of the following predefined *itemnum1/item1* pairs. Refer to Table 2-12.

itemnum2

Integer by value (required)

The function code defining the accompanying paired input or return field (*item2*). The values for *itemnum2* are described in the *item2* parameter description. Refer to Table 2-12.

item2

Type varies by reference (required)

HPACDPUT

Specifies the list characteristics to be returned from the intrinsic.

Functionality requests can be made using one of the predefined *itemnum2/item2* pairs. Refer to Table 2-12.

In Table 2-13, **itemnum 1 - 6** are for *itemnum1* and **itemnum 20 - 33** are for *itemnum2*.

Table 2-12. HPACDPUT Itemnum/Item Values

Itemnum	Mnemonic	Item Description
1	BA	Specifies the target file name. The <i>item1</i> value specified must be a byte array of 1 to 35 characters, word aligned. No wild card characters are allowed, a lockword can be included, the file name specified does not need to be fully qualified, and a file name less than 35 characters must be terminated with a null character or a carriage return.
4	I	Specifies the file number. The <i>item1</i> value specified must be a 16-bit integer value
5	N/A	Specifies all devices configured on the system. An <i>item1</i> value is not required with the option.
6	I	Specifies the logical device number (ldev). The <i>item1</i> value specified must be a 16-bit integer value.
20	BA	Create a new ACD. The <i>item2</i> value specified must be 1 to 279 character byte array.
21	BA	Add ACD pairs. The <i>item2</i> value specified must be a 1 to 279 character byte array.
22	BA	Replace ACD pairs. The <i>item2</i> value specified must be a 1 to 279 character byte array.
23	BA	Delete ACD pairs. The <i>item2</i> value specified must be a 1 to 279 character byte array.

Table 2-12. HPACDPUT Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
24	N/A	Delete an ACD. An <i>item2</i> value is not required with this option.
25	BA	Create a new ACD. The <i>item2</i> value specified must be a 35-character byte array, word aligned, of the file name containing the ACD specifications.
26	BA	Copy an ACD from a specified file. The <i>item2</i> value specified must be a 35-character byte array, word aligned, containing the file name that the ACD will be copied from.
28	I	Copy an ACD from a previously opened file number. The <i>item2</i> value specified must be a 16-bit integer containing the <i>filenum</i> of the file containing the ACD to be copied.
31	BA	Add ACD pairs. The <i>item2</i> value specified must be a 35-character byte array, word aligned, containing the file name of the ACD which contains the specifications.
32	BA	Replace ACD pairs. The <i>item2</i> value specified must be a 35-character byte array, word aligned, containing the file name of the ACD which contains the specifications.
33	BA	Delete ACD pairs. The <i>item2</i> value specified must be a 35-character byte array, word aligned, containing the file name of the ACD which contains the specifications.

ACD Status Returns The following error codes shown in Table 2-13 are associated with the ACD intrinsics.

Table 2-13. ACD Status Returns

Status Code	Description
-100	Target keyword not in target set.
-101	Function keyword not in function set.
-102	Generic filename not implemented.
-103	User specification must be specified.
-104	Invalid function key.
-105	Incomplete itemnum/item pair.
-250	Bad user specification.
-251	Duplicate access mode
-252	Duplicate permission.
-253	Contradictory modes.
-254	Bad mode.
-255	Missing open parameter.
-256	Missing close parameter.
-257	Missing Colon.
-258	Extraneous information after ACD specification.
-259	Bad account name.
-260	Embedded AT not allowed.
-261	User must be AT.
-262	Pound sign not allowed.
-263	Question mark not allowed.
-264	Missing mode.
-265	User specification not fully qualified.
-266	Bad user name.
-267	Missing user specification.
-268	Duplicate entry in user specification.
-269	Invalid file number or Ldev.
-270	Invalid target keyword.
-271	Invalid Ldev.
-272	Invalid target Ldev.
-273	Invalid source Ldev.
-274	Invalid device class.
-275	

Table 2-13. ACD Status Returns (continued)

Status Code	Description
-300	Entry does not exist.
-301	Source file ACD does not exist.
-302	Source file ACD corrupted.
-303	Target file ACD already exists.
-304	Target file ACD corrupted.
-305	Target file ACD does not exist.
-306	Source Ldev ACD does not exist.
-307	Source Ldev corrupted.
-308	Target Ldev ACD already exists.
-309	Target Ldev ACD corrupted.
-310	Target Ldev ACD does not exist.
-311	Error while opening ACD indirect file.
-312	Illegal ACD file code.
-313	Illegal ACD file record size.
-314	ACD file not ASCII.
-315	Invalid ACD file record format.
-316	Maximum entries (>20) exceeded.
-317	More than existing entries are to be modified.
-318	Entry already exists.
-319	Incompatible source target.
-320	Source target is the same.
-321	Insufficient capabilities to manipulate ACDs.
-322	Execution failed on all devices.
-323	User not allowed to read ACD.
-324	User not allowed to copy source ACD.
-325	Error opening target file.
-326	Error opening source file.
-400	ACD internal error.
-401	Required parameter missing.
-402	Invalid parameter.
-403	Unexpected step.

Table 2-13. ACD Status Returns (continued)

Status Code	Description
-404	Invalid keyword.
-405	No entries.
-406	Invalid starting entry number.
-407	I/O error in ACD.
-408	Cannot allocate ACD space.
-409	Cannot deallocate ACD space.
-410	Read/write beyond ACD.
-411	Error mounting private volume.
-412	Error dismounting private volume.
-413	I/O error in file label.
-414	Internal error.
-415	Invalid database.
-416	Directory error while updating entry.
-417	Cannot lock ACD.
-418	ACD has not been read.
-419	Incompatible version.
-420	Invalid FFILEINFO call.
-421	FCLOSE error.
-422	File does not exist.
-423	
-500	ACD corrupted.
-501	ACD does not exist.
-502	ACD already exists.
-503	Source ACD corrupted.
-504	Source ACD does not exist.
-505	Target ACD already exists.
-506	Error opening file.

Operation Notes

The HPACDPUT intrinsic allows programmatic adjustment, creation, and deletion of ACDs and their security provisions. The intrinsic is used to create, copy, and delete an ACD and to add, replace, modify, and delete pairs within an ACD.

Related Information

Manuals

MPE File System Reference Manual (30000-90236).

INITUSLF

Initializes a USL file to the empty state. (Intrinsic number 82)

Syntax

I	IV	IA
<code>uslerror:=INITUSLF(<i>uslfn</i>num,<i>record</i>);</code>		

Functional Return

uslerror

integer (optional)

Returns error information. If no error occurs, no value is returned. If an error occurs (condition code returns CCL), one of the following is returned:

Value	Meaning
-------	---------

- | | |
|---|--|
| 0 | An unexpected end-of-file was encountered when writing to <i>uslfn</i> um. |
| 1 | Unexpected I/O error occurred. |

Parameters

*uslfn*um

integer by value (required)

The file number of the USL file.

record

integer array (required)

Passes and returns buffer, corresponding to the first record of the USL file (record 0), to be initialized to the empty state. Set this 128-word array to all zeros. The intrinsic sets certain values in record 0 before returning to the calling program.

Condition Codes

CCE

Request granted.

CCG

Not returned.

CCL

Request denied. An error number is returned to *uslerror*.

Related Information

Manuals

MPE Segmenter Reference Manual (30000-90011).

IODONTWAIT

Initiates completion operations for an I/O request. (No intrinsic number assigned)

Syntax

I	O-V	IV	LA	I	L
<i>fnum:=IODONTWAIT(filenum,buffer,length,cstation)</i>					

Functional Return*fnum***integer (optional)**

Returns the file number for which the completion occurred. If no completion occurred, zero is returned.

Parameters*filenum***integer by value (required)**

The file number for which there is a pending I/O request. If you specify 0, the IODONTWAIT intrinsic checks for any I/O completion.

*buffer***logical array(optional)**

Passes an 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 was a read request.

*length***integer by reference (optional)**

Returns 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. This parameter is pertinent only if the original I/O request was a read request. The FREAD intrinsic always returns zero as its functional return if you specify NOWAIT I/O. In this case, the actual record length is returned in the *length* parameter of IODONTWAIT.

*cstation***logical by reference (optional)**

Returns the number of the calling station when completed (used for distributed systems).

IODONTWAIT

Operation Notes

The IODONTWAIT intrinsic operates the same as IOWAIT with one exception: if you call IOWAIT and no I/O has completed, then the calling process is suspended until some I/O completes; if you call IODONTWAIT and no I/O has completed, then control is returned to the calling process (CCE is returned and the result of IODONTWAIT is zero).

The application must be running in privileged mode to specify the FOPEN *aoption* NOWAIT I/O for nonmessage files.

Condition Codes

CCE	Request granted. If the functional return is not zero, then I/O completion occurred with no errors. If the functional return is zero, no I/O has completed.
CCG	An end-of-file was encountered.
CCL	Request denied. Normal I/O completion did not occur; there were no I/O requests pending, a parameter error occurred, or an abnormal I/O completion occurred.

Related Information

Additional Discussion	“Using the IOWAIT Intrinsic” in Chapter 4.
Manuals	<i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250).

IOWAIT

Initiates completion operations for an I/O request. (Intrinsic number 22)

Syntax

I	O-V	IV	LA	I	L
<i>fnum:=IOWAIT(<i>filenum</i>,<i>buffer</i>,<i>length</i>,<i>cstation</i>);</i>					

Functional Return*fnum*

integer by reference (optional)

The file number for which the completion occurred. If no completion occurred, zero is returned.

Parameters*filenum*

integer by value (optional)

The file number for which there is a pending I/O request. If you specify 0, the IOWAIT intrinsic waits for the first I/O completion.

buffer

logical array (optional)

Passes an 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 was a read request. This allows for proper recognition of :EOD where applicable.

length

integer by reference (optional)

Returns 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. This parameter is pertinent only if the original I/O request was a read request. The FREAD intrinsic always returns zero as its functional return if you specify NOWAIT I/O. In this case, the actual record length is returned in the *length* parameter of IOWAIT.

cstation

logical by reference (optional)

Returns the number of the calling station when completed (used for distributed systems).

Operation Notes

The IOWAIT intrinsic initiates completion operations for an I/O request. If you opened a file with the NOWAIT I/O mode *aoption* of the FOPEN intrinsic (*aoption* bit (4:1) = 1), calls to either the IOWAIT or IODONTWAIT intrinsic must follow all read and write requests. This intrinsic initiates completion operations for the associated I/O request, including data transfer into your buffer area if necessary.

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

Note

The application must be running in privileged mode to specify the FOPEN *aoption* NOWAIT I/O for nonmessage files.

Condition Codes

CCE	Request granted. I/O completion occurred with no errors.
CCG	An end-of-file 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.

Related Information

Additional Discussion	“Using the IOWAIT Intrinsic” in Chapter 4.
Manuals	<i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250).

JOBINFO

Provides access to job and session information. (Intrinsic number 180)

Syntax

```

O-V      IV      D      LA
JOBINFO(jsind,jsnum,jsstatus)
          IV      LA      I
          [,itemnum,item,itemerror] [ ... ];

```

Note

You can specify up to five *itemnum/item/itemerror* triples.

Parameters*jsind***integer by value (required)**

Indicates whether a session or job:

Value	Meaning
1	Session
2	Job

*jsnum***double by reference (required)**

Passes the job/session number for which information is requested. Job/session numbers can be displayed by the SHOWME and SHOWJOB commands. If this value is zero, JOBINFO returns information about your logon job/session (unless the first *itemnum* has a value of 1; see below). When you pass a 0 for *jsnum*, JOBINFO returns the actual job/session number in *jsnum*.

*jsstatus***logical array (required)**

Returns indicator of the success or failure of the call in a two-element array. The first element contains the error indicator. The second element should be ignored.

Value	Meaning
0	Successful call. All <i>itemerrors</i> equal zero.
1	Partially successful call. One or more <i>itemerrors</i> returned with nonzero values.
2	Unsuccessful call. All <i>itemerrors</i> returned with nonzero values.
3	Unsuccessful call. Syntax error in calling sequence.
4	Unsuccessful call. Unable to retrieve <i>jsnum</i> .
5	Process terminated. The process terminated during the start of retrieval.

*itemnum***integer by value (optional)**

Specifies which item value is to be returned. (Table 2-15)

*item***logical array (optional)**

A reference parameter (whose data type corresponds to the data type for the desired information) to which the desired information is returned.

All *itemnum* and *item* parameters are output parameters. The only exception is *itemnum*=1, when all the following conditions are met:

- The caller passes the first *itemnum,item,itemerror* triple.
- The first *itemnum*=1, and *item* contains a character string of the form *[jsname,]username.acctname*.
- The *jsnum* parameter is set to 0, and the *jsind* parameter is set to 1 or 2.

If all these conditions are met, JOBINFO returns information about the specified job/session in all other items and returns that job/session number in *jsnum*. The size of the first item is determined by the way it is used.

itemerror **integer by reference (optional)**

Returns the success or failure of the retrieval of each item. The returned values are:

Value	Meaning
0	Successful
1	Invalid <i>itemnum</i>
2	Information given not valid for specified <i>jsnum</i>
3	Insufficient capability
4	Information not available

Table 2-14. JOBINFO Itemnum/Item Values

Itemnum	Mnemonic	Item Description
1	LA	[jsname,]user.account: Can be used as an I/O parameter: ■ If used as an input parameter, a maximum of 26 ASCII characters, plus one for a binary 0 terminator, is allowed. The input string must be in the form [jsname,]user.account. The wildcard character @ is not allowed. ■ If used as an output parameter, the array must be 26 bytes long. The output is left-justified and padded with blanks.
2	LA	Session/job name: An ASCII output parameter. Arrays must be four words long. Output is left-justified and padded with blanks.
3	LA	User name: An ASCII output parameter. Arrays must be four words long. Output is left-justified and padded with blanks.
4	LA	User logon group: An ASCII output parameter. Arrays must be four words long. Output is left-justified and padded with blanks.
5	LA	User account: An ASCII output parameter. Arrays must be four words long. Output is left-justified and padded with blanks.
6	LA	User home group: An ASCII output parameter. Arrays must be four words long. Output is left-justified and padded with blanks.
7	D	Session/job introduction time: Returns a word to be used by the FMTCLOCK intrinsic. If for a scheduled job, it is the time when the job enters the WAIT state.
8	L	Session/job introduction date: Returns a double word to be used by the FMTCALENDAR intrinsic. If for a scheduled job, it is the date when the job enters the WAIT state.
9	LA	Input ldev/class name: An ASCII output parameter. Arrays must be four words long. Output is left-justified and padded with blanks.
10	LA	Output ldev/class name: An ASCII output parameter. Arrays must be four words long. Output is left-justified and padded with blanks.

Table 2-14. JOBINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
11	LA	Current job step: Returns a maximum of 283 ASCII characters and is the image of the command currently executing. The array must be long enough for the expected command image (the length can be obtained from <i>itemnum</i> =34).
12	I	Current number of active jobs
13	I	Current number of active sessions
14	I	Job input priority
15	D	Job/session number
16	I	Jobfence
17	I	Job output priority
18	I	Number of copies
19	I	Job limit (system)
20	I	Session limit (system)
21	I	Job deferred: Returns the values: 0 No 1 Yes
22	L	Main CI PIN for job/session
23	L	Original job spooled: Returns the values: 0 No 1 Yes
24	L	RESTART option: Returns the values: 0 No 1 Yes
25	L	Sequenced job: Returns the values: 0 No 1 Yes

Table 2-14. JOBINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
26	L	Term code: Returns the values: 0 Regular terminal 1 Regular terminal with special logon 2 APL terminal 3 APL terminal
27	L	CPU limit
28	L	Session/job state: Returns the values: 2 Executing 4 Suspending 32 Wait 48 Initialization 56 Scheduled
29	L	User's local attributes
30	I	\$STDIN spoolfile number: Returns data for current jobs and sessions (\$STDIN/\$STDLIST files only). Returns the spoolfile number as an integer.
31	I	\$STDIN spoolfile status: Returns data for current jobs and sessions (\$STDIN/\$STDLIST files only). Returns the values: 0 ACTIVE 1 READY 2 OPEN 3 RESERVED

Table 2-14. JOBINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
32	I	\$STDLIST spoolfile number: Returns data for current jobs and sessions (\$STDIN/\$STDLIST files only). Returns the spoolfile number as an integer.
33	I	\$STDLIST spoolfile status: Returns data for current jobs and sessions (\$STDIN/\$STDLIST files only). Returns the values: 0 ACTIVE 1 READY 2 OPEN 3 RESERVED
34	I	Length of <i>itemnum</i> =11 (current job step)
35	L	SET STDLIST=DELETE invoked: Returns the values: 0 \$STDLIST is saved 1 SET STDLIST=DELETE is invoked
36	L	Job information table data segment number

Operation Notes

System manager (SM) or account manager (AM) capability is required to retrieve information about jobs and sessions not logged under your user and account. If there is only AM capability, access is restricted to account session and job information; SM capability allows access to all session and job information.

Related Information

Additional Discussion “Identifying a Job or Session with JOBINFO” in Chapter 5.
Intrinsics JOBINFO

KILL

Deletes a child process of the calling process and all of its descendants. Process handling (PH) capability is required. (Intrinsic number 102)

Syntax

IV
`KILL(pin);`

Parameters

pin

integer by value (required)

The process identification number (PIN) of the child process to be deleted.

Operation Notes

All resources held by the deleted processes are released. Any remaining files opened by the deleted processes are closed and assigned the same disposition they had when opened.

Condition Codes

CCE

Request granted.

CCG

Request granted. The specified process is already being terminated.

CCL

Request denied. An illegal PIN was specified.

Related Information

Additional
Discussion

“Deleting Processes” in Chapter 3.

LOADPROC

Dynamically loads a segmented library (SL) procedure and any external procedures it has referenced. (Intrinsic number 80)

Syntax

I	BA	IV	I
<i>idnum</i> := LOADPROC(<i>procname</i> , <i>library</i> , <i>plabel</i>);			

Functional Return*idnum***integer (optional)**

Returns an identity number required by the UNLOADPROC intrinsic to unload the SL procedure dynamically. If a loader error occurs (condition code returns CCL), the identity number represents a loader error number.

Parameters*procname***byte array (required)**

Passes the name of the procedure to be loaded. The procedure name must be terminated with a blank and be <= 15-characters; excess characters will be truncated without a warning message and any procedure matching the truncated name will be loaded.

*library***integer by value (required)**

A value requesting a library search for the procedure residing in your logon group:

**Value Meaning**

- | | |
|---|---|
| 0 | Search system SL only |
| 1 | Search logon account SL, then system SL |
| 2 | Search logon group SL, logon account SL, then system SL |
| 3 | Search program file's account SL, then system SL |
| 4 | Search program file's group SL, program file's account SL, and then system SL |

*plabel***integer by reference (required)**

Returns the procedure's label. This is the external *plabel*, so that the SPL construct ASSEMBLE (PCAL 0) may be used.

LOADPROC

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. The value returned to <i>idnum</i> is a loader error code.

Related Information

Additional Discussion	“Dynamic Loading or Unloading of Library Procedures” in Section 4.
-----------------------	--

LOCKGLORIN

Locks a global resource identification number (RIN). Multiple RIN (MR) capability is required to lock more than one global RIN simultaneously. (Intrinsic number 34)

Syntax

IV	L	BA
LOCKGLORIN(<i>rinum</i> , <i>lockflag</i> , <i>rinpassword</i>);		

Parameters*rinum***integer by value (required)**

The RIN of the resource to be locked. This is the RIN furnished in the **GETRIN** command.

*lockflag***logical by reference (required)**

Passes and returns conditional or unconditional RIN locking specification through bit (15:1):

Value	Meaning
-------	---------

- | | |
|---|--|
| 0 | Locking takes place only if the RIN is currently unlocked. If the RIN is not unlocked, control returns to the calling process immediately with the condition code CCG. |
| 1 | Locking takes place unconditionally. If the RIN is currently locked, the calling process suspends until the RIN becomes unlocked. |

All other bits are ignored. See the discussion of condition codes for the values returned in ***lockflag***.

*rinpassword***byte array (required)**

Passes 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 nonalphanumeric ASCII character; a blank is recommended.

Condition Codes

The following condition codes are possible if ***lockflag*** bit (15:1)=1:

CCE

Request granted. If the calling process had already locked the RIN, ***lockflag*** bit (15:1) is set to 0. If the RIN is free, ***lockflag*** bit (15:1) is set to 1.

CCG

Not returned.

CCL

Request denied. Invalid RIN was specified. Either ***rinum*** is not a global RIN, or the value is out of bounds for the RIN table.

LOCKGLORIN

The following condition codes are possible if **lockflag** bit (15:1)=0:

CCE	Request granted. If the calling process had already locked the RIN, lockflag bit (15:1) is set to 0. If the RIN is free, lockflag bit (15:1) is set to 1.
CCG	Request denied. The RIN was locked by another job.
CCL	Request denied because of invalid RIN. Either rinum is not a global RIN, or the value is out of bounds for the RIN table.

Operation Notes

The LOCKGLORIN intrinsic can lock any global RIN assigned to a group of users, one job at a time. When this is done, any other jobs that attempt to lock this RIN are suspended.

To use the LOCKGLORIN intrinsic, the RIN number and the RIN password must be known. Multiple RIN (MR) capability is required to lock two or more global RINs simultaneously. If you do not have MR capability and attempt to lock two or more RINs simultaneously, the LOCKGLORIN intrinsic aborts the calling process and prints the following error message:

ONLY ONE RIN CAN BE LOCKED

Lock global RINs using the LOCKGLORIN and FLOCK intrinsics.

Related Information

Additional “Resource Management” in Chapter 3.
Discussion

LOCKLOCIN

Locks a local resource identification number (RIN). (Intrinsic number 32)

Syntax

IV	L
LOCKLOCIN(<i>rinum</i> , <i>lockflag</i>);	

Parameters*rinum***integer by value (required)**

A previously allocated local RIN; valid values range from 1 to the value specified in the *rincount* parameter of the GETLOCIN intrinsic.

*lockflag***logical by reference (required)**

Passes and returns conditional or unconditional RIN locking specification, through bit (15:1).

Value Meaning

- | | |
|---|--|
| 0 | Locking takes place only if the RIN is currently unlocked. If it is not, control returns to the calling process immediately with the condition code CCG. |
| 1 | Locking takes place unconditionally. If the RIN is currently locked, the calling process suspends until the RIN becomes unlocked. |

All other bits are ignored. See the discussion of condition codes for the values returned in *lockflag*.

Operation Notes

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

LOCKLOCRIN

Condition Codes	The following condition codes are possible if <i>lockflag</i> bit (15:1)=1:
CCE	Request granted. If the calling process has already locked the RIN, <i>lockflag</i> bit (15:1) is set to 1. If the RIN is free, <i>lockflag</i> bit (15:1) is set to 0.
CCG	Not returned.
CCL	Request denied. The RIN is invalid; the <i>rinnum</i> was too large, no local RIN was allocated, or <i>rinnum</i> specified a number less than or equal to zero.

The following condition codes are possible if *lockflag* bit (15:1)=0:

CCE	Request granted. If the calling process has already locked the RIN, <i>lockflag</i> bit (15:1) is set to 1. If the RIN is free, <i>lockflag</i> bit (15:1) is set to 0.
CCG	Request denied. The RIN was locked by another process.
CCL	Request denied. The RIN is invalid; the <i>rinnum</i> was too large, no local RIN was allocated, or <i>rinnum</i> specified a number less than or equal to zero.

Related Information	Additional Discussion	“Resource Management” in Chapter 3.
----------------------------	-----------------------	-------------------------------------

LOCRINOWNER

Determines process identification number (PIN) of the process that locked a local resource identification number (RIN). (Intrinsic number 36)

Syntax

I	IV
<i>pin</i> :=LOCRINOWNER(<i>rinnum</i>);	

Functional Return

pin

integer (required)

If the RIN is locked by the parent of the calling process, **LOCRINOWNER** returns 0. If the RIN is locked by any other process in your process structure, **LOCRINOWNER** returns the PIN of that process.

Parameters

rinnum

integer by value (required)

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

Operation Notes

The **LOCRINOWNER** intrinsic determines the PIN of the process that has a particular local RIN locked. After a process has acquired local RINs, other processes in the process structure can lock and unlock these RINs.

Condition Codes

CCE

Request granted.

CCG

Request denied. The local RIN specified by **rinnum** is not currently locked by any process.

CCL

Request denied. The **rinnum** parameter was invalid (that is, **rinnum** was less than or equal to 0, greater than the RIN table size, or greater than the number of local RINs currently allocated to this process structure).

Related Information

Additional Discussion

“Resource Management” in Chapter 3.

LOGINFO

Provides information about an opened user logging file (whole file set). (Available on version G.02.00 or later.) (Intrinsic number 215) User logging (LG) or system supervisor (OP) capability is required.

Syntax

O-V	DV	I	IV	BA
LOGINFO (<i>index</i> , <i>logstatus</i> [, <i>itemnum</i> , <i>item</i>] [...] ;				

Note



You can specify up to four *itemnum/item* pairs.

Parameters

<i>index</i>	double by value (required) Your access number to the user logging system. This is the value returned in the <i>index</i> parameter of the OPENLOG intrinsic.
<i>logstatus</i>	integer by reference (required) Returns one of the following values, indicating the success/failure of the intrinsic call:

Value	Meaning
0	Successful
2	Parameter out of bounds
3	Request to open or write to a logging process that is not running
4	Incorrect index parameter
5	Incorrect ; <i>mode</i> parameter
6	User request denied because logging process is suspended
7	Illegal capability; user logging (LG) or system supervisor (OP) capability required
8	Incorrect password passed to logging intrinsic.
9	Error occurred while writing logging file
12	System is out of disk space; logging cannot proceed
13	No more logging entries
14	Invalid access
15	End-of-file on user logging file
16	Invalid logging identifier
17	Either <i>itemnum</i> or <i>item</i> missing
18	Invalid <i>itemnum</i>

itemnum **integer by value (optional)**
 Specifies which *item* value is to be returned. (Refer to Table 2-15)

item **byte array (optional)**
 Returns the value of the item specified by the corresponding *itemnum*. Refer to Table 2-15.

Table 2-15. LOGINFO Itemnum/Item Values

Itemnum	Mnemonic	Item Description
1	D	Total number of records written in current file
2	D	Current file size
3	D	Current file space left
4	I	Number of users
5	D	Total records written in whole file set
6	BA	Current log file name
7	I	Current log file type: 0 DISK 1 TAPE 2 Sdisc 3 Ctape
8	BA	Previous log file name
9	I	Previous log file type: 0 DISK 1 TAPE 2 Sdisc 3 Ctape
10	L	CHANGELOG allowed (TRUE/FALSE)
11	L	AUTO allowed (TRUE/FALSE)
12	I	Current file sequence number
13	I	Log status: 0 INACTIVE 1 ACTIVE 2 Closelog pending 3 Stop pending

Related Information	Additional Discussion	“User Logging” in Chapter 3.
	Commands	SHOWLOGSTATUS

LOGSTATUS

Provides information about a currently opened user logging file. User logging (LG) or system supervisor (OP) capability is required. (Intrinsic number 214)

Syntax

D	LA	I
LOGSTATUS(<i>index</i> , <i>loginfo</i> , <i>logstatus</i>);		

Parameters

index

double by reference (required)

Your access number to the user logging system. This is the value returned in the *index* parameter of the OPENLOG intrinsic.

loginfo

logical array (required)

Returns the following information:

- The total records written in current logging file (a value in words 0 and 1 of *loginfo*).
- The size, in records, of the current logging file (a value in words 2 and 3 of *loginfo*).
- The space, in records, remaining in the current logging file (a value in words 4 and 5 of *loginfo*).
- The number of users using the logging system (a value in word 6 of *loginfo*).

logstatus

integer by reference (required)

Returns one of the following values, indicating the success/failure of the intrinsic call:

LOGSTATUS

Value	Meaning
0	Successful
1	User requested NOWAIT mode and the logging process is busy
2	Parameter out of bounds
3	Request to open or write to a logging process that is not running
4	Incorrect index parameter
5	Incorrect <i>mode</i> parameter passed to a logging intrinsic
6	User request denied because logging process is suspended
7	Illegal capability; user logging (LG) or system supervisor (OP) capability required
8	Incorrect password passed to logging intrinsic
9	Error occurred while writing logging file
12	System is out of disk space; logging cannot proceed
13	No more logging entries
14	Invalid access
15	End-of-file on user logging file
16	Invalid logging identifier

Related Information

Additional
Discussion

“User Logging” in Chapter 3.

Commands

SHOWLOGSTATUS

Special

Considerations

User logging (LG) or system supervisor (OP) capabilities required.

MAIL

Determines the status of the mailbox used by the calling process's parent or child. Process handling (PH) capability is required.

Syntax

L	IV	I
<i>mailstatus:=MAIL(<i>pin</i>,<i>length</i>);</i>		

Functional Return*mailstatus***logical (optional)**

Returns the status of the mailbox:

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

Parameters*pin***integer by value (required)**

The mailbox to be tested. If *pin* specifies the mailbox of a child process, it must be the process identification number (PIN) of that child. Zero specifies the mailbox of a parent process.

*length***integer by reference (required)**

Returns the length, in words, of any incoming mail in the mailbox.

Operation Notes

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 *length* parameter. This enables the calling process to initialize its stack in preparation for receipt of the message.

MAIL

Condition Codes	CCE	Request granted. The mailbox <i>mailstatus</i> was tested.
	CCG	Request denied. An illegal <i>pin</i> parameter was specified. The value of 3 is returned to the calling process through <i>mailstatus</i> .
	CCL	Not returned by this intrinsic that does not have WRITE access.
Related Information	Additional Discussion	“Interprocessor Communications” in Chapter 3.
	Special Considerations	Processor Handling (PH) capability required.

MYCOMMAND

Parses (delineates and defines) parameters for a user-defined command image. (Intrinsic number 71)

Syntax

```

      I          BA      BA      IV
entrynum:=MYCOMMAND(cmdimage,delimiters,maxparms,
                    I      DA      BA
                    numparms,params,dictionary,
                    BP
                    definition);

```

Functional Return*entrynum***integer (optional)**

The command entry number. If the *dictionary* parameter is not specified, 0 is returned.

Parameters*cmdimage***byte array (required)**

Passes one of two things:

- A command name (expected if you specify the *dictionary* parameter), followed by parameters, followed by a carriage-return character (%15). The command name is delimited by the first nonalphanumeric character and cannot be preceded by any leading blanks. One or more blanks is expected after the command name, preceding the parameter list. The parameters are formatted and referenced in the *params* array and *cmdimage* is converted to uppercase, and the byte array specified by *dictionary* is searched for a name matching the command.
- Command parameters (expected if you do not specify the *dictionary* parameter), followed by a carriage-return character (%15). These parameters are formatted. Leading and trailing blanks are ignored. Lowercase is upshifted.

MYCOMMAND

<i>delimiters</i>	byte array (optional) Passes 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 you omit this parameter, the delimiter array “,=; (carriage return)” is used.
<i>maxparms</i>	integer by value (required) The maximum number of parameters expected in <i>cmdimage</i>.
<i>numparms</i>	integer by reference (required) Returns the number of parameters found in <i>cmdimage</i>.
<i>params</i>	double array (required) Returns an array of <i>maxparms</i> entries that delineates the parameters. Each entry consists of two words. When the intrinsic is executed, the first <i>numparms</i> entries are returned to your process in this array, with the first entry corresponding to the first parameter, the second entry corresponding to the second parameter, and so forth. The parameter fields of <i>cmdimage</i> are delimited by the delimiters specified in <i>delimiters</i>. The byte pointer in the first word of each entry points to the parameter in <i>cmdimage</i>. The string in <i>cmdimage</i> is upshifted. The second word of <i>params</i> contains the delimiter number and parameter information. Each double in the array named by <i>params</i> contains the following information: ■ Word 1: The byte pointer to the first character of the parameter. If the parameter is empty or all blanks, the byte pointer indicates the location of the delimiter.

■ Word 2: Bits that describe the parameter:

Bits	Value/Meaning				
11:5	The zero-relative position of the delimiter in <i>delimiters</i> . For example, if you use the default <i>delimiters</i> array and the current parameter is delimited by a semicolon, this field contains 2.				
10:1	Special characters indicator bit. You can set this bit as follows: <table> <tr> <td>0</td><td>The parameter contains no special characters.</td></tr> <tr> <td>1</td><td>The parameter contains special characters other than those listed in <i>delimiters</i>.</td></tr> </table>	0	The parameter contains no special characters.	1	The parameter contains special characters other than those listed in <i>delimiters</i> .
0	The parameter contains no special characters.				
1	The parameter contains special characters other than those listed in <i>delimiters</i> .				
9:1	Numeric character indicator: <table> <tr> <td>0</td><td>The parameter does not contain numeric characters.</td></tr> <tr> <td>1</td><td>The parameter contains numeric characters.</td></tr> </table>	0	The parameter does not contain numeric characters.	1	The parameter contains numeric characters.
0	The parameter does not contain numeric characters.				
1	The parameter contains numeric characters.				
8:1	Alphabetic characters indicator: <table> <tr> <td>0</td><td>The parameter does not contain alphabetic characters.</td></tr> <tr> <td>1</td><td>The parameter contains alphabetic characters.</td></tr> </table>	0	The parameter does not contain alphabetic characters.	1	The parameter contains alphabetic characters.
0	The parameter does not contain alphabetic characters.				
1	The parameter contains alphabetic characters.				
0:8	The length of the parameter in bytes. This value is zero if you omit the parameter.				

MYCOMMAND

<i>dictionary</i>	byte array (optional) Passes the byte array that is to be searched for the command name in <i>cmdimage</i>. The format must be identical to that of the <i>dictionary</i> parameter in the SEARCH intrinsic. The command, delimited by a blank, is extracted from <i>cmdimage</i>, and the SEARCH intrinsic is called with the command name used as the <i>buffer</i> parameter of SEARCH. If the command name is found in <i>dictionary</i>, the functional return is its entry number. If the command is not found, or if the <i>dictionary</i> parameter is not specified, the functional return is zero. If <i>dictionary</i> is specified, but the command name is not found in <i>dictionary</i>, the parameters specified in <i>cmdimage</i> are not formatted. If this parameter is omitted, the functional return is 0.
<i>definition</i>	byte pointer (optional) Returns the address of the definition portion of the command entry in <i>dictionary</i>.

Operation Notes

Allows extracting and formatting of user-defined command parameters. Allows a request to search for a character array, serving as a command dictionary, for a specified command.

Aborts the calling process if the number of characters in *cmdimage* exceeds 255 characters and no delimiter is present. If the number of characters in *cmdimage* exceeds 255 characters and no delimiter is present, the calling process is aborted and the following error message is printed:

PARSED PARAM OF COMIMAGE > 255 CHARACTERS

Condition Codes

CCE	Request granted. The parameters were formatted, without exception. If <i>dictionary</i> was specified, the functional return is the command entry number.
CCG	Request granted. More parameters were found in <i>cmdimage</i> than were allowed by <i>mazparms</i> . Only the first <i>mazparms</i> of these parameters were formatted in <i>params</i> and returned.
CCL	Request denied. The <i>dictionary</i> parameter was specified, but the command name was not located in the array <i>dictionary</i> . The parameters in <i>cmdimage</i> were not formatted.

Related Information

Additional Discussion	“Formatting Command Parameters” in Chapter 5.
-----------------------	---

OPENLOG

Provides access to the user logging facility. User logging (LG) or system supervisor (OP) capability is required. (Intrinsic Number 210)

Syntax

D	LA	LA	I	I
OPENLOG(<i>index</i> , <i>logid</i> , <i>pass</i> , <i>mode</i> , <i>logstatus</i>);				

Parameters

index

double by reference (required)

Returns the user logging access. The user logging facility uses the value of the *index* parameter to check the validity of subsequent calls to the other user logging intrinsics. The value returned is valid only for the process that made the OPENLOG call.

logid

logical array (required)

Passes the user logging identifier. Contains up to eight alphanumeric characters. Arrays < 8-characters must be terminated with a nonalphanumeric character.

pass

logical array (required)

Passes the password associated with the user logging identifier by the GETLOG command.

mode

integer by reference (required)

Passes a value indicating whether the user logging facility should suspend the process if it cannot complete the request for service immediately:

Value	Meaning
0	WAIT
1	NOWAIT

If it is not possible to log the transaction and *mode* is set to NOWAIT, *logstatus* indicates that it could not complete the request.

OPENLOG

logstatus integer by reference (required)

Returns one of the following values, indicating the success/failure of the call:

Value	Meaning
0	Successful
1	NOWAIT mode requested; user logging process busy
2	Parameter out of bounds
3	Request to open or write to a user logging process not running
4	Incorrect index parameter passed to a logging intrinsic
5	Incorrect mode parameter passed
6	Request denied; user logging process suspended
7	Illegal capability; user logging (LG) or system supervisor (OP) capability required
8	Incorrect password passed
9	Error while writing to the user logging file
10	Invalid DST passed to logging system intrinsic
12	System out of disk space; user logging cannot proceed
13	Out of user logging entries
14	Invalid access to logging file
15	End-of-file on user logging file
16	Invalid user logging identifier

Operation Notes

The user logging facility establishes a logging buffer (LOGBUFF) entry for every active process and increments the user count only at the first OPENLOG call. Other counters keep track of the number of times OPENLOG and CLOSELOG are called for each user; the counter is incremented for every OPENLOG and decremented for every CLOSELOG for each user. This ensures that the entry in LOGBUFF is released only if it is the last CLOSELOG call (that is, call counter = 0). Effective with version G.02.00, the number of users and log entries are independent of the number of times the OPENLOG/CLOSELOG intrinsics are called within an application.

Related Information

Additional
Discussion
Commands

“User Logging” in Chapter 3.

LOG



PAUSE

Suspends the calling process for a specified number of seconds.
(Intrinsic Number 45)

Syntax

R

PAUSE(*interval*) ;

Parameters	<i>interval</i>	real (required) Passes the amount of time, in seconds, that the process pauses. The value of <i>interval</i> must be positive. The maximum time allowed is approximately 2,147,484 seconds (almost 25 days).
-------------------	-----------------	--

Condition Codes	CCE	Request granted.
	CCG	Request denied. Insufficient system table (timer request list) space.
	CCL	Request denied. A negative value was specified for <i>interval</i> .

Related Information	Additional Discussion	“Suspending the Calling Process” in Chapter 5.
----------------------------	-----------------------	--

PRINT

Prints character string on job/session listing device. (Intrinsic Number 65)

Syntax

LA	IV	IV
PRINT(<i>message</i> , <i>length</i> , <i>controlcode</i>);		

Parameters*message*

logical array (required)

Passes the string of ASCII characters to \$STDLIST.

Note

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

length

integer by value (required)

The length of the character string to be passed in *message*. If *length* is positive, the length is in words; if negative, the length is in bytes. If *length* exceeds the configured record length of the device, successive records are written only on terminals.

controlcode

integer by value (required)

A carriage-control code (refer to descriptions on carriage control directives in **FWRITE**).

Operation Notes

The **PRINT** intrinsic is limited in its usefulness; the full capability of the file system is not available. For example, the **FILE** command is not allowed and some file intrinsics cannot be used, because the *filenum* parameter, obtained from the **FOPEN** intrinsic, is not available to **PRINT** intrinsic users.

Condition Codes

CCE

Request granted.

CCG

End-of-data was encountered.

CCL

Request denied. An I/O error occurred. Further error analysis through the **FCHECK** intrinsic is not possible.

Related InformationAdditional
Discussion“Writing Output in the Job/Session List Device” in
Chapter 5.

Intrinsics

None

Commands

None

Manuals

Point-to-Point Workstation I/O Reference
(30000-90250).

PRINTFILEINFO

Prints a file information display on the job/session list device.
(Intrinsic Number 21)

Syntax

```
IV
PRINTFILEINFO(filenum);
```

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);
```

Parameters *filenum* integer by value (required)
The file number of the file to be displayed.

Operation Notes The PRINTFILEINFO intrinsic causes MPE to print a file information display on the standard list device in one of two formats. (Refer to Appendix A)

Related Information

Additional Discussion	“Writing a File System Error-Check Procedure” in Chapter 4.
Manuals	<i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250).

PRINTOP

Prints a character string on the system console. (Intrinsic Number 66)

Syntax

LA	IV	IV
PRINTOP(<i>message,length,controlcode</i>);		

Parameters

message

logical array (required)

Passes the character string to be output. The string is limited to 56 characters. Nonvideo-enhancing escape sequences are stripped out.

length

integer by value (required)

The length of the output string to be transmitted. If *length* is positive, the length is in words; if negative, the length is in bytes.

controlcode

integer by value (required)

One of the following cursor control values:

Value	Meaning
0	Moves the cursor to a new line after the message is printed
%320	Does not move the cursor after the message is printed

Note



If a null character (%000) is embedded in an array to be printed on the system console through PRINTOP, then PRINTOP prints only the portion of the array before the null character, regardless of the *length* specified.

Condition Codes

CCE

Request granted.

CCG

Not returned.

CCL

Request denied. A physical I/O error occurred. Further error analysis through the FCHECK intrinsic is not possible.

Related Information

Additional Discussion

“Writing Output to the System Console” in Chapter 5.

PRINTOPREPLY

Prints a character string on the system console and solicits a reply.(Intrinsic Number 67)

Syntax

I	LA	IV	IV
<i>length:=PRINTOPREPLY(message,length,zero,</i>			
	LA	IV	
<i>reply,maxlength);</i>			

Functional Return

length

integer (optional)

Returns the length of the reply from the System Operator. This length represents a word count if ***maxlength*** is positive or a byte count if ***maxlength*** is negative.

If ***maxlength*** is zero, then this intrinsic does not solicit a reply.

If an error occurs, the value returned is zero.

The ***length*** parameter can be zero, in which case only the standard message prefix is written on the system console . If both ***length*** and ***maxlength*** are zero, then CCL is returned.

Parameters

message

logical array (required)

Passes the characters which are output to the system console. The character string is limited to 50 characters.

length

integer by value (required)

The length of the output string to be transmitted. If ***length*** is positive, the length is in words; if negative, the length is in bytes. Do not specify a length of >50-bytes.

zero

integer by value (required)

Do not specify this parameter, but maintain parameter position with a value of 0.

reply

logical array (required)

Returns the input characters read from the system console.

PRINTOPREPLY

<i>maxlength</i>	integer by value (required)
The maximum length of the message to be read into the <i>reply</i> parameter. If <i>maxlength</i> is positive, it specifies a word count; if negative, it specifies a byte count. Do not specify a length of >31-bytes.	

Condition Codes	CCE	Request granted.
	CCG	Not returned.
	CCL	Request denied. A physical I/O error occurred. Further error analysis through the FCHECK intrinsic is not possible.

Related Information	Additional Discussion	“Writing Output to the System Console and Requesting a Reply” in Chapter 5.
---------------------	-----------------------	---

PROCINFO

Provides access to process information. (Intrinsic Number 111)

Syntax

```

      0-V      I      I      IV
PROCINFO(error1,error2,pin
          IV      BA
          [,itemnum,item] [ ... ] );

```

Note



You can specify up to six *itemnum*,*item* pairs.

Parameters

<i>error1</i>	integer by reference (required) Returns a value indicating the success or failure of the intrinsic call, as described in Table 2-17
<i>error2</i>	integer by reference (required) Returns a value that supplies additional error information in <i>error1</i> .
<i>pin</i>	integer by value (required) The process identification number (PIN) of the process for which information is to be returned in the <i>item</i> parameter. A zero returns information about the calling process. This parameter is not compatible with the <i>pin</i> parameter of the GETPROCINFO intrinsic.
<i>itemnum</i>	integer by value (optional) The item number of an information option, as defined in Table 2-17.
<i>item</i>	byte array (optional) Returns information about the process specified by <i>pin</i> as specified in Table 2-17.

Table 2-16. PROCINFO Error1,Error2 Descriptions

Error1	Description	Error2
0	Successful	0
1	Insufficient capability	Index of offending <i>itemnum</i>
2	Omission of required parameter	-1
3	Required parameter address (other than <i>error1</i>) out of bounds	Not used
4	Illegal array size	Array size passed to PROCINFO
5	Invalid <i>itemnum</i>	Index of offending <i>itemnum</i>
6	Invalid PIN; no information returned	-1
7	Unassigned PIN	-1
8	Unpaired <i>itemnum,item</i> parameters	Index of offending <i>itemnum,item</i> parameters

Note

The process aborts if the ***error1*** parameter address is illegal.

If an error condition is detected while processing an information request, the index of the *itemnum*, where the offending option was located is stored in ***error2***.

Table 2-17. PROCINFO Itemnum/Item Values

Itemnum	Mnemonic	Item Description
1	I	Process identification number (PIN) of calling process
2	I	Process identification number (PIN) of the parent of the process specified by <i>pin</i>. If the calling process is executing in privileged mode, requests for information are honored for any process. Otherwise, requests are honored as follows: 1. Complete information is returned for children of the calling process. 2. Information returned for descendants below children (grandchildren, great-grandchildren, and so on), and processes directly above the calling process, is limited to <i>itemnums</i> 2 through 7 and 10 only. Process handling capability is required for any user mode call unless the calling process is requesting information about itself.
3	I	Number of children of the process specified by <i>pin</i>. If the calling process is executing in privileged mode, requests for information are honored for any process. Otherwise, requests are honored as follows: 1. Complete information is returned for children of the calling process. 2. Information returned for descendants below children (grandchildren, great-grandchildren, and so on), and processes directly above the calling process, is limited to <i>itemnums</i> 2 through 7 and 10 only. Process handling capability is required for any user mode call unless the calling process is requesting information about itself.
4	I	Number of descendants (children, grandchildren, and so on) of the process specified by <i>pin</i>. If the calling process is executing in privileged mode, requests for information are honored for any process. Otherwise, requests are honored as follows: 1. Complete information is returned for children of the calling process. 2. Information returned for descendants below children (grandchildren, great-grandchildren, and so on), and processes directly above the calling process, is limited to <i>itemnums</i> 2 through 7 and 10 only. Process handling capability is required for any user mode call unless the calling process is requesting information about itself.
5	I	Number of generations (number of levels in the process tree substructure), including itself, of the process specified by <i>pin</i>. If the calling process is executing in privileged mode, requests for information are honored for any process. Otherwise, requests are honored as follows: 1. Complete information is returned for children of the calling process. 2. Information returned for descendants below children (grandchildren, great-grandchildren, and so on), and processes directly above the calling process, is limited to <i>itemnums</i> 2 through 7 and 10 only. Process handling capability is required for any user mode call unless the calling process is requesting information about itself.

Table 2-17. PROCINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
6	IA	Process identification numbers (PINs) of all children of the process specified by <i>pin</i>. A variable number of PINs are returned; <i>item</i> must be a 16-bit signed integer array. The first element of the array must be set by the calling process to indicate the array size. The array size should include the array size element (for example, for four PINs, the first entry that contains the array size should be 5). PINs are stored into the array, one PIN per element, starting with the second element and continuing until the array is filled or all PINs are returned. If the array is not filled, the remaining unused elements are padded with zeros. If the calling process is executing in privileged mode, requests for information are honored for any process. Otherwise, requests are honored as follows: 1. Complete information is returned for children of the calling process. 2. Information returned for descendants below children (grandchildren, great-grandchildren, and so on), and processes directly above the calling process, is limited to <i>itemnums</i> 2 through 7 and 10 only. Process handling capability is required for any user mode call unless the calling process is requesting information about itself.
7	IA	Process identification numbers (PINs) of all descendants (children, grandchildren, and so on) of the process specified by <i>pin</i>. A variable number of PINs are returned; <i>item</i> must be a 16-bit integer array. The first element of the array must be set by the calling process to indicate the array size. The array size should include the array size element (for example, for four PINs, the first entry that contains the array size should be 5). PINs are stored into the array, one PIN per element, starting with the second element and continuing until the array is filled or all PINs are returned. If the array is not filled, the remaining unused elements are padded with zeros. If the calling process is executing in privileged mode, requests for information are honored for any process. Otherwise, requests are honored as follows: 1. Complete information is returned for children of the calling process. 2. Information returned for descendants below children (grandchildren, great-grandchildren, and so on), and processes directly above the calling process, is limited to <i>itemnums</i> 2 through 7 and 10 only. Process handling capability is required for any user mode call unless the calling process is requesting information about itself.

Table 2-17. PROCINFO Itemnum/Item Values (continued)

Itemnum	Mnemonic	Item Description
8	I	Priority number in the master queue of the process specified by <i>pin</i> (same as Bits (0:16) of the GETPROCINFO intrinsic, functional return)
9	L	State and activation information of the process specified by <i>pin</i> (same as Bits (16:16) of the GETPROCINFO intrinsic, functional return)
10	BA	Program name that the process specified by <i>pin</i> is currently executing. The byte array for the program name must be a minimum of 28-bytes long. The name is returned in the form <i>file.group.account</i>, where <i>file</i> is the local file name, <i>group</i> is the group name, and <i>account</i> is the account name of the file containing the program that the specified process is currently executing. The name is returned left-justified with the unused locations filled with blanks. If the calling process is executing in privileged mode, requests for information are honored for any process. Otherwise, requests are honored as follows: 1. Complete information is returned for children of the calling process. 2. Information returned for descendants below children (grandchildren, great-grandchildren, and so on), and processes directly above the calling process, is limited to <i>itemnums</i> 2 through 7 and 10 only. Process handling capability is required for any user mode call unless the calling process is requesting information about itself.

Condition Codes	CCE	Request granted. Error1 and error2 return zero.
	CCG	Not returned.
	CCL	Request denied. Error codes set and returned in error1 and error2 .

PROCTIME

Returns the accumulated CPU time for a process. (Intrinsic Number 42)

Syntax

D

time:=PROCTIME;

Functional Return

time

double by reference (optional)
Returns the number of milliseconds that the process has accumulated.

Related Information

Additional Discussion

“Obtaining Process Run Time” in Chapter 5.

Commands

REPORT

PTAPE

Copies input from paper tapes which do not contain X-OFF control characters to a disk file. (Intrinsic Number 191)

Syntax

```

      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 character is encountered, returning control to you at the terminal.

Prior to calling this intrinsic, be sure to position the end-of-file pointer in the disk 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 disk file, the append-only *aoption* and a variable-length record format, before the first PTAPE call. In addition, 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 Control-Y character.

Parameters

filenum1

integer by value (required)

The file number of the user's terminal. This is the value returned by FOPEN when the terminal file was opened.

filenum2

integer by value (required)

The file number of the disk file to which the data is to be written.

PTAPE

Condition Codes

CCE

Request granted.

CCG

Request denied because an error occurred while writing to the specified disk 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 disk space or main memory, are available to satisfy the request.

(PTAPE requires 128 contiguous sectors of work area on *ldev1*.)

Related Information

Additional
Discussion

Appendix B, “Device Characteristics”

PUTJCW

Assigns the value of a particular job control word (JCW) in the job control word table. (Intrinsic Number 85)

Syntax

BA	L	I
PUTJCW(<i>jcwname</i> , <i>jcwvalue</i> , <i>jcwstatus</i>);		

Parameters

jcwname byte array (required)

Passes the name of the JCW. This array can contain up to 255 characters, beginning with a letter and ending with a nonalphanumeric character, such as a blank. An "@" causes all executing JCWs to be set to *jcwvalue*.

jcwvalue logical by reference (required)

Passes the value of the JCW.

jcwstatus integer by reference (required)

Returns the execution status:

Value	Meaning
0	Successful
1	Error, <i>jcwname</i> greater than 255 characters
2	Error, <i>jcwname</i> does not start with an alpha character
3	Error, JCW table overflow (no JCW with this name exists in table and unable to create new entry)
4	Error, <i>jcwname</i> attempted to assign a value to an MPE-defined JCW (OK, WARN, FATAL, or SYSTEM)
5	Error, cannot assign a value to a system-reserved JCW (Value 5 is only returned on versions G.00.00 or later)

Operation Notes

If a JCW 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.

There are three types of JCWs in the system:

- User-defined
- System-defined
- System-reserved

PUTJCW

An attempt to assign a value to a system-reserved JCW results in an error. The system-reserved JCWs are:

- HPDATE
- HPDAY
- HPHOUR
- HPMONTH
- HPMINUTE
- HPYEAR

Related Information

Additional Discussion	“User-Defined Job Control Words” in Chapter 5.
Intrinsics	FINDJCW, GETJCW, SETJCW
Commands	SETJCW, SHOWJCW

QUIT

Aborts the calling process. (Intrinsic Number 76)

Syntax

IV QUIT(<i>num</i>);

Parameters*num***integer by value (required)**

Any integer value in the range 0..16383. When 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.

Operation Notes

Aborts the calling process by:

- Transmitting an abort message to the list device of the calling process.
- Setting the job/session job control word (JCW) to indicate that the program terminated in an error state.
- Deleting the calling process and all of its descendants. All resources held by the deleted processes are released. Any remaining files opened by the deleted processes are closed and assigned the same disposition they had when opened.

In batch jobs, not using the **CONTINUE** command results in job termination.

This intrinsic affects the system job control word (JCW).

Related Information

Additional Discussion “Aborting a Process” in Chapter 5.

Special Considerations Split-stack calls permitted.

QUITPROG

Aborts the entire user process structure. (No intrinsic number assigned)

Syntax

IV
QUITPROG(*num*);

Parameters

num

integer by value (required)

Any integer value in the range 0 ... 16383. When 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.

Operation Notes

Aborts the entire user process structure (all children of the job/session main process) by:

- Transmitting an abort message to the list device of the job/session main process.
- Setting the job/session job control word (JCW) to indicate that the program terminated in an error state.
- Deleting all descendants of the job/session main process (the entire user program structure). All resources held by the deleted processes are released. Any remaining files opened by the deleted processes are closed and assigned the same disposition they had when opened.

In batch jobs, not using the CONTINUE command terminates the job. This intrinsic affects the system job control word (JCW).

Related Information

Additional Discussion “Aborting a Program” in Chapter 5.

Special Considerations Split-stack calls permitted.

READ

Reads an ASCII string from \$STDIN into an array. (Intrinsic Number 64)

Syntax

I

LA

IV

```
length:=READ(message,msglength);
```

Functional Return

length integer (optional)
Returns the length of the ASCII string that was read. If *msglength* is positive, the length specified is in words; if negative, the length specified is in bytes.

Parameters

message logical array (required)
Returns the ASCII characters that were read.

msglength integer by value (required)
The length of *message*. If *msglength* is positive, it specifies the length in words; if negative, it specifies the length in bytes. When the record is read, the first *msglength* characters are input. If *msglength* is >= record size of *message*, the entire record is transmitted.

Note



Passing a value >*message* can cause user data corruption or a CCL bounds violation if the system could be corrupted by the call.

Operation Notes

Reads an ASCII string from \$STDIN into an array. This is similar to calling the FREAD intrinsic against the file \$STDIN. This intrinsic is limited, full capability of the file system is not available. For example, FILE commands are not allowed and some file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not normally available to users of the READ intrinsic.

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

READ

READ interprets the following to be end-of-file (EOF) markers when they begin in the first column of a record:

- For sessions, a colon (:).
- For jobs, a colon (:) or the actual EOF.

When one of these indicators are encountered, CCG is returned and no transfer occurs.

If \$STDIN is redirected, only the file's actual EOF is interpreted as the valid EOF indicator.

Condition Codes

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

Related Information

Additional Discussion	“Transmitting Program I/O from Job/Session I/O Devices” in Chapter 5.
Manuals	<i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250)

READX

Reads an ASCII string from \$STDINX into an array. (No intrinsic number assigned)

Syntax

I	LA	IV	
$length := READX(message, msglength);$			

Functional Return

length

integer (optional)

Returns the length of the ASCII string that was read. If *msglength* is positive, the length specified is in words; if negative, the length specified is in bytes.

Parameters

message

logical array (required)

Returns the ASCII characters that were read.

msglength

integer by value (required)

The length of *message*. If *msglength* is positive, it specifies the length in words; if negative, it specifies the length in bytes. When the record is read, the first *msglength* characters are input. If *msglength* is $\geq$ record size of *message*, the entire record is transmitted.

Note



Passing a value $> message$ can cause user data corruption or a CCL bounds violation if the system could be corrupted by the call.

Operation Notes

The READX intrinsic reads an ASCII string from \$STDINX into an array. This is similar to calling the FREAD intrinsic against the file \$STDINX. However, the READX intrinsic is limited in its usefulness in that the full capability of the file system is not available to you. For example, FILE commands are not allowed. Also, you cannot use certain file intrinsics because the *filenum* parameter, obtained from the FOPEN intrinsic, is not normally available to users of the READX intrinsic.

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

READX

READX interprets :EOD, :EOF:, or in a job, :EOJ, :JOB, or :DATA as an end-of-file indicator.

If \$STDIN is redirected, only the file's actual EOF is interpreted as the valid EOF indicator.

Condition Codes

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

Related Information

Additional Discussion	"Transmitting Program I/O from Job/Session I/O Devices" in Chapter 5.
Manuals	<i>Point-to-Point Workstation I/O Reference Manual</i> (30000-90250)

RECEIVEMAIL

Receives mail from another process. Process handling (PH) capability is required. (Intrinsic Number 108)

Syntax

L	IV	LA	LV
<i>mailstatus:=RECEIVEMAIL(pin,location,waitflag);</i>			

Functional Return*mailstatus***logical (optional)**

Returns one of the following values:

Value	Meaning
0	Mailbox empty and waitflag bit (15:1)=0.
1	No message collected, the mailbox contained outgoing mail from the receiving process.
2	Message collected successfully.
3	Error occurred because an invalid <i>pin</i> was specified or a bounds check failed.
4	Request denied, waitflag%x; 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 both processes were suspended, neither could activate the other, and they may be deadlocked.

Parameters*pin***integer by value (required)**

The process sending the mail. If you specify a child process, *pin* is the process identification number (PIN) of that process. If you specify a parent process, *pin* is zero.

*location***logical array (required)**

The address of the buffer where the message is to be written.

RECEIVEMAIL

waitflag logical by value (required)

The action to be taken if the mailbox is empty. This is determined by the setting of bit (15:1), which has the following settings:

Value	Meaning
0	Return immediately to the calling process.
1	Wait until incoming mail is ready for collection.

Operation Notes The RECEIVEMAIL intrinsic lets a process collect mail transmitted to it by its parent or a child. 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 is currently in use by another process, RECEIVEMAIL waits until the mailbox is free before accessing it.

Condition Codes	CCE	Request granted. The mail was collected (value 2 is returned to <i>mailstatus</i>) or the mail was not collected because the mailbox contained outgoing mail from the receiving process (value 1 is returned to <i>mailstatus</i>).
	CCG	Request denied. An illegal <i>pin</i> parameter (value 3 is returned to <i>mailstatus</i>).
	CCL	Request denied. The <i>location</i> parameter does not define a “legal” stack address (value 3 is returned to <i>mailstatus</i>) or because both sending and receiving processes would be awaiting incoming mail, causing a deadlock (value 4 is returned to <i>mailstatus</i>).

Related Information	Additional Discussion	“Interprocess Communications” in Chapter 3.
	Manuals	<i>DSN/DS Reference Manual</i> (32190-90001).

RESETCONTROL

Allows a process to accept another subsystem break signal. (Intrinsic Number 55)

Syntax

RESETCONTROL;

Operation Notes

The process must have previously invoked a subsystem break trap handler with the XCONTRAP intrinsic. After the subsystem break trap handler is invoked, call RESETCONTROL when ready to receive another subsystem break signal. RESETCONTROL can be called from within the subsystem break trap handler or from any other procedure.

Condition Codes

CCERequest granted.

CCGNot returned.

CCLRequest denied. The trap handling procedure was not invoked.

Related Information

Additional Discussion“Control-Y Traps” in Chapter 5.

ManualsPoint-to-Point Workstation I/O Reference Manual(30000-90250)

RESETDUMP

Disables the abort stack analysis facility. Only the current process is affected. (Intrinsic Number 79)

Syntax

RESETDUMP;

Condition Codes

CCE	Request granted.
CCG	The abort stack analysis facility was disabled prior to the RESETDUMP call and remains disabled.
CCL	Not returned.

Related Information

Intrinsics	SETDUMP
Commands	RESETDUMP
Manuals	MPE Debug/Stack Dump Reference Manual(30000-90012). MPE V Commands Manual(32033-90006).

SEARCH

Searches a specially-formatted array for a specified entry or name. (Intrinsic Number 70)

Syntax

I 0-V BA IV BA BP
entrynum:=SEARCH(buffer,length,dictionary,definition);

Functional Return

entrynum **integer (optional)**
Returns the entry number of the word in dictionary that matches buffer. If the name specified in buffer is not found, a zero is returned.

Parameters

buffer **byte array (required)**
Passes the name for which the search is to be performed.

length **integer by value (required)**
The length, in characters, of the array buffer.

dictionary **byte array (required)**
Passes the specially formatted array that is to be searched for buffer.

If the name specified in buffer is found in dictionary, the functional return is its entry number. If the name specified in buffer is not found in dictionary, the functional return is zero.

definition **byte pointer (optional)**
Returns the address of the command entry definition in dictionary.

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.

Operation Notes

Because a search is linear, the most frequently used names should appear at the beginning of the array to ensure efficient searching.

Related Information

Additional “Searching Arrays” in Chapter 5.
Discussion

SENDMAIL

Sends mail to another process. Process handling (PH) capability is required. (Intrinsic Number 107)

Syntax

L	IV	IV	LA	LV
<i>mailstatus:=SENDMAIL(pin,length,location,waitflag);</i>				

Functional Return

mailstatus;

logical (optional)

Returns one of the following values:

Value	Meaning
0	Mail transmitted successfully and mailbox contained no previous mail.
1	Mail was transmitted successfully and mailbox contained mail sent previously that was overwritten by the new mail, or contained previous incoming/outgoing mail that was cleared.
2	Mail not transmitted successfully; mailbox contained incoming mail to be collected by the sending process (regardless of the waitflag setting).
3	Error occurred; an illegal <i>pin</i> parameter was specified or a bounds check failed.
4	Illegal WAIT request; would produce a deadlock.
5	Request denied; length exceeded the maximum mailbox size allowed by the system.
6	Request denied; storage resources for the mail data segment were not available.

Parameters

pin

integer by value (required)

The process to receive the mail. If you specify a child process, *pin* is the Process Identification Number (PIN) of that process. If a parent process is specified, *pin* is zero.

length

integer by value (required)

The length of the message, in words, transmitted from the stack of the sending process. If 0 is specified, the mailbox is emptied of any incoming or outgoing mail.

location

logical array (required)

The array (buffer) in the stack containing the message.

waitflag

logical by value (required)

SENDMAIL

Indicates the action to be taken if the mailbox contains previously sent mail, as specified in bit (15:1):

Bits	Value/Meaning
0	Cancel (overwrite) any mail sent previously with the current mail.
1	Wait until the receiving process collects the previous mail before sending current mail.

Operation Notes	If the receiving process mailbox contains an uncollected message from the calling process, the action taken depends on the waitflag parameter specified in the SENDMAIL call. If the mailbox currently is being used by another process, SENDMAIL waits until the mailbox is free and then sends the mail.
------------------------	--

Condition Codes	CCE	Request granted. The mail was sent (value 0 or 1 is returned to <i>mailstatus</i>), or the mail was not sent because the mailbox contained incoming mail to be collected by the sending process (value 2 is returned to <i>mailstatus</i>).
	CCG	Request denied. An illegal length parameter (value 5 is returned to <i>mailstatus</i>), an illegal pin parameter was specified (value 3 is returned to <i>mailstatus</i>), or storage for the mail data segment was not available (value 6 is returned to <i>mailstatus</i>).
	CCL	Request denied. The length or location parameter do not define a “legal” stack address (value 3 is returned to <i>mailstatus</i>), or both processes are waiting to send mail (value 4 is returned to <i>mailstatus</i>).

Related Information	Additional Discussion	“Interprocess Communication” in Chapter 3.
----------------------------	-----------------------	--

SETDUMP

Enables the stack analysis facility. (Intrinsic Number 78)

Syntax

LV
SETDUMP(*flags*);

Parameters

flags

logical by value (required)

A logical word whose bits specify the following:

- Bit (15:1)=1 - Specifies a DL to Q initial dump.
- Bit (14:1)=1 - Specifies a Q initial S dump.
- Bit (13:1)=1 - Specifies a Q-63 to S dump. This bit is ignored if bit (14:1)=1.
- Bit (12:1)=1 - Causes an ASCII dump of the octal content along with the octal values.
- Bits (0:12) are reserved for MPE (set to zero).

if bits (12:4)=0, *flags* will specify a display of registers and stack marker trace only.

Condition Codes

CCE

Request granted.

CCG

Request denied. The abort stack analysis facility was already enabled before the SETDUMP call.

CCL

Not returned.

Related Information

Intrinsics

RESETDUMP, STACKDUMP

Commands

SETDUMP

Manuals

MPE Debug/Stack Dump Reference Manual(30000-90012).

SETJCW

Sets bits in the system job control word (JCW). (Intrinsic Number 72)

Syntax

LV SETJCW(<i>jcword</i>);

Parameters

jcword

logical by value (required)

The user-defined bit contents of the system JCW for interprocess communication.

Bits Value/Meaning

0:1 Reserved for MPE

If (0:1)=1, the system outputs the following message when the program terminates, either normally or due to an error:

**PROGRAM TERMINATED IN ERROR STATE
(CIERR 976)**

In batch mode, the job is terminated unless the **CONTINUE** command is used. If JCW is set to exactly %140000 (bits (0:1)=1 and (1:1)=1 only), the **CIERR 976** message is replaced by:

**PROGRAM ABORTED PER USER REQUEST
(CIERR 989)**

1:15 Used for any purpose.

Related InformationAdditional
Discussion

“User-Defined Job Control Words” in Chapter 5.

Intrinsics

FINDJCW, GETJCW, PUTJCW

Commands

SETJCW, SHOWJCW

Manuals

MPE V Commands Reference Manual(32033-90006).

STACKDUMP

Sends a stack trace to **\$STDLIST** or to the file specified in the *filename* parameter. (Intrinsic Number 77)

Syntax

BA	I	L	DA
STACKDUMP(<i>filename</i> , <i>idnumber</i> , <i>flags</i> , <i>selec</i>);			

or (from SPL only):

0-V	BA	I	L	DA
STACKDUMP(<i>filename</i> , <i>idnumber</i> , <i>flags</i> , <i>selec</i>);				

Parameters

filename

byte array (optional)

Contains the filename 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.

An array of characters giving the file name of a new output file to be opened. The name should be terminated by any nonalphanumeric character except a slash (/) or a period (.). The same restrictions for the *filename* parameter in the **FOPEN** intrinsic apply to this parameter.

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

idnumber

integer by reference (optional)

If the intrinsic fails due to a file system error, the file system specific error number of the failure is returned here. Any value passed into the intrinsic through this parameter is ignored. An integer which is displayed in the header of the dump to identify the printout. If this intrinsic is unsuccessful, the value of *idnumber* is the corresponding file system error number.

Default: Identifying integer not displayed.

flags

logical by reference (optional)

STACKDUMP

A logical value used to specify the following options:

Bit (15:1) - Controls ASCII dump.

=0 Provide ASCII dump.

=1 Suppress ASCII dump.

Bit (14:1) - Controls trace back of stack markers.

=0 Display trace back of stack marker.

=1 Do not display trace back of stack marker.

Default: If Bits (14:2)=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* (30000-90012). 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” (for example, 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 (14:1)=0, in which case the trace back of stack markers is displayed.

Condition Codes

CCE

Request granted.

CCG

Request denied. An invalid address for the location of the *filename* parameter was detected. 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 *idnumber*.

Related Information

Intrinsics

DEBUG, SETDUMP

Commands

SETDUMP

Manuals

MPE Debug/Stack Dump Reference Manual(30000-90012).

STARTSESS

Initiates a session on the specified terminal. Programmatic sessions (PS) capability is required. (Intrinsic Number 195)

Syntax

IV	BA	I	D	IA
STARTSESS(<i>ldev</i> , <i>logonstring</i> , <i>jsid</i> , <i>jsnum</i> , <i>jsstatus</i>);				

Parameters

ldev

integer by value (required)

The logical device number of the target terminal (the terminal on which the session is to be created). The target terminal must be a real physical device, free and available to the system, job-accepting, and hardwired. In addition, the target terminal must be configured as **TYPE 16**, and **SUBTYPE 0** or **SUBTYPE 4**.

The target terminal must be set to the configured baud rate, since there is no baud rate sensing by **STARTSESS**.

logonstring

byte array (required)

Passes the logon string required to initiate the session on the terminal specified by *ldev*. The first characters in the string should be the session name, if specified, or the user name if no session name is assigned.

Passwords must be supplied in *logonstring*. (There is no prompting for passwords on the target terminal.) If passwords are required but not supplied in *logonstring*, **STARTSESS** fails.

You must place a carriage-return character (%15) in the element following the last valid character of *logonstring*.

The *logonstring* parameter is a superset of the **HELLO** command syntax and includes a new option, **NOWAIT**. The **NOWAIT** option allows you to override the requirement of a **Return** key entered at the target terminal.

jsid

integer by reference (required)

Returns the value 1, indicating that *jsnum* is a session number.

jsnum **double by reference (required)**

Returns a number uniquely identifying the session that was created on the target terminal.

jsstatus **integer array (required)**

Returns a two-element array containing status information. The second element always returns a zero. The first element returns a zero if no error occurs. If an error occurs, the first element returns one of the following error values:

Value	Meaning
-1479	Invalid time specified in <i>logonstring</i> ; no time limit is used
-1473	RESTART specified; ignored
-1465	OUTCLASS specified; ignored
-1464	INPRI then HIPRI specified; HIPRI used
-1463	INPRI too high, highest valid INPRI used
-1462	INPRI too low, lowest valid INPRI used
-1461	HIPRI then INPRI specified, INPRI used
-1459	Invalid PRI specified in <i>logonstring</i> ; default used
-1458	Bad TERMTYPE specified in <i>logonstring</i> ; default used
-1452	Unknown parameter found in <i>logonstring</i> , ignored
-1451	Ignored delimiter
0	Successful
1411	Logon failed due to IDD overflow
1412	Logon failed due to JMAT overflow
1424	Missing or invalid user name in <i>logonstring</i>
1426	Missing or invalid account name in <i>logonstring</i>

STARTSESS

Value	Meaning
1431	Specified session user lacks interactive access (IA) capability
1436	No such group in the account
1437	No such account in the system
1438	No such user in the account
1439	User lacks a home group, and a group was not specified in <i>logonstring</i>
1444	Password required but not specified in <i>logonstring</i>
7000	The <i>LDEV</i> is out of range
7001	The <i>LDEV</i> must not be virtual
7002	The device specified by <i>LDEV</i> is not a terminal
7003	The <i>LDEV</i> is not free
7004	The <i>LDEV</i> is not job-accepting
7005	The <i>LDEV</i> is in diagnostic mode
7006	The <i>LDEV</i> terminal is down
7007	DOWN pending on <i>LDEV</i>
7008	<i>LDEV</i> specified not a real device
7009	Caller requires programmatic sessions (PS) capability
7010	Semicolon (;) delimiter is missing
7011	The <i>LDEV</i> is not a valid integer
7012	The <i>LDEV</i> cannot be a negative number
7014	Session was aborted before logging on
7015	Logon failed because system is at its session limit
7016	Logon failed because the session's <i>INPRI</i> was <= system <i>JOBFENCE</i>
7017	Logon failed because the system was unable to obtain a PCB entry for the system process
7018	Logon failed because the system was unable to obtain a DST entry for the session's data stack
7019	Logon failed because the system was unable to obtain a DST entry for the session's JIT



Value	Meaning
7020	Logon failed because the system was unable to obtain a DST entry for the session's JDT
7021	Logon failed because the system was unable to obtain a file DST entry
7022	Logon failed because the system was unable to obtain an entry for the session in the JPCNT table
7023	Logon failed because the system was unable to obtain the XDD entry for the output device
7024	Logon failed because the system was unable to obtain the XDD entry for the input device
7025	Unknown error occurred. This happens only in the case of an illegal call from one internal procedure to another internal procedure when passing the error number
7026	Unable to allocate \$STDIN or logon failed because of a failure to open \$STDIN
7027	Unable to allocate \$STDLIST or logon failed because of a failure to open \$STDLIST
7029	Logon failed because the session's account is out of CPU time
7030	Logon failed because the session's account is out of CONNECT time
7032	Logon failed because account or user lacks IA capability
7033	Logon failed because the session's logon group is out of CPU time
7034	Logon failed because the session's logon group is out of CONNECT time
7035	Logon string was not terminated with a carriage-return character (%15)
7036	The LDEV has an invalid type and subtype for a programmatic session
7039	Internal programmatic session error
7040	Unable to allocate \$STDERR, or logon failed because of a failure to open \$STDERR
7042	Group is not specified, and there is no default home group

STARTSESS

Operation Notes

The terminal on which the session is to be created must be available; no other user may be logged on. The target terminal is not speed-sensed, so it must be set at the configured baud rate.

When the session is created, nothing is printed to the terminal until you press the **(Return)** key on the selected terminal (unless you specify **NOWAIT**). To override the need for a **(Return)**, you can enter the **NOWAIT** keyword parameter with the logon string when invoking the **STARTSESS** intrinsic. For example, passing the following logon string to **STARTSESS** starts a session and logon immediately without waiting for a **(Return)**:

MANAGER/password.SYS;NOWAIT

Never specify the **NOWAIT** option in the logon string without first ensuring that the target terminal is turned on and set at the default baud rate. If the target terminal is the system console and **NOWAIT** is specified, then you must have system manager (SM) capability. All other valid terminals can be specified with only programmatic sessions (PS) capability.

Related Information

Special Programmatic Sessions (PS) capability required.
Considerations

SUSPEND

Suspends a process. Process handling (PH) capability is required. (Intrinsic Number 103)

Syntax

```

O-V      LV  IV
SUSPEND(allow,rin);

```

Parameters

allow

logical by value (required)

The anticipated source of the call that later will reactivate the process, as indicated by bits (14:1) and (15:1):

Bits	Value/Meaning
15:1	Parent activation bit:
0	Process does not expect to be activated by its parent
1	Process expects to be activated by its parent
14:1	Child activation bit:
0	Process does not expect to be activated by one of its children
1	Process expects to be activated by one of its children

Note



If the process will be running with only process handling (PH) capability, at least one of these bits must = 1.

If (14:1)=1 and (15:1)=1, either the parent or a child can activate the suspended process.

Bits (0:14) are reserved for MPE .

rin

integer by value (optional)

A resource identification number (RIN). If *rin* is specified, it represents a local RIN that is locked by the process but is released when this process is suspended. This facility can be used to synchronize processes within the same job.

SUSPEND

Operation Notes

When **SUSPEND** is executed, the calling process relinquishes its access to the CPU until reactivated by an **ACTIVATE** intrinsic call. The calling process must specify the anticipated source of the **ACTIVATE** call (its parent or a child process). When the process is reactivated, it begins execution with the instruction immediately following the **SUSPEND** call.

Condition Codes

CCE	Request granted.
CCG	Not returned.
CCL	Request denied. The allow parameter is not valid, the specified RIN is not owned by this process, or the specified RIN was not locked.

Related Information

Additional Discussion	“Suspending Processes” in Chapter 3.
Special Considerations	Split-stack calls permitted. Process Handling (PH) capability required.

SWITCHDB

Switches DB register pointer. (Intrinsic Number 139)

SYNTAX

L	O-P	LV
<i>logindex</i> :=SWITCHDB(<i>index</i>);		

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

logindex

logical (optional)

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)

The logical index of the data segment to which the DB register is to be switched, as obtained through the **GETDSEG** intrinsic call in Privileged Mode. MPE checks the value specified for index to ensure that the process has acquired access to this segment previously. For an extra data segment, index must be a positive, nonzero 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.

Related Information

Additional Discussion

“Moving the DB Pointer” in Chapter 3.

Intrinsics

GETDSEG

Special Considerations

The program calling both **SWITCHDB** and **GETDSEG** which generated the value for index must be running in Privileged Mode. **GETDSEG** requires Data Segment Management (DS) capability.

TERMINATE

Deletes the calling process and all of its descendents (including all extra data segments held by them). Releases all resources held by the process and its descendants are released. All remaining files, opened by the process and its descendants, are closed and assigned the same disposition they had when opened. (Intrinsic Number 60)

Syntax

`TERMINATE;`

Related Information

Additional Discussion	“Deleting Processes” in Chapter 3 and “terminating Processes” in Chapter 5.
Special Considerations	Split-stack calls permitted.

TIMER

Returns system timer information. (Intrinsic Number 40)

Syntax

```
D
count:=TIMER;
```

Functional Return*count***double (optional)**

Returns the actual millisecond count since the midnight preceding the last system coldload.

Operation Notes

The resolution of the system timer is one millisecond; readings taken within a one-millisecond period can be identical. The system timer is reset to zero every 24-days at 12:00 midnight. If using the **TIMER** intrinsic to compute elapsed time, correct for this condition by:

- Subtracting the current count from the previous count.
- If the result is negative, add 2,073,600,000 (number of milliseconds in 24 days).

Related InformationAdditional
Discussion

“Obtaining System Timer Information” in Chapter 5.

Special
Considerations

Split-stack calls permitted.

UNLOADPROC

Dynamically unloads a segmented library (SL) procedure. (Intrinsic Number 81)

Syntax

IV
UNLOADPROC(*procid*);

Parameters

procid

integer by value (required)

The procedure's identity number, which was obtained from the **LOADPROC** call.

Condition Codes

CCE

Request granted.

CCG

Not returned.

CCL

Request denied. An invalid procid was specified.

Related Information

Additional
Discussion

"Dynamic Loading and Unloading of Library Procedures"
in Chapter 5.

UNLOCKGLORIN

The UNLOCKGLORIN intrinsic unlocks a global resource identification number (RIN) that was locked with the LOCKGLORIN intrinsic. (Intrinsic Number 35)

Syntax

IV

UNLOCKGLORIN(*rinum*);

Parameters

rinum

integer by value (required)

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. This RIN was not locked by this process.

CCL

Request denied. The specified RIN was not allocated.

Related InformationAdditional
Discussion

“Resource Management” in Chapter 3.

UNLOCKOCRIN

The **UNLOCKOCRIN** intrinsic unlocks a local resource identification number (RIN) that was locked by the **LOCKOCRIN** intrinsic. (Intrinsic Number 33)

Syntax

IV UNLOCKOCRIN(<i>rinum</i>) ;
--

Parameters

rinum

integer by value (required)

The locked RIN, designated by an integer from 1 to the value specified in the *rincount* parameter of the **GETOCRIN** intrinsic call.

Condition Codes

CCE

Request granted.

CCG

Request denied. The RIN specified is not locked by the calling process.

CCL

Request denied. The specified RIN is not allocated to this process.

Related Information

Additional
Discussion

“Resource Management” in Chapter 3.

WHO

Returns the access mode and attributes of the user calling the intrinsic.
(Intrinsic Number 69)

Syntax

O-V L D D BA

WHO(*mode, capability, localattr, username,*
 BA BA BA L
 groupname, acctname, homename, term);



Parameters

mode

logical by reference (optional)

Returns the current user's access mode. In this word, the bits have the following meanings:

Bits	Value/Meaning
15:1	0 The job/session input file and job/session list file are not interactive.
	1 The job/session input file and job/session list file form an interactive pair. A person responding through the input device can establish a dialog with a program that displays information on the list device.
14:1	0 The job/session input file and job/session list file are not duplicative.
	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.
12:2	00 The user is a system process.
	01 The user is accessing the system through a session.
	10 The user is accessing the system through a job.
	11 Not used.
0:12	Reserved for MPE

*capability;***double by reference (optional)**

Returns the user's file access attributes, user attributes, and capability class attributes if the appropriate bit = 1. The high-order word indicates file access attributes and user attributes; the low-order word indicates possession of capability-class attributes. The significance of these bits is as follows:

Bits	Value/Meaning
0:1	System manager (SM)
1:1	Account manager (AM)
2:1	Account librarian (AL)
3:1	Group librarian (GL)
4:1	Diagnostician (DI)
5:1	System supervisor (OP)
6:1	Volume set creation (CV)
7:1	Volume set usage (UV)
8:1	User logging (LG)
9:2	Reserved for MPE
11:1	Network administrator (NA) (on version G.01.00 or later)
12:1	Node manager (NM) (on version G.01.00 or later)
13:1	Communications system (CS)
14:1	Nonsharable devices (ND)
15:1	Save files (SF)
16:7	Reserved for MPE
23:1	Batch access (BA)
24:1	Interactive access (IA)
25:1	Privileged mode (PM)
26:1	Programmatic sessions (PS) (on version G.01.00 or later)
27:1	Reserved for MPE
28:1	Multiple RINs (MR)
29:1	Reserved for MPE
30:1	Extra data segments (DS)
31:1	Process handling (PH)

<i>localattr</i>	double by reference (optional) Returns the local attributes of the user, as defined by a user with the account manager attribute.
<i>username</i>	byte array (optional) Returns the user's name in an 8-character array.
<i>groupname</i>	byte array (optional) Returns the name of the user's logon group in an 8-character array.
<i>acctname</i>	byte array (optional) Returns the name of the user's logon account in an 8-character array.
<i>homename</i>	byte array (optional) Returns the name of the user's home group in an 8-character array. If a home group is not assigned, this array is filled with blanks.
<i>term</i>	logical by reference (optional) Returns the logical device number of the job/session input device. If this is a spooled (STREAM) batch job, then the logical device number is the virtual device. A virtual device simulates a spooling device. Users cannot own the actual spooling device, so the virtual device allows users access to spooling. Each virtual device is temporarily assigned a logical device number for the duration of the input or output spooling process. Since the same logical device number can be reassigned to another virtual device with different physical characteristics, such as page length, the virtual device is not permanently configured in the system configuration. You can also obtain the logical device number of the virtual device with FFILEINFO .

Related Information

Additional Discussion

“Determining the User's Access Mode and Attributes” in Chapter 5.

Manuals

Point-to-Point Workstation I/O Reference Manual (30000-90250).

WRITELOG

Writes database and subsystem file records to the user logging file. User logging (LG) or system supervisor (OP) capability is required. (Intrinsic Number 211)

Syntax

D LA I I I
<code>WRITELOG(<i>index,data,length,mode,logstatus</i>);</code>

Parameters

index	double by reference (required) Passes your access to the user logging system. This is the value returned in the index parameter of the <code>OPENLOG</code> intrinsic.
data	logical array (required) Passes the information to be logged. A log record contains 128 words of which 119 are available for writing data. The most efficient use of log file space is to structure arrays with lengths in multiples of 119 words.
length	integer by reference (required) Passes the length of the data in data. A positive value indicates words; a negative value indicates bytes. If the length is greater than 119 words or 238 bytes, the information in data is divided into two or more physical log records.
mode	integer by reference (required) Passes a value indicating whether the user logging facility should suspend the process if it cannot complete the request for service immediately. If it is not possible to log the transaction and mode is set to <code>NOWAIT</code> , the <code>WRITELOG</code> intrinsic indicates through logstatus that it could not complete the request.

Value	Meaning
0	WAIT
1	NOWAIT
2	Write the logging buffer to the disk logfile (disk logging) or the disk buffer file (serial logging) at the first opportunity.

logstatus

integer by reference (required)

Returns one of the following values, indicating the success/failure of the call:

Value	Meaning
0	Successful
1	NOWAIT requested, user logging process busy
2	Parameter out of bounds
3	Request to open or write to a logging process that is not running
4	Incorrect index parameter passed
5	Incorrect mode parameter passed
6	User request denied because logging process is suspended
7	Illegal capability. Must have user logging (LG) and system supervisor (OP) capabilities to use a logging intrinsic
8	Incorrect password passed to a logging intrinsic
9	Error occurred while writing
10	Invalid DST passed to a logging system intrinsic
12	System out of disk space; user logging cannot proceed
13	No more logging entries
14	Invalid access
15	End-of-file on user log file
16	Invalid logging identifier

Related Information

Additional
Discussion

“User Logging” in Chapter 4.

Intrinsics

BEGINLOG, ENDLOG

Special
Considerations

User logging (LG) or system supervisor (OP) capabilities required.

XARITRAP

Arms or disarms the user-written arithmetic trap handling procedure.
(Intrinsic Number 50)

Syntax

IV	IV	I	I
<code>XARITRAP(<i>mask</i>,<i>plabel</i>,<i>oldmask</i>,<i>oldplabel</i>);</code>			

Parameters

`mask`

integer by value (required)

A value determining which trap conditions, if enabled, invoke the user-written software trap handler, and which do not.

If a bit is on (=1), the corresponding trap condition becomes armed. If a bit is off (=0), the corresponding trap condition is disarmed. The bits and their associated arithmetic errors are:

Bits	Value/Meaning
15:1	3000 mode floating-point divide by zero
14:1	Integer divide by zero
13:1	3000 mode floating-point underflow
12:1	3000 mode floating-point overflow
11:1	Integer overflow
10:1	3000 mode double precision overflow
9:1	3000 mode double precision underflow
8:1	3000 mode double precision divide by zero
7:1	Decimal overflow
6:1	Invalid ASCII digit
5:1	Invalid decimal digit

	Bits	Value/Meaning
	4:1	Invalid source word count
	3:1	Invalid decimal operand length
	2:1	Decimal divide by zero
	0:2	Reserved for MPE (set to zero)
plabel	integer by value (required)	
	The address of the trap handling procedure. If the value is 0, the user-written arithmetic trap handler is disarmed.	
oldmask	integer by reference (required)	
	Returns the value of the previous <i>mask</i> .	
oldplabel	integer by reference (required)	
	Returns the label of the process's previous arithmetic trap handler. If no label was previously configured, 0 is returned.	

Operation Notes

XARITRAP can arm any combination of arithmetic traps; at any given time, there is only one user-written trap handler for all armed traps.

There is a difference between arming and enabling traps:

- Enabling a trap means that the occurrence of a trap condition is not ignored.
- Arming a trap is required so that, on a trap condition, a user-written routine is invoked and can take appropriate recovery actions.

The following summarizes what can occur when an arithmetic trap condition arises:

- If a trap is both enabled and armed, the user-written trap handler is invoked whenever a trap condition occurs.
- If a trap is enabled but not armed, an error message is output and the process aborts.
- If a trap is disabled, irrespective of whether it is armed, the trap is ignored, and execution of the process continues without any interruption.

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). The code domains are:

PROG	(User program)
GSL	(Group SL)
PSL	(Public SL)
SSL	(System SL, non-MPE segments)
MPSSL	(System SL, MPE segments)

XARITRAP

If, when a trap procedure is being enabled, the code of the caller is in one of the following states:

- Nonprivileged in PROG, GSL, or PSL, *plabel* must be nonprivileged in PROG, GSL, PSL.
- Privileged in PROG, GSL, or PSL, *plabel* may be privilege or nonprivileged in PROG, GSL or PSL.
- Privileged or nonprivileged in SSL, *plabel* may be in any non-MPESSL-segment.

Condition Codes

CCE	Request granted. The desired traps are now armed.
CCG	Request granted. All traps are now disarmed.
CCL	Not returned.

Related Information

Additional Discussion	“Enabling and Disabling Traps” in Chapter 5.
-----------------------	--

XCONTRAP

Arms or disarms a user-written subsystem break (Control-Y) trap handling procedure. (Intrinsic Number 54)

Syntax

IV	I
XCONTRAP(<i>plabel</i> , <i>oldplabel</i>);	

Parameters

plabel	integer by value (required) The external-type label of a user-written subsystem break trap handling procedure. If the value is 0, the subsystem break trap handler is disarmed for the process.
oldplabel	integer by reference (required) Returns the plabel of the process's previous subsystem break trap handler. If no plabel was previously configured, oldplabel returns 0.

Operation Notes

Call **XCONTRAP** to arm a user-written subsystem break trap handling procedure. This trap handler is invoked when an enabled subsystem break signal is received during an interactive session. Calling **XCONTRAP** from a job results in an error. On most terminals, transmit the subsystem break signal by pressing **CTRL Y**. For this reason, subsystem break traps are commonly called **CONTROL-Y** traps.

Only one process in a session can receive a subsystem break trap at any one time. The process that called **XCONTRAP** most recently receives the next subsystem break trap. A process's initial call to **XCONTRAP** enables the subsystem break trap. Once a process has received a subsystem break trap, and invoked the trap handler, it cannot receive another until it calls the **RESETCONTROL** intrinsic to re-enable the subsystem break trap. Only processes running in a session can arm subsystem break traps. The trap handler can be any procedure in the program or in the libraries to which the program is bound. The subsystem break trap handler has no parameters.

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). The code domains are:

PROG	(User program)
GSL	(Group SL)
PSL	(Public SL)
SSL	(System SL, non-MPE segments)
MPESSL	(System SL, MPE segments)

If, when a trap procedure is being enabled, the code of the caller is in one of the following states:

XCONTRAP

- Nonprivileged in PROG, GSL, or PSL, *plabel* must be nonprivileged in PROG, GSL, PSL.
- Privileged in PROG, GSL, or PSL, *plabel* may be privilege or nonprivileged in PROG, GSL or PSL.
- Privileged or nonprivileged in SSL, *plabel* may be in any non-MPESSL-segment.

Condition Codes

CCE	Request granted. Trap armed.
CCG	Request granted. Trap disarmed.
CCL	Request denied. An illegal plabel or XCONTRAP was called from a job.

Related Information

Additional Discussion	“Enabling and Disabling Traps” in Chapter 5.
-----------------------	--

XLIBTRAP

arms or disarms a user-written software library trap handling procedure.
(Intrinsic Number 52)

Syntax

IV I
XLIBTRAP(<i>plabel</i>,<i>oldplabel</i>) ;

Parameters

<i>plabel</i>	integer by value (required) The external-type label of the user-written trap handling procedure. If the value is 0, the trap handler is disarmed.
<i>oldplabel</i>	integer by reference (required) Returns the <i>plabel</i> of the process' previous software library trap handler. If no <i>plabel</i> was previously configured, 0 is returned.

Operation Notes

When a program begins execution, the user-written software library trap handler is disarmed automatically. When armed by the **XLIBTRAP** intrinsic, and subsequently activated by an error, the user-written software library trap handler assumes control.

This trap handler can be armed or disarmed by calling the **XLIBTRAP** intrinsic.

Condition Codes

CCE	Request granted. Trap armed.
CCG	Request granted. Trap disarmed.
CCL	Request denied. An illegal <i>plabel</i> was specified.

Related Information

Additional Discussion	"Enabling and Disabling Traps" in Chapter 5.
-----------------------	--

XSYSTRAP

Arms or disarms a user-written system trap handling procedure. Intrinsic Number 53)

Syntax

IV I
XSYSTRAP(<i>plabel</i> , <i>oldplabel</i>);

Parameters

plabel	integer by value (required) The external-type label of the trap handling procedure. If the value is 0, the user-written software trap handler is disarmed.
oldplabel	integer by reference (required) Returns the plabel of the process's previous software system trap handler. If no plabel was previously configured, 0 is returned.

Operation Notes

When a program begins execution, the user-written system trap handler is disarmed automatically. When armed by the XSYSTRAP intrinsic, and subsequently activated by an error, the user-written system trap handler assumes control.

The user-written trap handler can be armed or disarmed by calling the XSYSTRAP intrinsic.

Condition Codes

CCE	Request granted. Trap armed.
CCG	Request granted. Trap disarmed.
CCL	Request denied. An illegal plabel was specified.

Related Information

Additional Discussion	"Enabling and Disabling Traps" in Chapter 5.
-----------------------	--

ZSIZE

Alters current DB to Z area of the stack. (Intrinsic 136)

Syntax

I	IV
<i>newsize</i> :=ZSIZE(<i>size</i>);	

Functional Return

newsize

integer by reference (optional)

Returns the size, in words, actually granted.

Parameters

size

integer by value (required)

The desired register offset of Z, from DB. This is the number of words from DB to Z. This value must be ≥ 0 .

Operation Notes

ZSIZE alters the size of the current DB to Z area of the stack by adjusting the register offset of the Z address from the DB address (DB to Z).

The ZSIZE intrinsic increases the Z address (stack expands) or decreases the Z address (stack contracts). If the DB to Z area size requested exceeds the maximum size permitted for the DL to Z area, only the maximum size allowed is granted.

All changes to the DB to Z area are made in increments or decrements of 128 words; the size granted may differ from the size requested.

Condition Codes

CCE

Request granted.

CCG

Request granted. The requested size exceeded the maximum limits of the DL to Z area. The maximum limit is granted, and the value is returned in *newsize*.

CCL

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

Related Information

Additional Discussion

“Changing Stack Sizes” in Chapter 5.

Section Divider

3. Optional Capabilities



Optional Capabilities

Some MPE functions can only be performed by users with the optional capabilities mentioned in Chapter 1. This chapter discusses the optional capabilities needed to perform these functions.

Privileged Mode Capability

Users with standard MPE capabilities can only access their own code and data areas in main memory. A user with privileged mode (PM) capability, however, can access all areas of the system and can use all features of the hardware. They can access all system tables, and invoke all system instructions, including those in the privileged central processor instruction set. In short, a user with PM capability can use the computer on the same terms as MPE itself. MPE does not make any distinction between a privileged user and the operating system.

Caution



The normal checks and limitations that apply to 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. Hewlett-Packard will not support, correct, or attend to any modification of the MPE operating system software.

You can use privileged mode capability in the following ways:

- To write permanently privileged programs that are loaded and executed entirely in privileged mode.
- To write temporarily privileged programs that dynamically enter and leave privileged mode during execution, as required.

Permanently Privileged Programs

A code segment is loaded and executed directly in privileged mode when all the following conditions exist:

1. **\$CONTROL PRIVILEGED** is used or any of the segment's procedures have **OPTION PRIVILEGED**.
2. The program is prepared with privileged mode capability and resides in a group and account having privileged mode capability. You enter the appropriate capability-class attribute in the *caplist* parameter of the **:PREP** or **:PREPRUN** command that prepares the program. You must have privileged mode capability in order to enter a privileged capability class attribute and must be in a group and account with privileged mode capabilities. Refer to the *MPE V Commands Reference Manual* (32033-90006) for discussions of the **:PREP** and **:PREPRUN** commands.

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 ON in the appropriate CST entries.

When you add a segment to a Segmented Library (through the **-ADDSL** Segmenter command), the procedures within the segment are checked to determine whether any of them are privileged. If they are, 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 privileged mode capability in your group and account. Refer to the *MPE Segmenter Reference Manual* (30000-90011) 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 privileged mode dynamically. For example, just before a set of privileged instructions is encountered, the program can be switched to 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 instructions) and exits (EXIT instructions) when encountered in either mode:

- In privileged mode, when a PCAL instruction is executed, privileged mode is retained even though the destination code segment may have a nonprivileged CST entry. When an EXIT instruction is encountered, the resulting mode depends on the status word in the stack marker.
- In non-privileged mode, when a PCAL instruction is encountered, the mode is obtained from the CST entry for the destination code segment. When an EXIT instruction occurs, the resulting mode is taken from bit (0:1) of the status in the stack marker. If the entry indicates privileged mode, ((0:1)=1) 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. Refer to the *Machine Instruction Set Reference Manual* (30000-90022) for further discussions of the PCAL and EXIT instructions.

A code segment is loaded and begins execution as a temporarily privileged segment (in non-privileged mode) when three conditions are met:

1. The program is prepared with privileged mode capability, by entering the appropriate capability-class attribute in the *caplist* parameter of the **:PREP** or **:PREPRUN** command. This requires that you have privileged mode capability as a user and must be in a group and account with privileged mode capabilities.
2. **\$CONTROL PRIVILEGED** was not used.
3. None of the segment's procedures have **OPTION PRIVILEGED**.

The **NOPRIV** parameter of the **:PREPRUN** and **:RUN** commands forces all segments in the program to begin execution in user mode. When a temporarily privileged segment is loaded, the CST entry corresponding to that segment has its privileged mode bit (1:1)=0.

If you possess privileged mode capability, you can also call all intrinsics available to users with the data segment management capability (DS), provided that you follow these rules:

- 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 dependent on the space available in the Process Control Block Extension (PCBX) and cannot be calculated.
- When a temporarily privileged process calls a data segment intrinsic while in non-privileged mode, the data segment index returned to the calling process can also 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 returned index 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 non-privileged mode to Privileged Mode. This intrinsic turns on the privileged mode bit in the status register ((0:1)=1), but leaves the privileged mode bit in the Code Segment Table (CST) entry for the executing segment ((1:1)=0). Thus, if additional segments are run as part of the program, they will run in privileged mode unless **GETUSERMODE** is specifically called to return to the non-privileged mode, since the status register not the CST determines a mode change when in privileged mode.

Figure 3-1 contains a program that uses the **GETPRIVMODE** intrinsic to switch to privileged mode. privileged mode is necessary temporarily because the program opens a file with both **NOBUF** and **NOWAIT** *aoptions* specified in the **FOPEN** intrinsic call. privileged mode capability (PM) is required to call this intrinsic, however, caution should be used since your I/O could overwrite other data on the system.

The program in Figure 3-1 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. Statement 00019000 in the program switches the program from non-Privileged to privileged mode before the next statement opens a file with both the **NOBUF** and **NOWAIT** *aoptions* specified.

The statement **CCG(2);** in line 00019000 causes the program to quit if **CCG** (signifying that the program already is running in privileged mode) is returned.

```

PAGE 0001  HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY OUTPUT(0:16):="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  INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,GETPRIVMODE,GETUSERMODE,
00014000 00005 1          IOWAIT,QUIT;
00015000 00005 1  <<END OF DECLARATIONS>>
00016000 00005 1          OUT:=FOPEN(OUTPUT,4,1,,DEV); CCL(1); <<LINEPRTR OUTPT>>
00017000 00015 1          WHILE (I:=I+1)<MAXTRM DO          <<LOOP-SET UP TRMS>>
00018000 00023 1              BEGIN
00019000 00023 2                  GETPRIVMODE; CCG(2);          << NOWAIT FOPEN>>
00020000 00027 2                  FILE:=FOPEN(TNAM,%405,%4404,36,DEV(3));<<INPUT TERM>>
00021000 00042 2                  CCL(3);          <<CHECK FOR ERR>>
00022000 00042 2                  GETUSERMODE; CCG(4);          <<NOWAIT I/O>>
00023000 00046 2                  OPEN(I):=FILE;          <<SAVE FILE NMBS>>
00024000 00051 2                  FWRITE(FILE,PROMPT,1,%320); CCNE(5);<<OUTPUT PRMPT>>
00025000 00061 2                  IOWAIT(FILE); CCNE(6);          <<COMPLETE REQST>>
00026000 00072 2                  FREAD(FILE,BUFR(I*36),-72); CCNE(7);<<INPT DATA>>
00027000 00106 2              END;
00028000 00113 1  WAIT:
00029000 00113 1          FILE:=IOWAIT(0,,LGTH); CCL(8);          <<WAIT FOR 1ST DNE>>
00030000 00125 1          IF > THEN          <<EOF ON TERM READ>>
00031000 00126 1              BEGIN
00032000 00126 2                  FCLOSE(FILE,0,0); CCL(9);          <<TERMINAL FILE>>
00033000 00134 2                  IF (DONE:=DONE+1)>=MAXTRM THEN GO EXIT;<<TERMS CLSD?>>
00034000 00140 2              END
00035000 00140 1          ELSE
00036000 00142 1              BEGIN
00037000 00142 2                  I:=-1;          <<SET BUFFER INDEX>>
00038000 00144 2                  DO I:=I+1          <<INCR BUFR INDX>>
00039000 00144 2                      UNTIL OPEN(I)=FILE OR I=MAXTRM;<<SRCH FOR FILE NO>>
00040000 00154 2                      IF I=MAXTRM THEN QUIT(10);          <<FILE NOT FOUND>>
00041000 00161 2                      FWRITE(OUT,BUFR(I*36),-LGTH,0); <<COPY INPUT TO LP>>
00042000 00171 2                      CCNE(11);          <<CHECK FOR ERR>>
00043000 00174 2                      FWRITE(FILE,PROMPT,1,%320); CCNE(12);<<OUTPT PRMPT>>
00044000 00204 2                      IOWAIT(FILE); CCNE(13);          <<COMPLETE REQUEST>>
00045000 00215 2                      FREAD(FILE,BUFR(I*36),-72); CCNE(14);<<INPT DATA>>
00046000 00231 2              END;
00047000 00231 1              GO TO WAIT;          <<CONTINUE>>
00048000 00232 1  EXIT:END.

```

Figure 3-1. Using the GETPRIVMODE and GETUSERMODE Intrinsics (Program DSINIT)

Entering non-privileged mode

The **GETUSERMODE** intrinsic is used to change a temporarily privileged program from Privileged to non-privileged mode, by changing the privileged mode bit in the status register to OFF ((0:1)=0).

The intrinsic call **GETUSERMODE;**, is illustrated in line 00022000 of Figure 3-1.

Moving the DB Pointer

If you have the data segment management capability (DS) 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.

For 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**, enter the following intrinsic call:

```
SET:=SWITCHDB(INDEX2):
```

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, nonzero integer. To switch back to the stack, this parameter must be zero.

Since a user-mode call to **GETDSEG** generates a value for *index* of the assigned entry or an error code of %2000-%2004, a Privileged Mode call to **GETDSEG** must be made to ensure that the value for *index* is the actual segment entry number for the data that was assigned. The calling process is aborted if the **SWITCHDB** intrinsic is called from a program which is not running in privileged mode.

Scheduling Processes

Every process in the system is assigned a priority. When a process is ready to run, it is placed in the READY list. When the dispatcher runs, it selects the process in memory with the highest priority for execution.

The master queue (see Figure 3-2) 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 access 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 another process in the queue if no process with a higher priority is ready to be dispatched. This time allocation is controlled by the system timer. Processes that are not scheduled in a subqueue are scheduled in the master queue.

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 (OP) or privileged mode (PM) capability can manage priority numbers. The standard subqueues (priority classes) are as follows:

AS	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 (+ rank to a maximum of 99)
BS	A linear subqueue containing processes of high priority. It is accessible to users having <code>MAXPRI=BS</code> . Normally, priority for a BS process is 100. However, by specifying a rank >0 in the <code>GETPRIORITY</code> intrinsic, the process may be set in the master queue at <i>min</i> value of (100 + rank, 149). Scheduling Type: Linear Priority Range: 100 (+ rank to a maximum of 149)

CS	A circular subqueue generally devoted to interactive sessions. A CS process whose CPU time between priority changes exceeds the “average short transaction” time will be lowered in priority, but not below the C Subqueue Priority Limit, called CLIMIT, which may be set by the :TUNE command. Refer to the <i>MPE V Commands Reference Manual</i> (32033-90006) for more information on the :TUNE command. Scheduling Type: Circular Priority Range: CBASE-CLIMIT
DS	A circular subqueue generally devoted to batch jobs. A DS process whose CPU time between priority changes exceeds the background quantum will be lowered in priority, but not below the D Subqueue Priority Limit, or DLIMIT, which may be set by the :TUNE command. Refer to the <i>MPE V Commands Reference Manual</i> (32033-90006) for more information on the :TUNE command. Scheduling Type: Circular Priority Range: DBASE-DLIMIT
ES	A circular subqueue generally used for background processes. An ES process whose CPU time between priority changes exceeds the background quantum, will be lowered in priority, but not below ELIMIT. Such a process will have a minimal impact on the performance of processes in other subqueues. Scheduling Type: Circular Priority Range: EBASE-ELIMIT

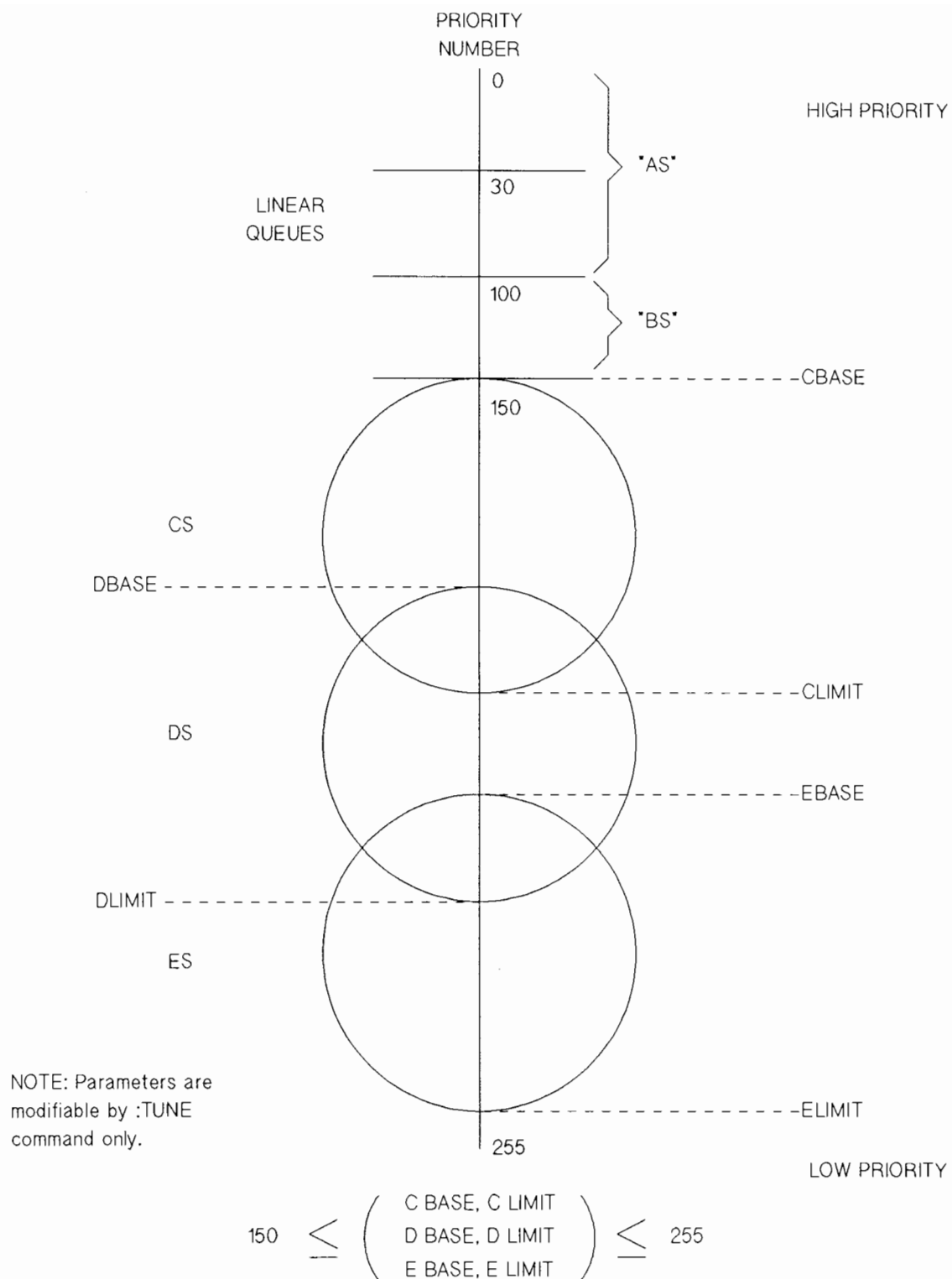


Figure 3-2. Master Queue Structure



In all cases, it should be remembered that low numeric values mean high priority in the system.

The system manager/system supervisor (only users with OP capability) has the ability to modify the values of the starting priority (BASE) and priority limits (LIMIT) for each circular queue, as well as the average short transaction limit and background quantum, via the **:TUNE** command.

A CS process is given a priority of CBASE when it begins (see Figure 3-2). When a process stops (e.g. for disk I/O, terminal I/O, preemption), its new priority is determined so that it may be 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 lower than CLIMIT.

DS and ES processes begin at DBASE and EBASE respectively, and are rescheduled according to the same criteria used for CS processes. The exception is that a fixed value (the value of *max* which has been specified for the DS subqueue) is used for the DS and ES subqueues 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 user with PH capability. 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 highest priority within that class (relative to other processes already assigned the same class).

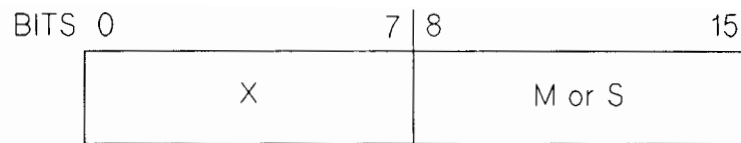
In a circular subqueue, the actual priority of the process is modified as other processes use central processor time.

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.

Priorities are assigned to processes through the *priorityclass* parameter of the **CREATE** and/or **GETPRIORITY** intrinsic. Because users with the privileged mode (PM) capability can schedule processes within the master queue, the *priorityclass* parameter can take on the following values:

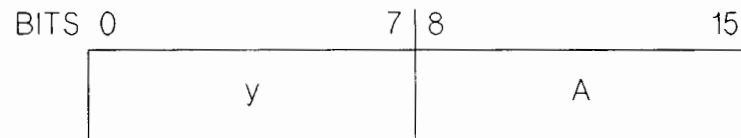
- A string of two ASCII characters describing the standard priority class (subqueue) in which the process is to be scheduled: **AS**, **BS**, **CS**, **DS**, or **ES**.
- An ASCII character or characters (*x*) specifying a valid subqueue, followed immediately by the single-character string **M**, indicating the master queue, or **S**, indicating the subqueue.

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.

- An ASCII character or characters (*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.

Data Segment Management Capability

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”. (Refer to the *HP 3000 Computer Systems General Information Manual* (5953-7553) for further discussion 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. 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 (DS) 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 as private to the creating process. All private data segments created by a process are either deleted explicitly with the **FREEDSEG** intrinsic or are destroyed automatically when the process terminates. Sharable data segments are saved until explicitly deleted by the last process which is sharing them, or until that process terminates, at which time they are destroyed.

Extra data segments are not directly addressable by user processes. They can be accessed only through intrinsics (**DMOVIN**, **DMOVOUT**) that move data between the user's stack and the extra data segments. If a program (or process running a program) without data segment management capability attempts to call these intrinsics, that process is aborted. The data segment management capability is assigned to the program at **:PREP** time by a user with this capability (**:PREP ... ;CAP=DS**).

The maximum number of extra data segments allowed per process and the maximum size of each segment is determined at system configuration time.

A user who possesses the data segment management capability, 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 of 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, assigned by MPE, to the calling process that allows this process to reference the segment in later intrinsic calls.

The **GETDSEG** intrinsic is also used to assign the segment an identity that either allows other processes in the same job or session to share the segment, or 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 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- or session-wide name that allows any process to identify the data segment in order to obtain a logical index.

Figure 3-3, 3-4, and 3-5 each contain a program which illustrates the use of the **GETDSEG**, **DMOVOUT**, or **DMOVIN** intrinsic. Together, these three programs do the following:

1. Create an extra data segment which can be shared by all three processes.

2. Compute Julian calendar dates (a Julian date is the sequential number of the day within the year; e.g. February 1 is represented as "32").
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 3-5, each of which shares the same code, but has its own data stack.
5. Each of the processes created in Step 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 3-3, 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 3-4, called DSBOS, creates and activates two processes. Each of the two processes created is a process to run the program shown in Figure 3-5. Thus, each process shares the same code, but has its own data stack. The program in Figure 3-5 called DSACCS, opens a terminal for input/output, acquires the extra data segment created by DSINIT, requests a month and day from the user, then transfers the Julian dates contained in that particular month into its own data stack.

Because DSBOS (Figure 3-4) has created two processes for the program shown in Figure 3-5, and has activated both processes, the functions performed by DSACCS are duplicated (i.e. two terminals are opened for input/output, or two users can enter the month and day).

Note



The three programs in Figure 3-3, Figure 3-4, and Figure 3-5 must specify the Data Segment (DS) or Process handling (PH) capability when they are prepared, as follows:

DSINIT (Figure 3-3):

```
:PREP $OLDPASS,DSINIT;CAP=DS,IA,BA
```

DSBOS (Figure 3-4):

```
:PREP $OLDPASS,DSBOS;CAP=PH,IA,BA
```

DSACCS (Figure 3-5):

```
:PREP $OLDPASS,DSACCS;CAP=DS,IA,BA
```

In all cases above, it is assumed that \$OLDPASS contains the compiled USL file for each of the three programs.

In Figure 3-3, the statement (00011000/00012000) initializes a 12-word integer array to represent the number of days in each month of the year:

```
INTEGER ARRAY MAXDAY(0:11):=31,28,31,30,31,30
```

31,31,30,31,30,31;

The next statement (00013000) of program DSINIT declares a 372-word integer array and initializes all 372 words to -1:

INTEGER ARRAY CALENDAR(0:371):=372(-1);

The two **FOPEN** intrinsic calls open **\$STDIN** and **\$STDLIST** so that **FREAD** and **FWRITE** intrinsic calls can be issued against these files.

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1989, 4:35 PM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1  INTEGER IN,OUT,LGTH,MONTH,DAY,YEAR,DATE:=1,DSLNGTH:=372;
00006000 00005 1  LOGICAL DSINDX;
00007000 00005 1  ARRAY READ(0:14):="GENERATE CALENDAR DATA SEGMENT";
00008000 00017 1  ARRAY BUFR(0:1):=2(" ");
00009000 00001 1  BYTE ARRAY BBUF(*)=BUFR;
00010000 00001 1  ARRAY REQST(0:5):="ENTER YEAR: ";
00011000 00006 1  INTEGER ARRAY MAXDAY(0:11):=31,28,31,30,31,30,
00012000 00006 1  31,31,30,31,30,31;
00013000 00014 1  INTEGER ARRAY CALENDAR (0:371):=372(-1);
00014000 00001 1  DEFINE CCL = IF < THEN QUIT#,
00015000 00001 1  CCNE= IF <> THEN QUIT#;
00016000 00001 1
00017000 00001 1  INTRINSIC FOPEN,FREAD,FWRITE,GETDSEG,DMOVOUT,BINARY,QUIT;
00018000 00001 1
00019000 00001 1  <<END OF DECLARATIONS>>
00020000 00001 1
00021000 00001 1  IN:=FOPEN(INPUT,%45); CCL(1); <<$STDIN>>
00022000 00012 1  OUT:=FOPEN(OUTPUT,%414,1); CCL(2); <<$STDLIST>>
00023000 00025 1
00024000 00025 1  FWRITE(OUT,HEAD,15,0); CCNE(3); <<PROGRAM ID>>
00025000 00035 1
00026000 00035 1  GETDSEG(DSINDX,DSLNGTH,"JD");CCL(4);<<SHARED EXTRA DS>>
00027000 00044 1
00028000 00044 1  FWRITE(OUT,REQST,6,%320); CCNE(5); <<REQST CLNDR YR>>
00029000 00054 1  LGTH:=FREAD(IN,BUFR,-4); CCNE(6); <<INPUT YEAR>>
00030000 00065 1  YEAR:=BINARY(BBUF,LGTH); CCNE(7); <<CONVERT YEAR>>
00031000 00075 1
00032000 00075 1  IF YEAR MOD 4=0 THEN MAXDAY(1):=29;<<FIX FEB-LEAP YR>>
00033000 00105 1
00034000 00105 1  FOR MONTH:=0 UNTIL 11 DO <<INDEX 12 MONTHS>>
00035000 00112 1  FOR DAY:=0
00036000 00127 1  UNTIL MAXDAY(MONTH)-1 DO <<INDEX DAYS/EA MO>>
00037000 00132 2  BEGIN
00038000 00140 2  CALENDAR(MONTH*31+DAY):=DATE; <<SET JULIAN DATE>>
00039000 00141 2  DATE:=DATE+1 <<INCR JULIAN DATE>>
00040000 00143 1  END;
00041000 00143 1  DMOVOUT(DSINDX,0,372,CALENDAR); <<JULIAN CLNDR TO DS>>

00042000 00150 1  CCNE(8); <<CHECK FOR ERROR>>
00043000 00153 1 END.
PRIMARY DB STORAGE=%021; SECONDARY DB STORAGE=%00636
NO. ERRORS=000; NO. WARNINGS=000
PROCESSOR TIME=0:00:03; ELAPSED TIME=0:00:10

```

Figure 3-3. Using the GETDSEG and DMOVOUT Intrinsics (Program DSINIT)

```

PAGE 0001    HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1989, 4:35 PM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1  INTEGER IN,OUT,LGTH,PIN1,PIN2;
00006000 00005 1  BYTE ARRAY PFILE(0:6):="DSACCS ";
00007000 00005 1  ARRAY MESSAGE(0:29):="WHEN ALL JULIAN DATE OPERATIONS ARE ",
00008000 00022 1                                "COMPLETE TYPE: DONE.",%6412,"? ";
00009000 00036 1  ARRAY BUFR(0:1);
00010000 00036 1  BYTE ARRAY BBUF(*)=BUFR;
00011000 00036 1  DEFINE CCL= IF < THEN QUIT#,
00012000 00036 1                CCNE= IF <> THEN QUIT#;
00013000 00036 1
00014000 00036 1  INTRINSIC FOPEN,FWRITE,FREAD,CREATE,ACTIVATE,QUIT;
00015000 00036 1
00016000 00036 1  <<END OF DECLARATIONS>>
00017000 00036 1
00018000 00036 1          IN:=FOPEN(INPUT,%45); CCL(1);          <<$STDIN>>
00019000 00012 1          OUT:=FOPEN(OUTPUT,%414,1); CCL(2);  <<$STDLIST>>
00020000 00025 1
00021000 00025 1          CREATE(PFILE,,PIN1,1); CCNE(3); <<SON 1-TERMD1 FILE>>
00022000 00037 1          CREATE(PFILE,,PIN2,2); CCNE(4); <<SON 2-TERMD2 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);<<TERMNATN INSTR>>
00029000 00077 1          LGTH:=FREAD(IN,BUFR,-4); CCNE(8);  <<SUSPD FOR READ>>
00030000 00110 1
00031000 00110 1          IF BBUF<>"DONE" THEN GO WAIT;<<END IF DONE-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 3-4. Creating and Activating Two Son Processes (Program DSBOS)

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1989, 4:35 PM
00001000 00000 0 $CONTROL 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,DSLGH,DATE,CURRENT:=1;
00006000 00004 1 LOGICAL DSINDX,PARM=Q-4;
00007000 00004 1 ARRAY READ(0:9):="JULIAN DATE CALENDAR";
00008000 00012 1 ARRAY BUFR(0:1);
00009000 00012 1 BYTE ARRAY BBUFR(*)=BUFR;
00010000 00012 1 ARRAY MESSAGE(0:16):="MONTH = ","DAY = ","JULIAN DATE = ";
00011000 00021 1 BYTE ARRAY BMSG(*)=MESSAGE;
00012000 00021 1 ARRAY DATES(0:30);
00013000 00021 1 DEFINE CCNE = IF <> THEN QUIT#;
00014000 00021 1 INTRINSIC FOPEN,FREAD,FWRITE,GETDSEG,DMOVIN,BINARY,ASCII,QUIT;
00015000 00021 1 <<END OF DECLARATIONS>>
00016000 00021 1 NAME(6):=PARM; <<FORMALDES #=1 OR 2>>
00017000 00003 1 FNO:=FOPEN(NAME,%405,4,36,DEV); <<TERM FILE TERMIO# >>
00018000 00015 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00019000 00020 1 FWRITE(FNO,READ,10,0); CCNE(2); <<PROGRAM ID>>
00020000 00030 1 GETDSEG(DSINDX,DSLGH,"JD"); <<SHARED CALENDAR DS>>
00021000 00034 1 IF <= THEN QUIT(3); <<ERR OR NONEXISTENT>>
00022000 00037 1 GETMO:
00023000 00037 1 FWRITE(FNO,MESSAGE,4,%320); CCNE(4);<<REQUEST MONTH>>
00024000 00047 1 MOVE BUFR:=" "; <<BLANK READ BUFFER>>
00025000 00061 1 LGTH:=FREAD(FNO,BUFR,-2); CCNE(5);<<INPUT MONTH>>
00026000 00072 1 IF LGTH=0 THEN GO EXIT; <<NO MONTH - DONE>>
00027000 00075 1 MONTH:=BINARY(BBUFR,LGTH); <<CONVERT MONTH>>
00028000 00102 1 IF <> THEN GO GETMO; <<IF BAD TRY AGAIN>>
00029000 00103 1 IF NOT(1<=MONTH<=12) THEN GO GETMO;<<ILLEGAL MONTH CHK>>
00030000 00112 1 GETDA:
00031000 00112 1 FWRITE(FNO,MESSAGE(4),3,%320); CCNE(6);<<REQUEST DAY>>
00032000 00123 1 MOVE BUFR:=" "; <<BLANK READ BUFFER>>
00033000 00132 1 LGTH:=FREAD(FNO,BUFR,-2); CCNE(7);<<INPUT DAY>>
00034000 00143 1 DAY:=BINARY(BBUFR,LGTH); <<CONVERT DAY>>
00035000 00150 1 IF <> THEN GO GETDA; <<IF BAD TRY AGAIN>>
00036000 00151 1 IF NOT(1<=DAY<=31) THEN GO GETDA; <<ILLEGAL DAY CHK>>
00037000 00156 1 IF MONTH<>CURRENT THEN <<MONTH NOT IN BUFR>>
00038000 00161 1 BEGIN
00039000 00161 2 DMOVIN(DSINDX,(MONTH-1)*31,31,<<GET MO FRM CLNDR>>
00040000 00166 2 DATES); CCNE(8); <<CHECK FOR ERROR>>
00041000 00173 2 CURRENT:=MONTH; <<UPDATE MONTH IN BUFR>>
00042000 00175 2 END;

```

(Part 1) Using the GETDSEG and DMOVIN Intrinsics (Program DSACCS)

```

00043000 00175 1      JDATE:=DATES(DAY-1);          <<GET JULIAN DATE>>
00044000 00201 1      IF JDATE<0 THEN GO GETDA;      <<UNINITIALIZED DATE>>
00045000 00204 1      MOVE MESSAGE(14):="    ";      <<BLANK OUTPUT BUFR>>
00046000 00216 1      LGTH:=ASCII(JDATE,10,BMSG(28)); <<CNVRT JULIAN DATE>>
00047000 00225 1      FWRITE(FNO,MESSAGE(7),10,0); CCNE(9);<<OUTPT DATE-TERM#>>
00048000 00236 1      GO GETMO;                      <<CONTINUE>>
00053000 00237 1      EXIT:END.

```

Figure 3-5 (Part 2) Using the GETDSEG and DMOVIN Intrinsics (Program DSACCS)

An extra data segment is created with statement 00026000 in Figure 3-3:

```
GETDSEG(DSINDEX,DSLNGTH,"JD");
```

The parameters specified are:

<i>index</i>	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.
<i>length</i>	DSLNGTH , which has been initialized to 372 words (see statement
<i>id</i>	"JD", which specifies that this data segment is sharable by other processes in the same job/session. Any process which is to create or share an extra data segment must have the Data Segment (DS) capability. If the data segment being created is private to the creating process, specify 0 for <i>id</i> .

Statements 00028000, 00029000, and 00030000 in Figure 3-3 request the user to enter the calendar year, and convert this ASCII string to a binary value.

Statement 00032000 of Figure 3-3 checks if the year is equally divisible by four, and if it is, adds the twenty-ninth day to February for the leap year.

The six statements beginning on line 00034000 of Figure 3-3 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**).

Statement 00038000 of Figure 3-3 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** is filled with Julian dates. All elements of the array were initialized to -1, and positions in the array that retain the value -1 signify invalid dates.

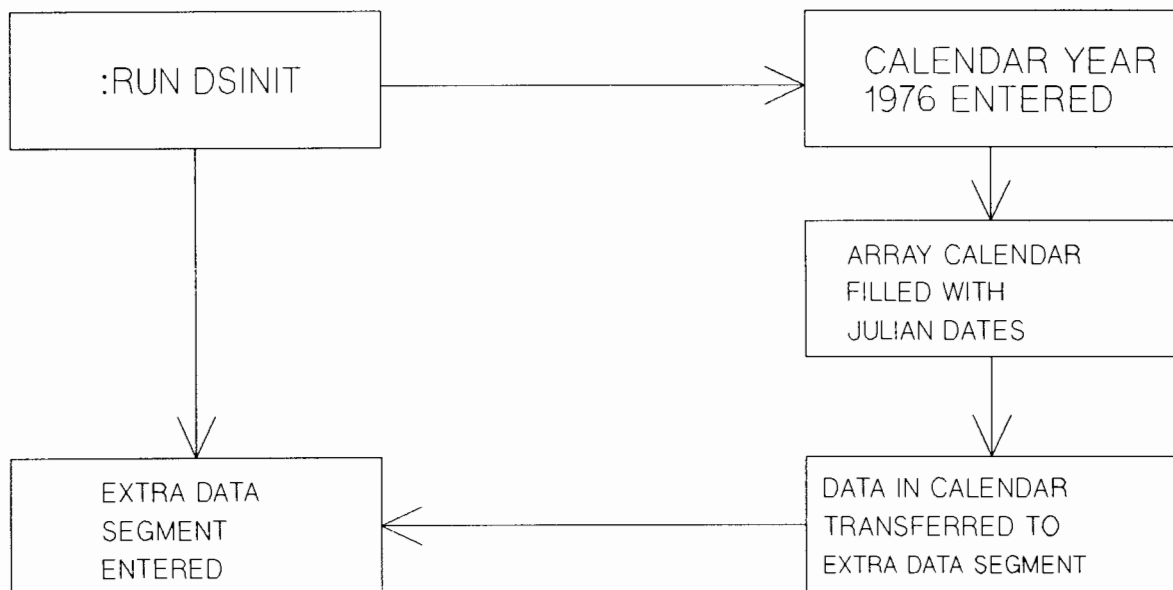
The data contained in **CALENDAR** is transferred from the stack to the extra data segment with statement 00041000 of Figure 3-3:

DMOVOUT (DSINDEX,0,372,CALENDAR)

The parameters specified are:

<i>index</i>	DSINDEX , which contains the logical index returned by MPE when the GETDSEG intrinsic was executed.
<i>disp</i>	0 , which specifies the first word in the data segment. This is the starting location for the first word transferred from CALENDAR to the extra data segment.
<i>number</i>	372 , which specifies the size, in words, of the data block to be transferred.
<i>location</i>	CALENDAR , which specifies the starting address in the stack, of the data block to be transferred.

At this point, the following events have occurred:



When the `:RUN DSBOS` command is entered, the program in Figure 3-4 (DSBOSS) executes.

Statements 00018000 and 00019000 of Figure 3-4 open `$STDIN` and `$STDLIST` to accept `FREAD` and `FWRITE` intrinsic calls.

Statement 00021000 of Figure 3-4:

```
CREATE(PFILE,,PIN1,1);
```

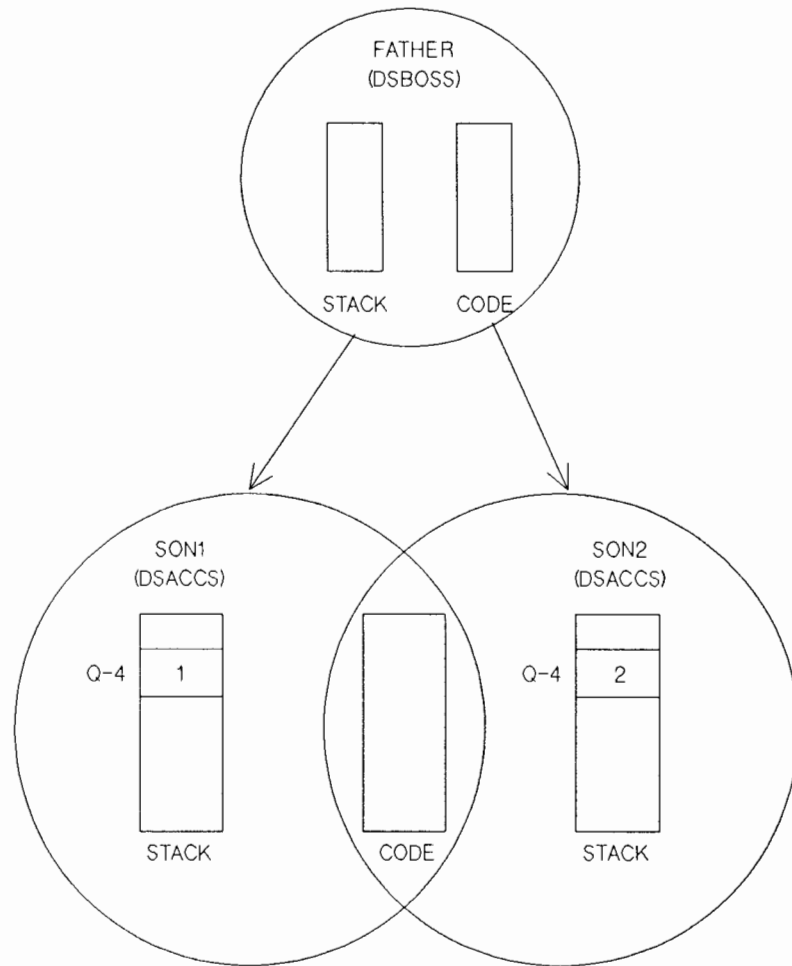
creates a process. The parameters specified are:

<i>programe</i>	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 3-5, thus the process is created for this program.
<i>entryname</i>	Omitted; a comma place holder is used. The primary entry point is specified by default.
<i>pin</i>	PIN1 , a word to which the process identification number of the process will be returned.
<i>param</i>	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.

Statement 00022000 of Figure 3-4 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; the *susp* parameter 0 specifies that the father process will not be suspended when the sons are activated. Program DSBOS, therefore, has created and activated two processes as follows:



The four statements beginning with 00027000 of Figure 3-4 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 resuspends itself. However, while the father was active, SON2 terminated. The attempt by SON2 to reactivate the father failed, because the father was already active. Thus, when the father resuspends itself, it suspends indefinitely because both sons have terminated.

The two processes, SON1 and SON2, both of which are DSACCS, are shown in Figure 3-5.

Statement 00017000 of Figure 3-5:

```
FNO:=FOPEN(NAME,%405,4,36,DEV);
```

opens a terminal for input/output. The parameters specified are:

formal designator

NAME, which contains the string **TERMIO1** when this call is issued by SON1 and **TERMIO2** when the call is issued by SON2. Note that in statement 00003000 of Figure 3-5, **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 DSBOS.) Thus, by using different formal designators, SON1 opens one terminal, and SON2 opens another.



Note



Unlike disk 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, the 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:

(14:2)=01 The file is an old file.

(13:1)=1 Type ASCII.

(10:3)=000 Actual file designator is the same as the formal file designator.

(8:2)=00 Fixed-length records.

(7:1)=1 Carriage-control character expected.

aoptions

4, which specifies input/output access.

recsize

'6', 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,DSLNGTH,"JD");
```

Note that the process quits unless CCG, signifying that an extra data segment with this identifier exists already, is returned. (See Figure 3-5, Statement 00025000.)

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.

Statement 00039000 of Figure 3-5:

```
IF MONTH<>CURRENT THEN
```

checks whether the month information is different than the information currently in the stack. If it is, statement 00041000 of Figure 3-5:

```
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 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 statement 00041000 for Figure 3-5>:

```
CURRENT:=MONTH;
```

The Julian date is computed with statement 00043000 of Figure 3-5:

```
JDATE:=DATES(DAY-1);
```

and this information is output to the user.

The following examples illustrate using the three programs in Figure 3-3, Figure 3-4, and Figure 3-5:

EXAMPLE 1:

```
:RUN DSINIT
```

```
GENERATE CALENDAR DATA SEGMENT
```

```
ENTER YEAR: 1983
```

```
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 = 39
```

```
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;TERMIO2
```

```
JULIAN DATE CALENDAR
```

```

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 requests the user to enter the year for which Julian dates are required. When 1983 is entered, DSINIT creates an extra data segment, fills an array with Julian dates for the year 1983, transfers this data to the extra data segment, and terminates.

The `:RUN DSBOS` command causes DSBOS to execute. DSBOS 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. First a user enters a `:DATA` statement on a terminal. (Remember that SON1 and SON2 have each opened a terminal for input/output.) Then, MPE searches the device class directory for a terminal with the formal designator **TERMIO1** 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 statement 00044000 of Figure 3-5:

```

IF JDATE < 0 THEN GO GETDA;

```

DSACCS prompts for a new day and the user enters 30; DSACCS computes the Julian date for November 30 and displays:

```

JULIAN 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 by using the **FREEDSEG** intrinsic. If the extra data segment is private, or if it is a sharable (nonprivate) data segment not currently assigned to any other process in the job/session, it is deleted from the entire job/session. If the extra data segment is sharable (nonprivate), and currently assigned to any other process in the job/session, 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:

```
FREEDSEG(INDX,0);
```

Because the data segment is private to this process, the *id* parameter is specified as 0 and the segment will therefore cease to exist.

Transferring Data from an Extra Data Segment to the Stack

A process can copy a block of words from an extra data segment to the stack with the **DMOVIN** intrinsic. A bounds check is performed by the intrinsic on both the extra data segment and the stack to ensure 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 3-5 and described earlier in this chapter.

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 ensure 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 3-3, and described earlier in this chapter.

Changing the Size of an Extra Data Segment

You can change the current size of an extra data segment with the **ALTDSEG** intrinsic. In a typical application, disk 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 is usually allocated in increments of 512 words (depending on virtual memory configured in the system configuration). 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 disk 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);

The parameters specified are:

<i>index</i>	INDEX is a word containing the logical index of the extra data segment, obtained through the GETDSEG intrinsic call.
<i>inc</i>	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.
<i>size</i>	SIZE is a word to which the new size of the data segment is returned after incrementing or decrementing has occurred.

Process Handling Capability

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 users with standard MPE capabilities. Such users have no control over processes or their structure, since 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 process handling (PH) which is discussed in this chapter. 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 example, 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 may be shared by many processes because the segments are not owned by the processes.

When a user enters MPE, a process is created for that user. This process is called a Job/Session Main Process (JSMP). The process is linked into the Command Interpreter (CI), 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 (such as programs, procedures, private code, sharable code), and one of the main elements needed to establish a new process is a starting address (a "program label"). Beginning with the starting address, the process follows the sequence of the code until its deletion.

PROGEN, 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 or other processes, and in particular the "User Controller Process" (UCOP). All these processes may, if required, have their own structure of descendants.

PROGEN is the ancestor of all processes in MPE, including system processes, as the User Controller Process is the ancestor of all user processes currently in existence. The UCOP is consequently the root of the user process Tree Structure. The sons of the UCOP are called (User) main processes or the JSMPs.

The father/son relationship between processes is used primarily to maintain control from top to bottom throughout the structure. In most cases the father is held "responsible" for what happens to its son: creation, deletion and other special actions.

Organization of User Processes

When you logon (initiate a job or session), 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: a job is associated with a batch type of access, while a 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.

Active and Suspended Process Substates

During its life span (that is, between creation and 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 substates can be controlled by users: active and suspended.

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 programmatically terminate one of your processes. The termination destroys the process and all of its descendants, and resets the links of the remaining processes for the session or job.

Interprocess Communication (IPC) is a facility of the file system which permits multiple user processes to communicate easily and efficiently with each other. 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 **FWRITE** and deletions made by **FREAD**: one process may submit records to the file with the **FWRITE** intrinsic while another process takes records from the file using **FREAD**.

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.

The flow of information between a given process and a message file is unidirectional. 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 same 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. For more detail on Interprocess Communication, refer to the *MPE File System Reference Manual* (30000-90236).

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.

You can also create processes with the **CREATEPROCESS** intrinsic. It contains a superset of the **CREATE** intrinsic and, therefore, is more flexible and expandable. **CREATEPROCESS** allows you to assign **\$STDIN** and **\$STDLIST** values to a file when the process is created. By using **CREATEPROCESS** you are not limited to the system defaults for **\$STDIN** and **\$STDLIST**.

Once a process is created, it must be activated with the **ACTIVATE** intrinsic in order to run. When a process is activated, it may or may not 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 activated only by its son. A father process that has been suspended when a son process was activated can be reactivated automatically when the son process execution ends, if the *flags* parameter of the **CREATE** intrinsic bit is set ((15:1)=1). A program that has been suspended with the **SUSPEND** intrinsic can be reactivated by its father or any of its sons, as specified in the *susp* parameter of the **SUSPEND** intrinsic.

Figure 3-6 contains a program which illustrates the **CREATE** and **ACTIVATE** intrinsics. Statement 00027000 reads the name input by the user on **\$STDIN**, and stores the name in the logical 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 made equivalent to **NAME** in statement 00008000 in the program. Additionally, the string must be terminated by a blank, and statement 00030000 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 statement 00034000:

```
CREATE(BNAME,,PIN,,1);
```

The following parameters were specified in the **CREATE** intrinsic call. All other parameters were omitted in the **CREATE** intrinsic call:

<i>progrname</i>	Specified by BNAME , which contains the name entered by the user.
<i>entryname</i>	Omitted; a comma place holder is used. The primary entry point of the created process is specified by default.
<i>pin</i>	The process identification number (PIN), to be used by the ACTIVATE intrinsic, is returned to the word PIN .
<i>param</i>	Omitted; a comma place holder is used. A word filled with zeros is specified by default.
<i>flags</i>	1, which specifies that this (the father) process will be reactivated automatically by MPE when the created process execution ends (bit(15:1)=1).

All other parameters are omitted in the **CREATE** intrinsic call.

Statement 00037000:

```
FWRITE(OUT,ACTMSG,8,0);
```

displays the message:

ACTIVATE PROCESS

and statement 00039000 of Figure 3-6 calls the **ACTIVATE** intrinsic to activate the process. The following parameters were specified:

<i>pin</i>	Specified by PIN , which contains the Process Identification Number of the process to be activated, as returned to the CREATE intrinsic by the system.
<i>susp</i>	If <i>susp</i> is specified, the calling process will be suspended when the called process is activated. When 2 (bit(14:1)=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 <i>susp</i> had not been specified, the calling (father) process would not have been suspended when the called process was activated.

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1989, 4:35 PM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1  INTEGER IN,OUT,LGTH,PIN;
00006000 00005 1  ARRAY REQST(0:8):=%6412,"PROGRAM ENTITY = ";
00007000 00011 1  ARRAY NAME(0:13);
00008000 00011 1  BYTE ARRAY BNAME(*)=NAME;
00009000 00011 1  ARRAY CRMSG(0:6):="CREATE PROCESS";
00010000 00008 1  ARRAY ACTMSG(0:7):="ACTIVATE PROCESS";
00011000 00010 1  DEFINE CCL=IF < THEN QUIT#,
00012000 00010 1           CCG=IF > THEN QUIT#,
00013000 00010 1           CCNE=IF <> THEN QUIT#;
00014000 00010 1
00015000 00010 1  INTRINSIC FOPEN,FREAD,FWRITE,CREATE, ACTIVATE,QUIT;
00016000 00010 1
00017000 00010 1  <<END OF DECLARATIONS>>
00018000 00010 1
00019000 00010 1           IN:=FOPEN(INPUT,%45);           <<$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 PGM 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 3-5. Using the CREATE and ACTIVATE Intrinsics (Program PROG)

A sample run of the program listed in Figure 3-6: (named PROG) results in the following:

:RUN PROG

PROGRAM ENTITY = SPL.PUB.SYS

CREATE PROCESS

ACTIVATE PROCESS

HEWLETT-PACKARD 32100A.08.1 SPL[4W] TUE, OCT 28, 1989, 4:35 PM

>\$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 ENTITY = SEGDVR.PUB.SYS

CREATE PROCESS

ACTIVATE PROCESS

SEGMENTER SUBSYSTEM (C.O)

-USL \$OLDPASS

-PREPARE \$NEWPASS

-EXIT

PROGRAM ENTITY = \$OLDPASS

CREATE PROCESS

ACTIVATE PROCESS

* TEST PROCESS EXECUTING *

PROGRAM ENTITY = RETURN

END OF PROGRAM

:

When SPL.PUB.SYS is entered in response to the "PROGRAM ENTITY =" 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 PROG at the statement following the **ACTIVATE** intrinsic call, and causing the "PROGRAM ENTITY =" message to be displayed again.

The response:

```
SEGDVR.PUB.SYS
```

causes PROG to create and activate the Segmenter Driver (a programmatic entry point to the Segmenter subsystem). The Segmenter displays the following and a prompt (-):

```
SEGMENTER SUBSYSTEM (C.0)
```

The small program written in SPL and compiled into the USL file \$OLDPASS (the default USL file since a *uslfile* parameter could not be included in the SPL.PUB.SYS response) is identified with the Segmenter command:

```
-USL $OLDPASS
```

The next command:

```
-PREPARE $NEWPASS
```

prepares the SPL program and the Segmenter is exited with the command:

```
-EXIT
```

Once again, PROG 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 and displays:

```
*TEST PROCESS EXECUTING*
```

then ends execution, reactivating PROG.

A **RETURN** (signifying no input) is entered in response to the "PROGRAM ENTITY =" request and the program terminates.

The example below uses PROG to create and activate a duplicate of itself:

```
:RUN PROG
```

```
PROGRAM ENTITY = PROG  
CREATE PROCESS  
ACTIVATE PROCESS
```

```
PROGRAM ENTITY = PROG  
CREATE PROCESS  
ACTIVATE PROCESS
```

```
PROGRAM ENTITY = RETURN **3**
```

```
PROGRAM ENTITY = RETURN **2**
```

```
PROGRAM ENTITY = RETURN **1**
```


END OF PROGRAM

:

When PROG is entered in response to the "PROGRAM ENTITY =" request, the calling process (PROG) creates a duplicate of itself and activates this process (for clarity, call this PROG1). This process executes and requests a file name. The user enters PROG again, causing yet another duplicate (call this one PROG2) to be created and activated. At this point, PROG is suspended: it has created and activated a duplicate process. The duplicate process (PROG1) has, in turn, created and activated a duplicate of itself (PROG2). Thus, it also is suspended.

The third process (PROG2) executes and displays:

PROGRAM ENTITY =

A carriage return (see `RETURN` **3** in the example) causes this process to stop executing and control returns to PROG1. PROG1 displays:

PROGRAM ENTITY =

A second carriage return (see `RETURN` **2** in the example) causes this process to stop executing and control returns to PROG.

PROGRAM ENTITY = is displayed once more, this time by the original process. Another carriage return (see `RETURN` **1** in the example) causes PROG 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 can also release a local Resource Identification Number (RIN) when the process is suspended by specifying the RIN as a parameter in the intrinsic call.

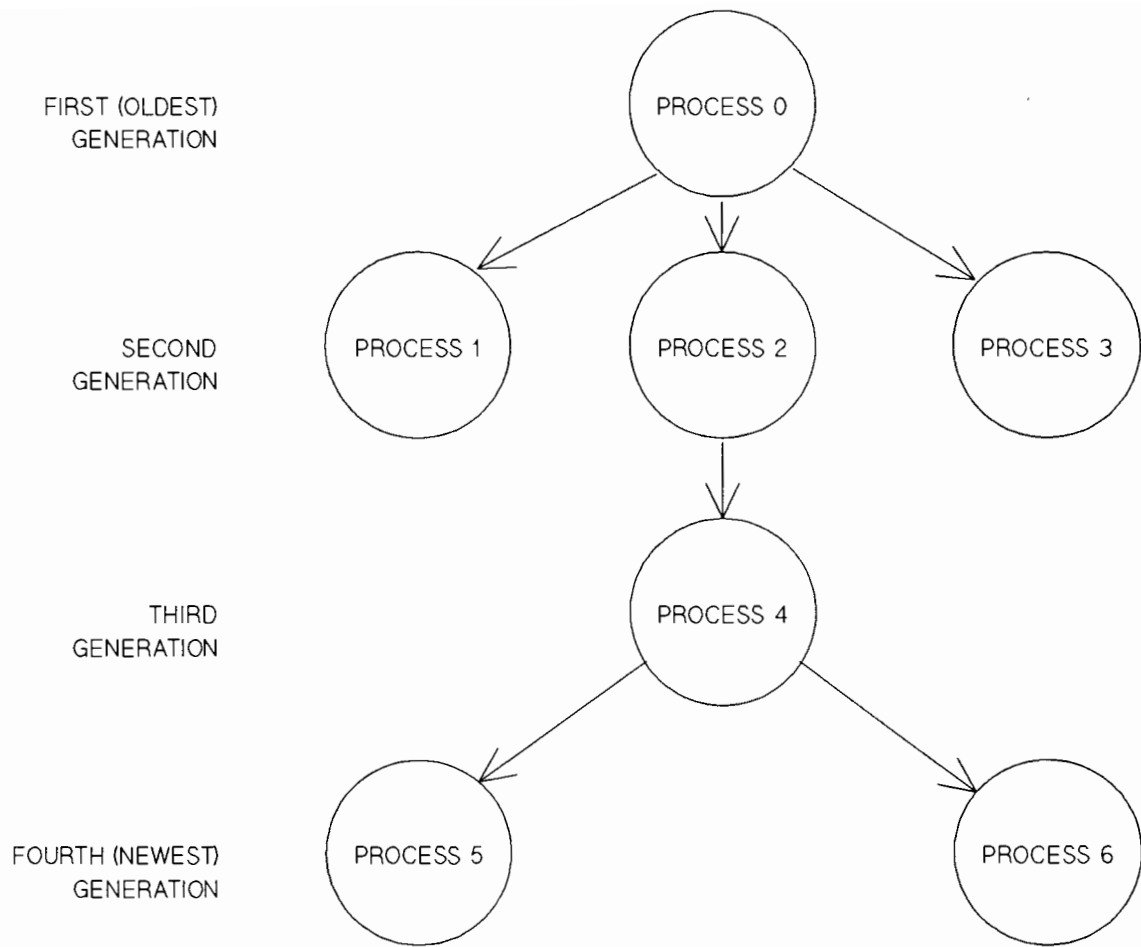
The intrinsic call:

SUSPEND(3,RINNUM);

would cause the process to suspend itself and release the local RIN specified by **RINNUM**. The parameter 3 (bit (14:1)=1 and bit (15:1)=1) specifies that the process expects to be reactivated by its father or one of its sons.

Deleting Processes

A process can delete itself with the **TERMINATE** intrinsic, or delete 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 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 descendants of that process, as shown in Figure 3-6. Within a process tree structure, the newest generations are deleted first. Within each generation, processes are deleted in the order of their creation.



(WHEN PROCESS 2 IS DELETED, THE FOLLOWING STRUCTURE REMAINS.)

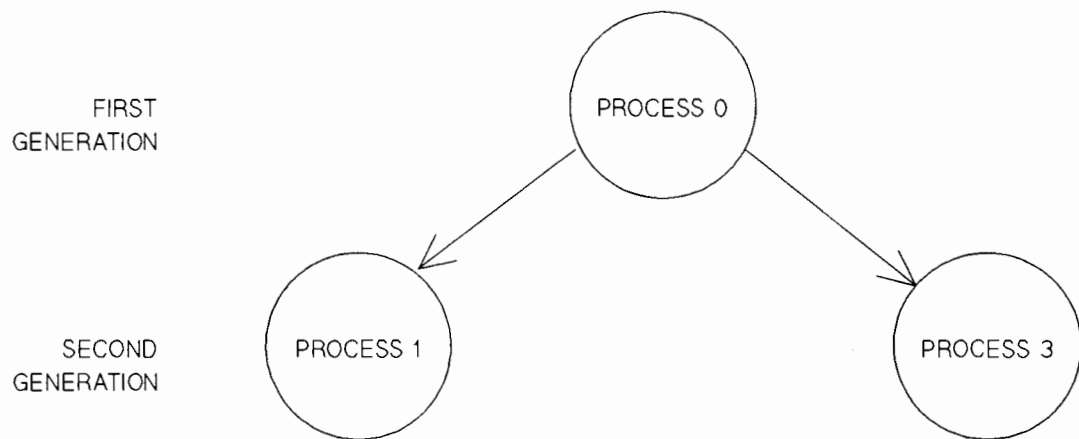


Figure 3-6. Process Deletion

Interprocess Communication

In the 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.

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. This feature should not be confused with message files, which can be used to transfer information between unrelated processes.

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, there is only one mailbox between any two processes.

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 may test 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 to ensure that the array specified actually falls within the limits of the process stack.
3. The receiving process optionally may test 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 outgoing 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. The length of the mail is returned in MCOUNT .
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 mailbox is currently being used by other intrinsics, the **SENDMAIL** intrinsic waits until it 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 three words.
<i>location</i>	LOCAT , a logical array in the stack containing the message to be sent.

waitflag **WAITSTAT**, a logical word. If bit (15:0)=1, any mail sent previously will be overwritten. If bit (15:1)=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 <i>pin</i> 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. If this size exceeds one word, then the maximum size cannot exceed the maximum DST size for 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 currently being used 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 (0 for father process).
<i>location</i>	MDATA , a logical array in the stack in which the incoming mail will be stored.
<i>waitflag</i>	WAITSTAT , a logical word. If bit (15:1)=1, the intrinsic will wait until the incoming mail is ready for collection. If bit (15:0)=1, the intrinsic will return to the calling process immediately.

One of the following status codes is returned to **STAT**:

Status	Meaning
0	The mailbox was empty (and WAITSTAT bit (15:1)=0).
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 <i>pin</i> 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 both processes were suspended, neither could activate the other, and they may be deadlocked.

Avoiding Deadlocks

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. 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. Avoid this situation.
2. An attempt by a process to lock a global **RIN** succeeds only if both the following 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 can 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 *HP 3000 Computer Systems General Information Manual* (5953-7553). The standard linear subqueues are:

- The **AS** subqueue, containing system processing only.
- The **BS** subqueue, containing processes of very 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 (batch).
- The **ES** subqueue operates at very low priority (background).

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 **DS**, 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 (**PIN**) 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 could be returned to **SOURCE**:

- 1 Indicating that the process was activated by its father.
- 2 Indicating that the process was activated by a son.

Determining Father Process

A process can determine the process identification number of its father with the **FATHER** intrinsic.

The process identification number of the father is returned to **PIN**. For example, the following intrinsic call could be used:

```
PIN:=FATHER
```

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);
```

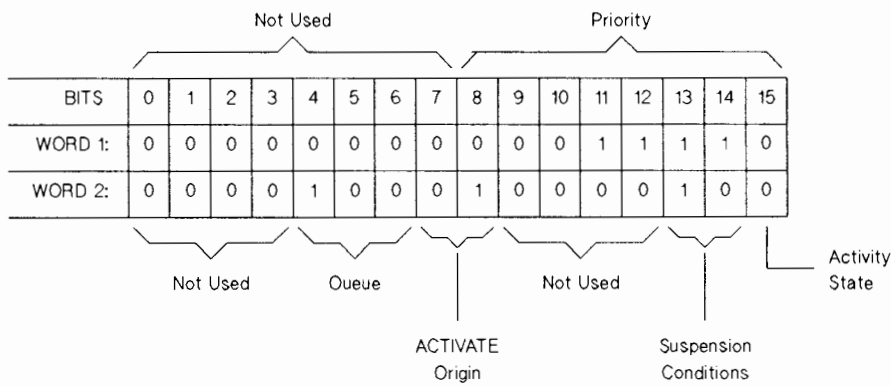
**Determining Process
Priority and State**

A process can request the return of a double-word message denoting the following information about its father or sons with the **GETPROCINFO** intrinsic (refer to the **GETPROCINFO** discussion in Chapter 2). 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 is a son process, the process identification number of the son process is 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)	00011110	Process has priority 30 in master queue.
	(0:8)	00000000	Not used.
2	(15:1)	0	Process is suspended.
	(14:1)	0	Would not expect the source of activation to be the father.
	(13:1)	1	The source of expected activation is the son.
	(9:4)	0000	Not used.
	(7:2)	01	Origin of last ACTIVATE call was father of this process.
	(4:3)	100	Circular subqueue.

Resource Management

Any element of the HP 3000 which a program can access through MPE is called a resource. A resource can be an input or output device, a file, subroutine, procedure, code segment, or the data stack.

Occasionally, you may want to manage a specific resource shared by a set of jobs or processes, so that no two jobs or processes can use the resource at the same time. To accomplish resource management within a job or session, or between processes, the jobs or processes must cooperate. For example, if Job B must not access a particular file when Job A is using it, both jobs should contain provisions for a "hand-shaking" arrangement overseen by MPE when both jobs are active. Job B will be denied access to the file while Job A is accessing it, or be suspended until Job A releases its exclusive access.

Under this arrangement, when Job A has exclusive access to the file, when Job B attempts to access the same file, access will be denied, and Job B will be suspended until Job A releases its exclusive access. Then Job B can resume execution and access the file. However, if Job B has been suspended by MPE, it is unable to access the file and in fact cannot execute at all.

The hand-shaking arrangement is based on an arbitrary Resource Identification Number (RIN) made available to users (for inter-job management) or assigned to the job (for inter-process management). Within their jobs or processes, cooperating programmers relate a RIN to a particular resource through the statement structure of the job or process. When a job or process seeks exclusive access to a resource, it requests MPE to lock the RIN associated with the resource. The 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 others can lock it.

A RIN is not a physical entity, nor is it logically assigned to any resource. The association of a RIN and a resource is accomplished only by the structure of the code which operates the job or process which uses the RIN. The RIN number is always known to MPE, but the resource with which it is associated is not. For this reason, all cooperating programs must specify what RIN is associated with which resource.

Processes run by users with standard MPE capabilities can lock only one global RIN (used at the unrelated-process level) at a time. Users with Multiple RIN (MR) capability can lock more than one global RIN at a time.

Users with Multiple RIN (MR) capability, must avoid deadlocking, which occurs when two or more suspended processes cannot resume because they are mutually blocked. For example, you have two processes A and B, and two RINs 1 and 2. Each process attempts to lock both RINs in the following sequence: Process A tries to lock RIN 1 then RIN 2, and Process B tries to lock RIN 2 then RIN 1. Process A will lock RIN 1 then have to wait for RIN 2 to be released by Process B and Process B will lock RIN 2 and have to wait for Process A to release RIN 1 before it can continue. The processes are mutually blocked and thus a deadlock has occurred.

Deadlocks can be avoided by ranking. If the RINs had been ranked (for example, ranking order A=1; B=2), then Process B should have tried to lock RIN 1 before RIN 2. However, since Process A already locked RIN 1, Process B would have to wait until Process A had released RIN 1 and locked RIN 2 which in this example would be free. A WARMSTART is necessary to free the locked processes. Ranking also applies to processes and procedures.

Inter-Job Level (Global) RINs

The RINs used at the inter-job level are called "global" RINs. Global RINs prevent simultaneous access to a resource by two or more processes. Each global RIN is a positive integer which is unique within MPE. Global RINs are acquired and released through MPE commands, and locked and unlocked through MPE intrinsics.



Acquiring Global RINS .

Before users can engage in cooperative resource management through a RIN, one user must request the RIN and assign it a RIN password that enables all who know the password to lock the RIN. This is done with the **:GETRIN** command:

:GETRIN *rinpassword*

where *rinpassword* is a required password of up to eight alphanumeric characters, beginning with a letter, which locks the RIN.

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 the form:

RIN: *rinum*

The user who enters the **:GETRIN** command can use the RIN to lock and unlock the resource in the current session, or in future jobs and sessions. The RIN and its password are passed to other users to permit them to lock and unlock the resource in their jobs and sessions. All users do this by including the RIN as a reference parameter in intrinsic calls that lock and unlock resources. Users can use the RIN until the user who issued the **:GETRIN** command for that resource releases the RIN.

Note



MPE regards the user who issues the **:GETRIN** command as the owner of the RIN. Only the owner of a RIN can release it.

The total number of RINs that MPE can handle is specified when the system is configured, but can never exceed 1024.

Refer to the *MPE V Commands Reference Manual* (32033-90006) for details on the **:GETRIN** command.

Releasing Global RINS

The owner of a global RIN (the user who issued the **:GETRIN** command) can release the RIN to the RIN pool managed by MPE. Only the owner can release the RIN.

To release a RIN, enter the command:

:FREERIN *rin*

where *rin* is the RIN to be released.

Refer to the *MPE V Commands Reference Manual* (32033-90006) for details on the **:FREERIN** command.

Locking and Unlocking Global RINS

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 attempt by another process to lock this RIN fails. Note that depending on the call, the calling process may be suspended until the RIN is released.

To lock a global RIN, a user must know the *rinpassword* specified with **:GETRIN**, and its number returned by **:GETRIN**. Users with standard MPE capability can lock only one global RIN at a time.

The **LOCKGLORIN** intrinsic is useful when locking an entire file would inconvenience other users. For example, if several users are trying to access and update a large file simultaneously, one will succeed in locking the file and the others will be suspended until the file is unlocked. The **LOCKGLORIN** intrinsic, however, lets users lock a portion of a file so that other users are suspended less often.

The **UNLOCKGLORIN** intrinsic does not check whether the *rinnum* parameter specifies the most recently locked global RIN. When global RINs are locked and unlocked in any order by concurrent processes, deadlocks can occur. An effective way to avoid deadlocks is to assign a rank to each RIN which is used by all processes locking them.

Figure 3-8 shows a program using 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 chance of another user updating it simultaneously. In the program, the other users are not suspended when attempting to access records elsewhere in the file.

The file **BOOKFILE**, illustrated in Figure 3-9, contains the titles and status of the 20 books in a library. The program of Figure 3-8 will access this file.

BOOKFILE contains 20 records, so the program must acquire five RINs (the program uses four records per RIN). This is accomplished by issuing the command:

```
:GETRIN BOOKRIN
```

where **BOOKRIN** is the *rinpassword* specified in the program to lock the RIN. (See statements 00006000 through 00032000 in Figure 3-8.)

```

PAGE 0001    HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1989, 4:35 PM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1  BYTE ARRAY NAME(0:8):="BOOKFILE ";
00006000 00006 1  BYTE ARRAY PASSWD(0:7):="BOOKRIN ";
00007000 00005 1  INTEGER IN, OUT, BOOK, LGTH, ACCNO, RIN;
00008000 00005 1  LOGICAL DUMMY, COND:=TRUE;
00009000 00005 1  ARRAY BUFR(0:35);
00010000 00005 1  BYTE ARRAY BBUFR(*)=BUFR;
00011000 00005 1  ARRAY HEAD(0:13):="LIBRARY INFORMATION PROGRAM.";
00012000 00016 1  ARRAY REQUEST(0:7):=%6412,"ACCESSION NO: ";
00013000 00010 1  ARRAY CHANGE(0:9):="          NEW LOCATION: ";
00014000 00012 1  EQUATE RINBASE=2, RECDS'PER'RIN=4, MAXRIN=6;
00015000 00012 1  DEFINE CCL =IF < THEN QUIT#,
00016000 00012 1          CCNE=IF <> THEN QUIT#;
00017000 00012 1          INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,FREADDIR,FWRITE
00018000 00012 1          LOCKGLORIN,UNLOCKGLORIN,QUIT,BINARY;
00019000 00012 1  <<END OF DECLARATIONS>>
00020000 00012 1          IN:=FOPEN(INPUT,%45); CCL(1);          <<$STDIN>>
00021000 00012 1          OUT:=FOPEN(OUTPUT,%414); CCL(2);      <<$STDLIST>>
00022000 00024 1          BOOK:=FOPEN(NAME,%5,%304); CCL(3);    <<OLD DISC FILE>>
00023000 00037 1          FWRITE(OUT,HEAD,14,0); CCNE(4);      <<PROGRAM ID>>
00024000 00047 1  LOOP:
00025000 00047 1          FWRITE(OUT,REQUEST,8,%320); CCNE(5);<<REQUEST BOOK NO>>
00026000 00057 1          LGTH:=FREAD(IN,BUFR,-10); CCNE(6);    <<INPUT NUMBER>>
00027000 00070 1          IF LGTH=0 THEN GO EXIT;              <<NO INPUT-EXIT>>
00028000 00073 1          ACCNO:=BINARY(BBUFR,LGTH);          <<CONVERT NUMBER>>
00029000 00100 1          F <> THEN GO LOOP;                  <<IF BAD TRY AGAIN>>
00030000 00101 1          RIN:=RINBASE+(ACCNO/RECDS'PER'RIN);  <<COMPUTE RIN NO>>
00031000 00105 1          IF NOT(RINBASE<=RIN<=MAXRIN)
00032000 00117 1              THEN GO LOOP;<<BNDS CHK RIN>>
00033000 00123 1          LOCKGLORIN(RIN,COND,PASSWD);          <<LOCK FILE SUBSET>>
00034000 00135 1          FREADDIR(BOOK,BUFR,36,DOUBLE(ACCNO));CCL(7);
00035000 00136 1              <<READ DATA>>
00036000 00146 1          IF > THEN GO AGAIN;                  <<EOF, TRY AGAIN>>
00037000 00156 1          FWRITE(OUT,BUFR,36,0); CCNE(8);      <<DISPLAY DATA>>
00038000 00161 1          FWRITE(OUT,CHANGE,10,%320); CCNE(9);
00039000 00167 1              <<REQUEST A CHANGE>>
00040000 00201 1          BUFR(19):=" ";
00041000 00201 1          MOVE BUFR(20):=BUFR(19),(16);        <<BLANK OLD LOCN>>
00042000 00201 1          LGTH:=FREAD(IN,BUFR(19),17); CCNE(10);<<READ NEW LOCN>>
00043000 00201 1          IF LGTH>0 THEN                      <<NEW LOCN ENTERED>>

```

Figure 3-7. (Part 1)Using the LOCKGLORIN and UNLOCKGLORIN Intrinsics

```

00041000 00204 1      BEGIN
00042000 00204 2      FWRITEDIR(BOOK,BUFR,36,DOUBLE(ACCNO));<<MODIFY FILE>>
00043000 00213 2      CCNE(11);                                <<CHECK FOR ERROR>>
00044000 00216 2      END;
00045000 00216 1      FCONTROL(BOOK,2,DUMMY); CCL(12);      <<FORCE RECD POST>>
00046000 00225 1 AGAIN:
00047000 00225 1      UNLOCKGLORIN(RIN); CCNE(13);          <<UNLOCK SUBSET>>
00048000 00232 1      GO LOOP;                                <<CONTINUE>>
00049000 00234 1 EXIT:END.

```

Figure 3-7 (Part 2) 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: THE LIFE OF MERENB	LOCN: AVAILABLE
TITLE: INTRO TO TAI CHI CHUAN	LOCN: AVAILABLE
TITLE: TOM SAWYER	LOCN: AVAILABLE
TITLE: TREASURE ISLAND	LOCN: AVAILABLE
TITLE: A CHRISTMAS CAROL	LOCN: AVAILABLE
TITLE: THE WIZARD OF OZ	LOCN: AVAILABLE
TITLE: THE DARK CRYSTAL	LOCN: AVAILABLE
TITLE: SPEED RACER	LOCN: AVAILABLE
TITLE: ULTRAMAN GOES TO TOWN	LOCN: AVAILABLE
TITLE: H.M.S. PINAFORE	LOCN: AVAILABLE
TITLE: FEAR OF FLYING	LOCN: AVAILABLE
TITLE: SNOW WHITE	LOCN: AVAILABLE
TITLE: DR. DOOLITTLE	LOCN: AVAILABLE
TITLE: TALES OF MOTHER GOOSE	LOCN: AVAILABLE
TITLE: AESOP'S FABLES	LOCN: AVAILABLE
TITLE: THE GULAG ARCHIPELAGO	LOCN: AVAILABLE

Figure 3-8. BOOKFILE

The program in Figure 3-8 establishes the RIN number limits 2 and 6 (statement 00014000), thus using only RINs 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 RINs in sequence, and because the program wants consecutive RINs to keep track of them more easily, it may be necessary to enter more :GETRIN commands before the program is run to acquire the five consecutive RINs. Extra RINs can be released with the :FREERIN command.

The statement:

```
FWRITE(OUT,REQUEST,8,%320);CCNE(5);
```

requests a book number from the user and performs a condition code check. Note that in statement 00016000 Figure 3-8, CCNE has been defined as:

```
IF <> THEN QUIT#;
```

This eliminates the need to repeat the entire statement at every point in the program where a condition code check is required. Instead, the statement CCNE and an arbitrary number (5) 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 to be locked is computed with the statement:

```
RIN:=RINBASE+(ACCNO/RECDS'PER'RIN);
```

RINBASE and RECDS'PER'RIN have been equated to 2 and 4, respectively (see statement 00014000 of Figure 3-8). Thus, if a book number 3 is entered by the user, the RIN number to be locked would be RIN 2, computed in the following way (using integer division):

$$RIN = 2 + (3/4)$$

$$= 2 + 0$$

The record specified by the book number is displayed for the user and the "NEW LOCATION:" is requested. The existing location information is filled with blanks with statements 00037000 and 00039000:

```
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:

```
FWRTEDIR(BOOK,BUFR,36,DOUBLE(ACCNO));
```

In case the opened file is a buffered file, the statement:

```
FCONTROL(BOOK,2,DUMMY);
```

ensures that the process buffers are posted to disk before the RIN is unlocked.

In this type of program, it is important that the number of records per block be equal to the number of records per RIN. 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 will be computed and locked.

When **RETURN** is pressed, signifying no input, the program terminates.

The results of the program in Figure 3-8 are shown below:

:RUN LIBIN

LIBRARY INFORMATION PROGRAM.

ACCESSION NO: 3

TITLE: JUNGLE BOOK LOCN: AVAILABLE

NEW LOCATION: FACULTY LOAN - DR. JOHNSON

ACCESSION NO: 10

TITLE: THE DARK CRYSTAL LOCN: AVAILABLE

NEW LOCATION: LOANED CARD# 451, DUE APRIL 1

ACCESSION NO: 3

TITLE: JUNGLE BOOK

LOCN: FACULTY LOAN - DR. JOHNSON

NEW LOCATION:

ACCESSION NO: 9

TITLE: THE WIZARD OF OZ LOCN: AVAILABLE

NEW LOCATION: INTERLIBRARY LOAN - UNIV. OF OZ

ACCESSION NO: 3

TITLE: JUNGLE BOOK

LOCN: FACULTY LOAN - DR. JOHNSON

NEW LOCATION: AVAILABLE

ACCESSION NO: **RETURN** , ,

END OF PROGRAM

:

Interprocess (Local) Level RINs

Interprocess RINs are called local RINs. Local RINs are used to prevent simultaneous access of a resource by two or more processes within the same job. Each local RIN is a positive integer that is significant to processes within the job only.

Local RINs are assigned, managed, and released with the **GETLOCRIN**, **LOCKLOCRIN**, and **FREELOCRIN** intrinsics.

Acquiring Local RINs

Like global RINs, local RINs must be acquired by the user before they can be used by the processes within a job. For example:

```
GETLOCRIN(6);
```

would acquire six local RINs, 1 through 6. Multiple RIN (MR) capability is not required and it is the user's responsibility to avoid deadlocks.

Locking and Unlocking Local RINS

Any local RIN assigned to a job can be locked, by one process at a time, by using the **LOCKLOCRI**N 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 i

For example, to lock RIN number 6 (acquired by **GETLOCRI**N) unconditionally, the following call could be used:

```
LOCKLOCRI(6,COND);
```

Unconditional locking is denoted when bit (15:1)=1 of the logical word **COND**. If bit (15:1)=0, locking will take place only if the RIN is immediately available.

To unlock the same RIN, the intrinsic call:

```
UNLOCKLOCRI(6);
```

could be used. The call above 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 **LOCKLOCRI**N and **UNLOCKLOCRI**N intrinsic calls are used, consider two processes, a father and its son, within a job:

FATHER PROCESS	SON PROCESS
.	.
.	.
.	.
LP:=FOPEN(LIN, ...);	LP:=FOPEN(LIN, ...);
.	.
.	.
.	.
GETLOCRI(3);	LOCKLOCRI(1,TRUEVAL);
FWRITE(LP, ...);	FWRITE(LP, ...);
	.
LOCKLOCRI(1,TRUEVAL);	.
CREATE(DSCEND, ...);	.
FWRITE(LP, ...);	FWRITE(LP, ...);
.	.
.	.
.	.
UNLOCKLOCRI(1);	UNLOCKLOCRI(1);
.	.
.	.
.	.

Suppose the father and son processes wanted to use RIN 1 to manage the line printer (designated as LP) so that the son process could never use the

printer while the father is using it. This is coded in the examples above. When the father first references LP, the son has not been created and the printer need not be locked. But just before creating the son, the father locks the printer RIN. The father issues all of its print requests before unlocking the printer. Before the son accesses the printer, it tries to lock it, fails, and is suspended. When the father unlocks the printer, the son locks it and issues print requests.

Identifying Local RIN Owners

LOCRINOWNER identifies at any time the PIN of the process that has a particular local RIN locked. This is useful, for example, when father and son processes are being synchronized through calls to the **ACTIVATE** and **SUSPEND** intrinsics.

Consider the following example in which a father process acts as a monitor for several son processes. The father waits in a suspended state 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** determines the PIN of the son that activated the father, since that son will have the **WHICHSONRIN** RIN locked. The father then performs its required duty. The father finally activates the son that activated the father, and the father suspends, releasing the synchronization RIN, and waiting for another son to activate it again. An example of Process Synchronization with **LOCRINOWNER** is:

<pre> 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; . . . </pre>	<pre> SON PROCESS . . . LOCKLOCRIN (whichsonrin) LOCKLOCRIN (synchrin); ACTIVATE (father); SUSPEND (waitforfather,synchrin); UNLOCKLOCRIN (whichsonrin); . . . </pre>
---	--

Freeing Local RINS

To free all local RINs currently reserved for your job, enter:

```
FREELOCIN;
```

User Logging

The MPE user logging facility provides a structure for documenting additions and modifications to user data bases and subsystem files. Documentation can be recorded on disk, tape, serial disk, and cartridge tape. Logging is done programmatically by means of the following intrinsics, which you must have user logging (LG) or system supervisor (OP) capabilities to use:

- **LOGSTATUS** displays status information about currently opened log files.
- **OPENLOG** provides access to a logging facility.
- **WRITELOG** writes journal entries to a logging file. This creates a copy of any modifications to a data set.
- **CLOSELOG** closes access to the logging facility.
- **FLUSHLOG** writes the contents of the user logging memory buffer to the disk destination file (disk buffer file when logging to a serial device). This intrinsic writes no special records.
- **BEGINLOG** posts a special record to the user logging file to mark the beginning of a logical transaction in the log file.
- **ENDLOG** posts a special record to the user 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.
- **LOGINFO** displays status information about opened log files in the data set.

If a data set is lost, the user logging file can be used with a copy of the data set file to recover the lost transactions.

How User Logging Works

When requesting serial files within the logging facility, entries are placed in the buffer area of the logging data segment. Once this buffer area is full, the contents are written to the disk buffer file on disk. (See Figure 3-10.) If there is no space available in the memory buffer, the result depends upon the *mode* parameter specified:

- If *mode*=0 (WAIT) is specified, the process is suspended until the logging process writes the contents of the memory buffer to disk. Then, the process is reactivated (similar to WAITIO). Space becomes available, and your request is moved into the memory buffer.
- If *mode*=1 (NOWAIT) is specified the communications area of the data segment sends an error message indicating the transaction has not been completed. The user needs to resubmit the request (similar to NOWAITIO).

Effective with the G.02.00 release the user can have the system automatically write a changelog record and close the user logging file when

it becomes full by specifying the *auto* parameter of the **:ALTLOG** or **:GETLOG** command. A new, permanent user logging file is then opened, which has the same characteristics as the previous file. It is important to note that user logging files can be changed automatically only when the storage media is disk, not tape. To use the *auto* parameter of the **:ALTLOG** and **:GETLOG** commands the filename of the existing user log file must end with a three digit numeric (e.g. **fname001**). Then when the new file is constructed the portion preceding the three digits remains unchanged and the last three digits are incremented by one (i.e. **fname002**). Refer to the *MPE V Commands Manual* (32033-90006) for additional information on the *auto* parameter of the **:ALTLOG** and **:GETLOG** commands.

The buffer area in "LOGBUFF" has a maximum capacity of 4K words (32 records). This buffer area will be posted, entered, or transferred to the disk buffer file (when logging to a serial device) or the disk destination file (when logging to disk) if:

- A user calls **FLUSHLOG**, **BEGINLOG**, **ENDLOG**, or **WRITELOG** in *mode=2*. (Refer to the **WRITELOG** intrinsic in Chapter 2 for information on *mode=2*.)
- There is not enough space in the buffer for another record.
- The process is interrupted at any time.

The logging process and logging buffer function differently according to the type of file specified by the user.

When a serial log file is specified, the logging process acts as an interface between the logging buffer file and the serial log file by writing records to the serial log file requested. This process is independent of your process. The user can add records to the logging memory buffer at the same time the logging process is writing records from the disk buffer to the serial device. As soon as the logging process has made space available in the logging buffer file, the user's process will be reactivated.

LOG RECORD AT OPENLOG:

0	2	3	4	6	7	11	12	24	25	127
rec#	cksum	code	time	date	logid	log#	creator	pcb		

LOG RECORD AT WRITELOG:

0	2	3	4	6	7	8	9	127
rec#	cksum	code	time	date	log#	len	user area	

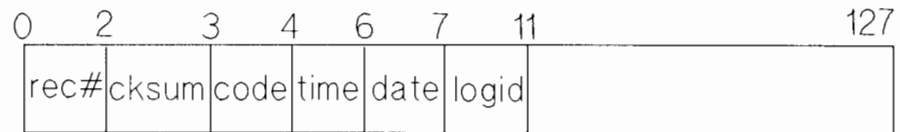
LOG RECORD AT CLOSELOG:

0	2	3	4	6	7	11	12	24	25	127
rec#	cksum	code	time	date	logid	log#	creator	pcb		

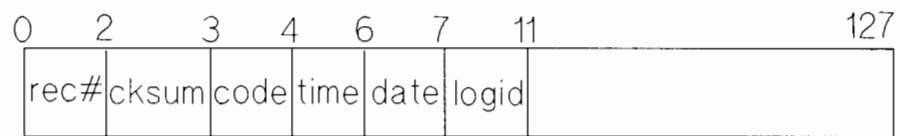
CRASH MARK:

0	2	3	4	6	7	127
rec#	cksum	code	time	date		

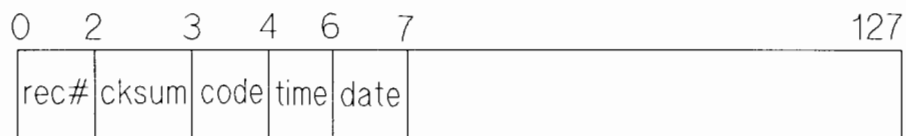
HEADER RECORD (START/RESTART):



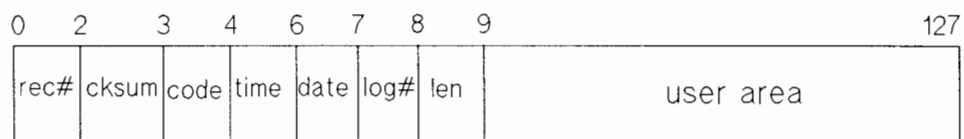
TRAILER RECORD (STOP):



NULL RECORD:



BEGIN TRANSACTION MARKER:



END TRANSACTION MARKER:

0	2	3	4	6	7	8	9	127
rec#	cksum	code	time	date	log#	len	user area	

LOG RECORD AT CHANGELOG:

0	2	3	4	6	7	11	12	14
rec#	cksum	code	time	date	logid	seq num	c-time	c-date
15	33	34	52	53	71	72	127	
f-file-name	f-type	p-file-name	p-type	c-file-name	c-type			

CODE DEFINITION:

Code =

- 1 Open log record
- 2 User/subsystem record
- 3 Close log record
- 4 Header record
- 5 Trailer record
- 6 Restart record
- 7 Continuation of user or subsystem record
- 9 Crash marker
- 10 End transaction record
- 11 Begin transaction record
- 12 Change log record (resides in new file; points to old file)
- 13 Change log record (resides in old file; points to new file)

(SPACE) Null record

Note



If Code = 12 *p-file-name* = previous file in the set, *p-file-name* = previous file type in the set

If Code = 13 *p-file-name* = next file in the set *p-file-name* = next file type in the set

DATA FIELDS OF LOG RECORDS:

REC#	= Double Integer
CKSUM	= Integer
CODE	= Integer
TIME	= Double Integer (from CLOCK intrinsic)
DATE	= Integer (from CALENDAR intrinsic)
LOGID	= ASCII
LOG#	= Integer
LEN	= Integer
USERAREA	= ASCII
CREATOR	= ASCII
PCB	= Integer
SEQ NUM	= Integer
C-DATE	= Double Integer
C-TIME	= Double
F-FILE-NAME	= ASCII
P-FILE-NAME	= ASCII
C-FILE-NAME	= ASCII
F-TYPE	= Integer
P-TYPE	= Integer
C-TYPE	= Integer

The code in Word 3 of each logging record identifies the type of record. For example, a “1” in the second half of the third word indicates an **OPENLOG** record.

Privileged users can define a subsystem code in the first half of the logging record code word bits (0:8). This code is passed in the *index* parameter of the “OPENLOG” intrinsic.

The checksum (CKSUM) algorithm uses the Exclusive-Or (XOR) function against a base of negative one. The null record is used as a filler.

The length field (LEN) contains the number of words in the entire transaction (i.e. the length passed to **WRITELOG**, **BEGINLOG**, or **ENDLOG**). If a continuation record is part of the transaction, that record will also contain the data length. For example, if a length of 140 (words) is passed to the intrinsic, the LEN field will contain 140. Since the user area will only accommodate 119 words the remaining 21 words will be stored in a continuation record. However, the LEN field of the continuation record will indicate the total number of words in the transaction (140 in this example). A positive number indicates words; a negative number refers to bytes.

User Logging Procedures

To utilize the user logging facility, the programmer uses the logging intrinsics in the applications program to write data to a logging file to be used for recovery. The user must also write a recovery procedure program that can read the log file and apply it to a backup copy of the data set to recover data lost in a system failure. User logging does power failure recovery automatically.

The suggested user logging procedure includes the following steps:

1. Select a logging identifier on the system using **:GETLOG**. If data security is necessary, use passwords/ lockwords provided in the **:GETLOG** command. Specify the configured device class name, i.e. **DISC**, **TAPE** for magnetic tape, **SDISC** for serial disk device, and **CTAPE** for cartridge tape unit (such as HP 9140/9144). These must be configured device class names or an **FOPEN** failure will result.
2. Design the application and data structures.
3. Decide what information must be logged in order to recover the data structures of the application. The log file will be applied to the backup copy of the data structure to recover data lost because of a system failure or power failure. Include the appropriate calls to the user logging intrinsics in your application program to log the data necessary for recovery.
4. Design and program the recovery procedure. The backup copy of the application data structure is necessary to the recovery procedure. The recovery program must recognize the user logging file record formats.
5. If the logging identifier specified in the **:GETLOG** command is associated with a disk file, build that file with the correct code (**;CODE=LOG**), and enough disk space to contain one day's output. Be generous with the disk file space. Divide the file into several extents; allocate only the first extent. This minimizes the impact of a large file on the system. The user logging process allocates additional extents when necessary (one at a time).
6. Have the System Operator start the logging process for your logging identifier. Refer to the *MPE V Commands Reference Manual* (32033-90006) for more information on the **:LOG** command.
7. If there are changes in the data sets, run your logging application program for the user logging file you specified in the **:GETLOG** command.

If it becomes necessary to recover the data sets, restore the backup copy that was saved and run the recovery procedure to incorporate any changes made to the data structures.

Suggested Log File Uses

Use of a log file starts with your application, especially if many users are accessing the same user log file. Set up a log file separate from your data base log file with information which points to your entry. The separate log file might include the

- User, group, and account names you are using in your application. This can be determined by using the **WHO** intrinsic.
- Job/session input device number, which can also be determined by using the **WHO** intrinsic. This number provides only a history which does not help directly in recovery.
- Process identification number (PIN) of the process accessing the log files. Use the **GETPROCID** intrinsic to obtain the PIN.
- Date and time of opening the 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.
- The fully qualified file name of the data set log file being accessed, including the ASCII code of the **LOGID**.

To recover the file listing, read the additional file and open the log file. Search the log file for the **LOGID** and the creator that match in the **OPENLOG** record CODE 1. Verify it 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.

Section Divider

4. Accessing and Altering Files



Accessing and Altering Files

This chapter provides background information on the MPE file system and some typical operations performed with the MPE file system intrinsics. For more information on the MPE file system, refer to the *MPE File System Reference Manual* (30000-90236)

- Open files with **FOPEN**.
- Parse a file designator and validate that it is syntactically correct with **FPARSE**.
- Request access and status information about a file with **FFILEINFO** and **FGETINFO**.
- Request file error information with **FCHECK**.
- Read records (or a portion of a record) from a file with **FREAD** and **FREADDIR**.
- Write a record (or a portion of a record) to a file with **FWRITE** and **FWRITEDIR**.
- Move a specific record from a file into a buffer preparatory to reading the record to the stack with **FREADSEEK**.
- Initiate completion operations for an I/O operation with the **IOWAIT** intrinsic.
- Read a file label on a labeled magnetic tape file with **FREADLABEL**.
- Write a file label on a labeled magnetic tape file with **FWRITELABEL**.
- Obtain information from the file label of a disk file with **FLABELINFO**.
- Read a user-defined label from a disk file with **FREADLABEL**.
- Write a user-defined label onto a disk file with **FWRITELABEL**.
- Update a record on a disk file with **FUPDATE**.
- Space forward or backward on a disk or tape file with **FSPACE**.
- Reset the logical record pointer to any logical record in a fixed-record length disk file with **FPOINT**.
- Perform control operations on a file (or the device on which the file resides) with **FCONTROL**.
- Activate and deactivate access mode options with **FSETMODE**.
- Rename an open disk file with **FRENAME**.
- Determine if an input file and a list file are interactive and/or duplicative with **FRELATE**.
- Coordinate access to shared files with **FLOCK** and **FUNLOCK** intrinsics.
- Close a file with **FCLOSE**.

File Device Relationships

Devices required by files are allocated by MPE. You can specify these devices by class (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). 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. This is the name that is coded into the program, along with the program's specifications for the file. For more information on formal file designators refer to the *MPE File System Reference Manual* (30000-90236). Actual subsystem formal file designators can be found in the *MPE V Commands Reference Manual* (32033-90006). MPE then determines the device on which the file resides, or its disk 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.

Non-Sharable Device Access

Any device which can only handle one set of data at a time is a non-sharable device. The specification of a device by class 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 System Operator if necessary. After the Operator answers, **FOPEN** continues execution. (The device specification is ignored when **\$STDIN**, **\$STDINX**, or **\$STDLIST** is 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 System 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 (**MULTI**) or Inhibit Buffering (**NOBUF**) if records must be transmitted in the same order as requested. In this instance **MULTI** is preferred.

File Domains

The set of all permanent disk files in MPE is known as the system file domain. Within this system file domain, files are assigned to accounts and accounts are then organized into groups. You logon 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 logon, but thereafter (if the default MPE file security provisions are in effect) you will have unlimited access to any file within your logon or home group. (However, if the file is protected by a lockword, you must know the lockword.) You can also read and execute programs residing in any file in the **PUB** group of your account, and in the **PUB.SYS** group/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 logon. Note that once you logon, you do not need to know the passwords for other accounts and groups to access files assigned to them. You only need to know the account and group names. If any of these files are protected by a file lockword, you do need to know this lockword.

For every job or session running in the system, MPE recognizes an accompanying file domain, called the "job file domain" or "session file domain". This domain contains all temporary files opened and closed within the job or session without being saved (that is, declared as 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 new files, not saved temporary or permanent files when closed).

When a new file is opened it is known only to the program that creates it, and will exist only while the program is being executed. At this time, the file name assigned by you need not be unique. But if the program tries to save the file as permanent or job temporary (via `FCLOSE`), MPE determines whether another file with the same 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 chapter, with the exception of **FOPEN**, **FGETINFO**, **FFILEINFO**, **FDEVICE-CONTROL**, and **FRENAME**, can be called with the DB register pointing to a data segment other than the calling process' stack (split-stack mode). 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.

Opening a File

Before a user process can read, write, or otherwise manipulate a file, the process must initiate access to that file by opening it with the **FOPEN** intrinsic call. This call applies to files on all devices. When the **FOPEN** intrinsic is executed, it returns the file number used to identify the file in subsequent intrinsic calls to the user process.

If the file is opened successfully (indicated by the CCE condition code), the file number returned is a positive integer. If the file cannot be opened (indicated by 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. This uses two file numbers. 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 device on which the file resides.
- Allocating the device on which the file resides to your process. If the file resides on a non-sharable device, such as magnetic tape, and the user has Non-sharable Device (ND) capability, **FOPEN** determines whether the System Operator must approve allocation of the device, such as an unlabeled magnetic tape, or provide a particular media, such as a specific volume for a labeled magnetic tape request or special forms for a line printer. If so, **FOPEN** requests the System Operator to respond appropriately. Disk files generally can be shared concurrently among jobs and sessions. Magnetic tape and unit-record devices are generally allocated exclusively to the requesting job or session. For example, different processes within the same job may open and have concurrent access to a file on the same magnetic tape or unit-record device, if the file has been opened with **MULTI**-access. However, this device cannot be accessed by another job until all accessing processes in this job have issued a corresponding **FCLOSE** call.

- 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 user labels (for files on disk). For new files on disk, **FOPEN** specifies the number of user labels to be written.
- Allocating to the file the number of extents initially requested (for new disk 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 disk. This information overrides information from any other source. Label formats are presented in the *MPE File System Reference Manual* (30000-90236).
 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 is only true, if file equations were not disallowed.
 4. The parameter list of this **FOPEN** intrinsic call.
 5. System default values provided by MPE (when values are not obtainable from the above sources).

When information from one of these five sources conflicts with that from another, preempting takes place according to the order of precedence shown above. To determine the specifications actually taking effect, the user can call the **FGETINFO/FFILEINFO** intrinsics, described later in this chapter. (Notice that certain sources do not always apply or convey all types of information. For example, 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 disk file, the **FOPEN** call will specify whether the file is an old or new file; an old file is an existing 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 terminals and magnetic tape units, follow the old/new specification in your **FOPEN** call.

When a job attempts to open an old file on a non-sharable device, MPE searches for the file in the Input Device Directory (IDD). If the file is not found, a message is transmitted to the System Console requesting the Operator 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**, **\$STDINX**, 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 usually required. The Operator must make the tape available, unless the tape is already mounted and recognized by MPE, it is auto-allocating, or the device is being reopened by the same or related process.

How To Use Files

The remainder of this chapter 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 restating the purpose of each intrinsic as discussed in Chapter 2.

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 directly from your program. The compilers for other languages, such as **FORTRAN** and **COBOL**, emit code which calls **FOPEN** and opens the file for you. In any case, however, whether called explicitly by your program, or called for you by the language's compiler, 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 disk file. (A new disk 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 comprises a number of control blocks that are created, and which contain information about the file. In addition, if the new disk file is opened with buffered access, a buffer space is allocated in the Access Control Block (ACB) to contain the number of records per block that the user has specified in the **FOPEN** call. The buffer space is large enough to hold a block of information to be sent to the disk.

The control blocks are 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. The next thing that occurs is that file space is allocated to the file. On each system disk or private volume disk there is a table of free space managed 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. Upon the successful completion of an **FOPEN** call, an integer value is returned to the calling program as the file number. This integer value is an index into the AFT, and the appropriate AFT entry in turn then points to the control block that belongs to this particular file.

Figure 4-1 shows the stack and the AFT entry pointing to the control block. The control block contains buffers, in this case, enough room for three logical records. In the example shown in Figure 4-1, each record is 80 bytes and the records are grouped into a block of three.

A partial list of items contained in a disk 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 and domain bits).
- Number of extents.
- Extent size.
- File code.

The linkage, then, goes from the user's stack, via the AFT, to the control block; from the control block there is a pointer to the label on the disk itself. In the simplified example of Figure 4-1 the file label is shown on the system disk. However, it could be on any disk 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.

If the **FWRITE** intrinsic was called to write a single 80-byte record, that record would be moved from the user's stack to position number 1 in the buffer. As soon as that physical move from the stack to the buffer is complete, the **FWRITE** is also complete as far as the program is concerned. However, no actual write to the disk takes place. An **FWRITE** call to write record number 2 would consist of a similar move from the stack position number 2 in the buffer. Subsequently, record number 3 would occupy position 3 in the buffer. Immediately after the buffer becomes full, when the third record has been moved to the buffer, the entire block of information is then transferred to the disk. Thus, when a file is accessed in a buffered manner, records actually are moved from the stack to the buffer. Then, when the last record in a block has been moved to the buffer, a physical write of a complete block occurs. That is, a whole block (in this case three records) is transferred to the system disk.

At **FCLOSE** time, the access interface is dismantled. If the file is a new file, you must now decide whether you want the file to remain in the system as a permanent file, as a job/session temporary file, or whether you want the file to be deleted from the system. 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 file system in the Access Control Block (ACB), is posted in the system directory along with your logon account and the group you have specified. 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 disk at a certain sector address. Referring to Figure 4-1, you can see that in the system directory there will be an entry that includes the file name and some sector address, for example, %123. The sector address %123 then points to the first extent which includes the label.

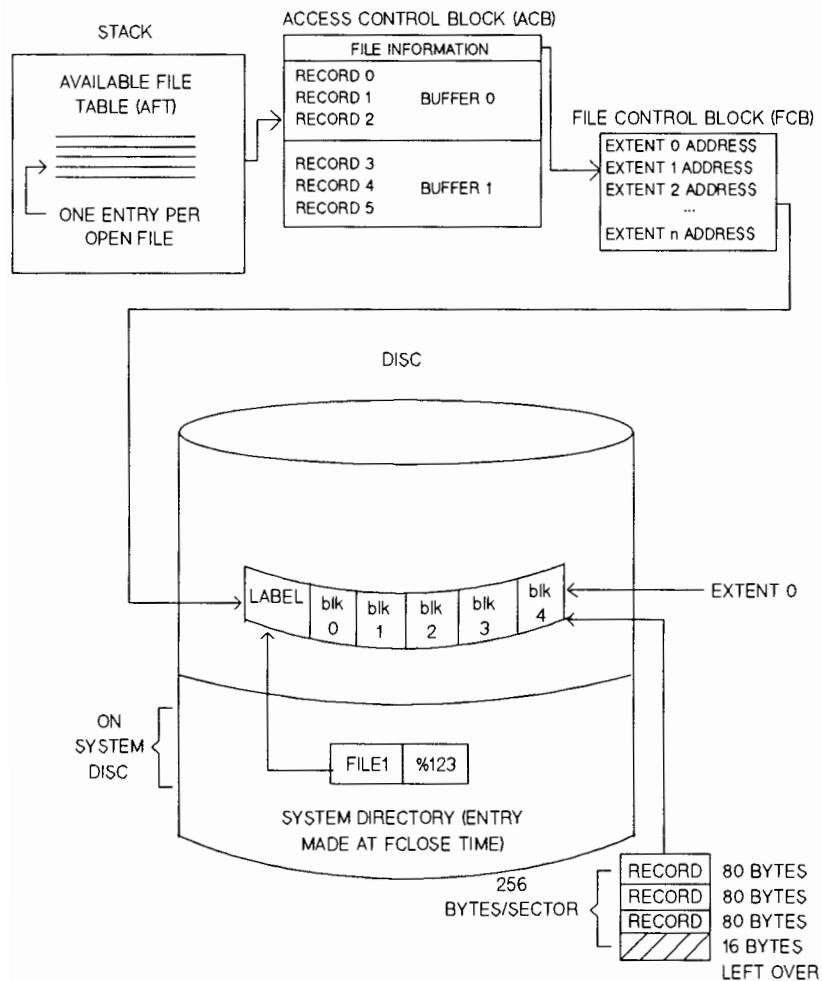


Figure 4-1. File Access Interface for New Disk Files

As soon as the name of the file, pointer to the file label, and accounting information such as creation date, have been entered in the system directory the file access interface is dismantled. The control blocks and buffers are deleted from the system and the entry in your stack in the Available File Table (AFT) is removed.

When the **FCLOSE** operation is complete, the disk file becomes a permanent file in the system. It can now be opened as an old disk file. If a file with the name of **FILE1** already exists in your logon account and specified group, this will not be noticed until **FCLOSE** is called, and, at that time, results in an error.

Figure 4-2 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. The resulting operation would be similar to the previous example with one exception: since an existing file is being opened, it is not necessary to allocate disk space for it. It is necessary for the system to establish a mechanism for the user to access the disk file. In other words, the system makes an entry in the Available File Table (AFT), creates a new Access Control Block (ACB) and File Control Block (FCB), pointing to the existing file on the disk.

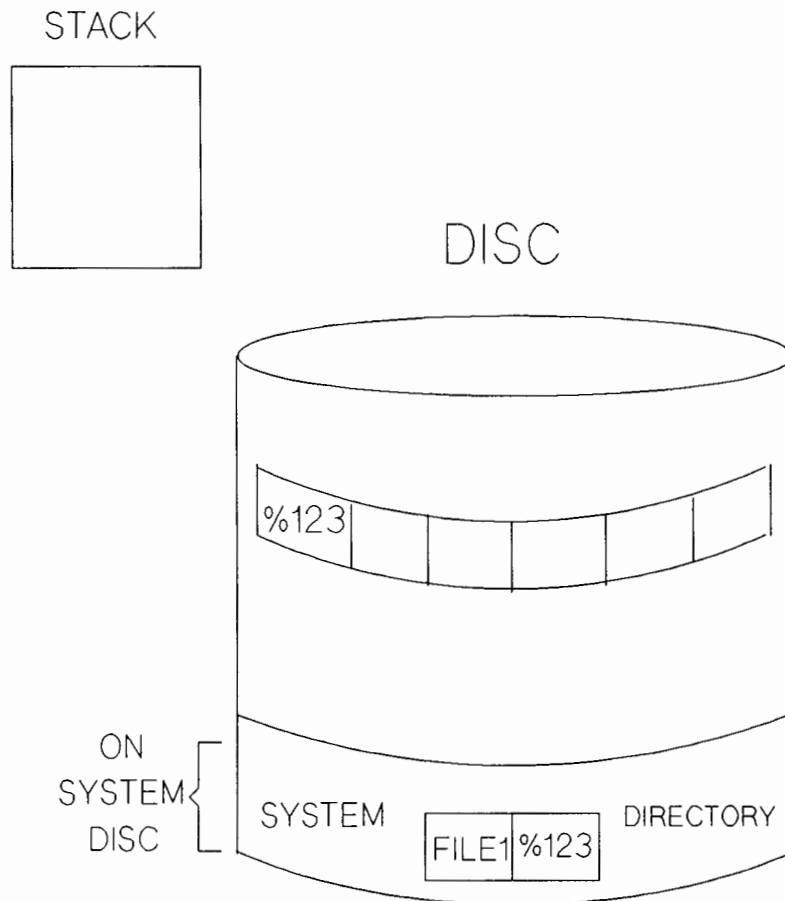


Figure 4-2. Directory File Name and Label Address Pointer

Figure 4-3 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; MPE searches the system directory under the appropriate account and group for this file.

Once the file is found, MPE establishes the access mechanism, consisting of an ACB and FCB as before, and a map from the FCB of the extent address on disk.

The other type of old file is the job or session temporary file. One of the differences between a job/ session 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.

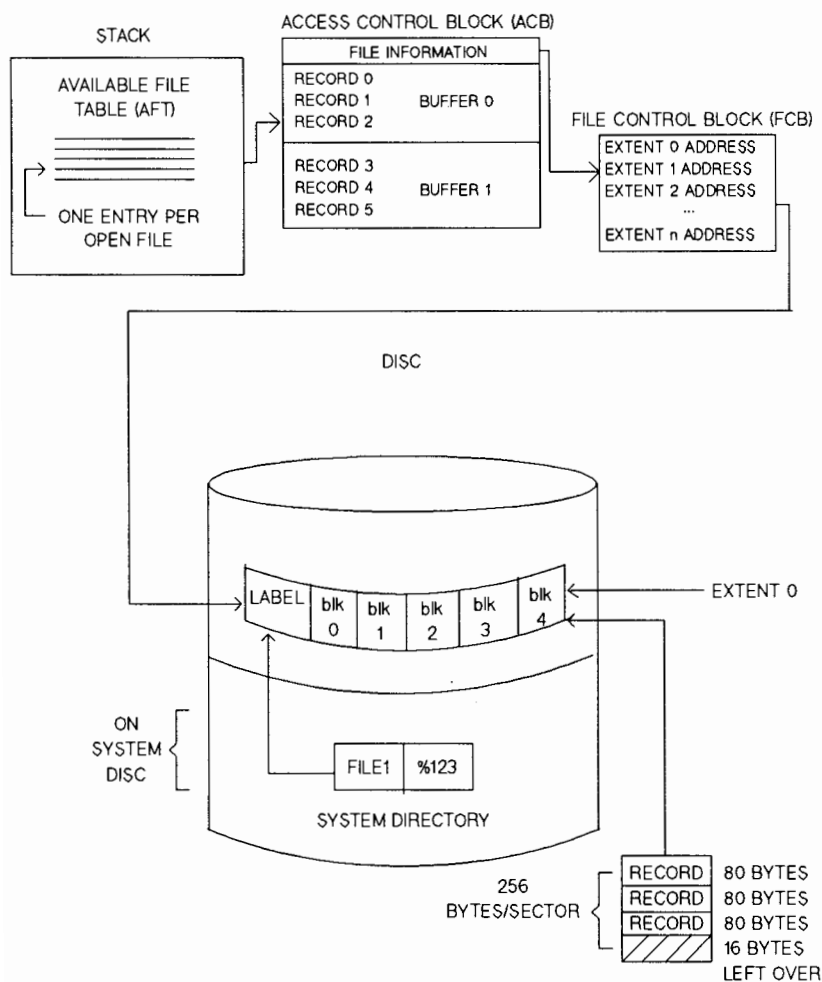


Figure 4-3. File Access Interface for Old Disk Files

File characteristics are obtained from different sources, depending on whether the file is a new disk file, an old disk file, or a file on a device other than disk.

For a new disk file, created with **FOPEN** *options* bits (14:2)=00, the characteristics are obtained from **FOPEN** intrinsic call parameters and

defaults. These characteristics may be overridden by **:FILE** command parameters. The disk file and the file label are created according to the characteristics specified above. (This label remains with the file during its entire existence on the system.)

Old permanent disk files are opened with **FOPEN** *foptions* bits (14:2)=01. Old temporary disk files are opened with **FOPEN** *foptions* bits (14:2)=10. In either case the file characteristics are obtained from the disk file label, **FOPEN** intrinsic parameters and default conditions. Some of these characteristics may be overridden by **:FILE** command parameters. The disk file label may override either of the above specifications. If *foption* bits (14:2)=11 the existing file can be either an old temporary or an old permanent file with the Job Temporary File Directory searched first.

When a file is opened on a device other than disk, the file characteristics are established by **FOPEN** intrinsic call parameters and defaults. Any **:FILE** command parameters will override these specifications and defaults. However, any device-dependent restraints imposed by the file system will override both of the above settings. If you have the Non-Sharable Device (ND) capability, the file system allows you to open a physical I/O device in the same manner as you would open a disk file. (Disks are the only devices which are considered by MPE to be sharable among several users.) When a non-sharable device has been opened by **FOPEN**, 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 blocking factor of one. A summary of these restrictions is presented in Table 4-1.

It should also 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 disk in transit between the physical devices and the user program. Because data can be temporarily buffered in a disk 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 devicefile are insignificant to the program.

One exception is applications which write large reports. Because a spoolfile is a disk file, it has a maximum size of 32 extents, each of which is of a fixed configured size. This size may be too small and therefore, all extents may become filled before the job is finished. As a result, data may be lost. The solution is to have the system manager raise the number of sectors per spoolfile extent.

The system manager reconfigures the extent size in the **INITIAL/SYSDUMP** dialog. The appropriate question is "**# OF SECTORS PER SPOOLFILE EXTENT**". A configured extent size of 384 sectors means the spoolfile has room for approximately 25,000 lines.

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 rest of the program is still running.

When a non-sharable devicefile is opened, the device has to be allocated by the system so that the calling process can access the file. MPE classifies devices as "NEW or OLD", "OLD only", or "NEW only", depending on the device type. Table 4-2 shows the manner in which devices are classified. Included in Table 4-2 is the device name, its device type (in octal), and whether it is considered to be OLD/NEW, OLD only, or NEW only. If **FOPEN** specified a device type that is considered by MPE to be output only, MPE considers this to be a NEW file. Normally, NEW files do not require special attention. If the device is available, it will be allocated to the user. Printer forms message, plotter, or magnetic tape requests are the exceptions, however, and require Operator intervention.

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)
Blocking factor = 1
Domain = 1 (OLD permanent)
If not ASCII, then NOBUF
If FOPEN aoptions access type = 1,2,3, then FOPEN fails (access
violation)

INPUT/OUTPUT DEVICES (PARALLEL)

Terminals
ASCII
NOBUF
Undefined-length records
Blocking factor = 1

INPUT/OUTPUT DEVICES (SERIAL)

Magnetic Tape Drive/Serial Disk Drive/Card Reader/Punch
No restriction

OUTPUT ONLY (SERIAL)

Line Printer/Card Punch/Paper Tape Punch/Plotter
If Line Printer, ASCII only
Undefined-length records
Blocking factor = 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

Figure 4-4. Device-Dependent Restrictions

Table 4-1. Classification of Devices

DEVICE NAME	DEVICE TYPE NUMBER (OCTAL)	CLASSIFICATION
Moving-Head Disk	00	NEW or OLD
Flexible Disk	02	NEW or OLD
7911,7912,7914,7933, or 7911/7912 Integrated Cartridge Tape Unit and 7945 Disk, and 9144 Cartridge Tape Unit.	03	NEW or OLD
Card Reader	10	OLD only
Paper Tape Reader	11	OLD only
Terminals, DSN/DS pseudo terminals, NS/3000 pseudo devices, multipoint terminals, multipoint supervisor, asynchronous terminal controller.	20	NEW or OLD
Intelligent Network Processor (INP)	21	NEW or OLD
Printing Reader Punch	24	NEW or OLD
DSN/MRJE Pseudo Device	26	NEW or OLD
Magnetic Tape Drive	30	NEW or OLD
Line Printer	40	NEW only
Card Punch	41	NEW only
Plotter	43,44,45	NEW only

The flowchart shown in Figure 4-4 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 predefined input file, for example, a file identified with the **:DATA** command. If no such file is found, MPE sends a message to the System Operator asking for the logical device number of the input device.

If an input/output type of device is specified, 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 requests 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.

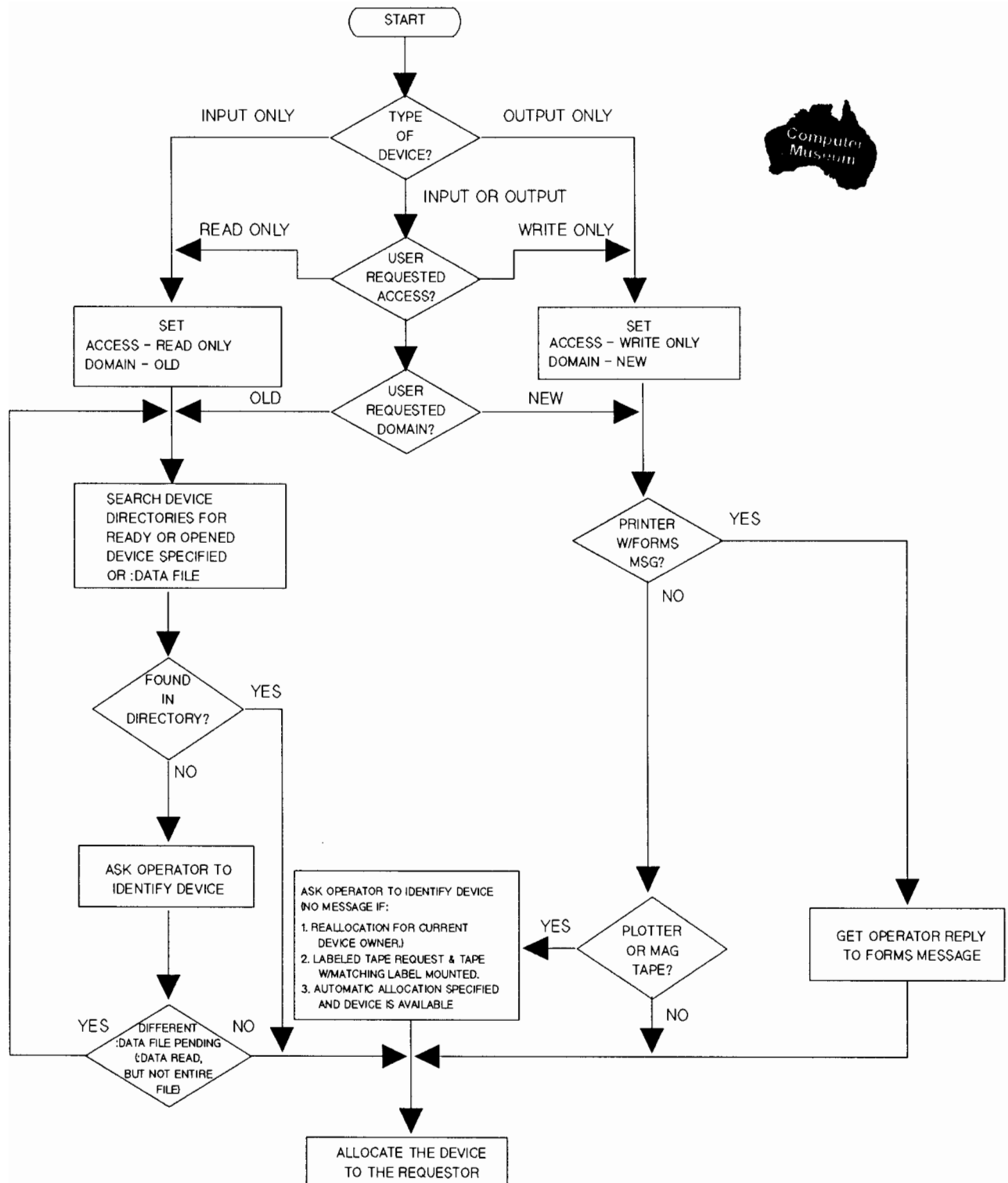


Figure 4-5. Device Allocation Flowchart

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 disk 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 user name and account name and (optionally) job name and/or file 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 disk spoolfile. A permanent **:DATA** file can be created on disk. This file can be made available to user programs by using the **:STREAM** command. This also results in a disk spoolfile. Refer to the *MPE V Commands Reference Manual* (32033-90006) for additional information of the **:STREAM** command.

If the entry in the device directory indicates that the device is opened, a user process has already opened the devicefile (device or spoolfile) successfully. 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 class name. 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 printer can result in intermixing of output data.

A condition code error is returned to the calling process if:

- The device type specified an input-only device and the requested access was write only.
- The device type specified an output-only device and the requested access was read only.

A message to the Operator will be printed if:

- The device is a card reader (spooled or unspooled) and a predefined file with a **:DATA** card) cannot be located to match the file requested.
- The device is a line printer and uses the forms message option. If the line printer is spooled, the Operator dialog takes place when the file is printed, not when it is created by the user program.
- The device is a magnetic tape device and automatic allocation, described below, is not used.
- The device is a plotter.

The Operator message is omitted if:

- The device is being reopened by logical device number by a process in the same process tree as the one which originally opened the device (except that forms message requests always result in an Operator message).
- For magnetic tapes and serial disks only:
 1. The call to **FOPEN** specifies an ANSI or IBM labeled tape or serial disk, and media with a valid matching label has already been mounted and recognized by
 2. The **FOPEN** request properly specifies a magnetic tape or serial disk device which has been configured for automatic allocation, and which is available (not assigned to another process tree). Refer to the *MPE V System Operation and Resource Management Reference Manual* (32033-90005) for a description of the requirements for automatic allocation of magnetic tapes.

Parsing and Validating File Designators

The **FPARSE** intrinsic parses and validates a file designator string to determine if it is syntactically correct. You can employ this intrinsic to check a formal designator representing a file before attempting to open the file via **FOPEN**. **FOPEN** also calls **FPARSE**, however, by calling **FPARSE** directly through your program, syntax errors are easier to identify.

MPE file designators used for the file system and two user interface commands include a remote environment ID (*envid*). This allows the user to indicate that a file is to be accessed from a remote environment established by the user with the **:DSL** or **:REMOTE HELLO** command. **FPARSE** facilitates the changes required for the file designator extension. It provides the only location within MPE where file designators are parsed and syntax is checked.

The optional *items* (input) and *vectors* (output) arrays enable you to acquire parsing information for the file designator; namely the length of each item and its position in the string. The items array must be set up before the call to **FPARSE**. This is done by entering the item numbers for which parsing information is required into the items array. They may be entered in any order you desire. The items array is terminated with a zero entry. The possible items (*itemnums*), as shown in the examples below, are file name (1), lockword (2), group name (3), account name (4), and environment ID (5). The environment ID is treated as a single item and is not parsed into environment name, domain, and organization.

The following are examples of the *items* and the *vectors* array pair. The order of entries in the *vectors* array corresponds to the order of items in the *items* array. Each double word entry in the vectors array will return the byte offset of the item in the first word, and the length in bytes of the item in the second word. However, the last entry of the *vectors* array has a different meaning from that of the other entries: The second word gives the total length of the file string, and the first word gives a system file code when applicable.

EXAMPLE 1:

In Example 1 the file string is
"FILENAME/LOCKWORD.GROUP.ACCOUNT:ANIMAL.INDDCL.HPBCG":

111111111122222222223333333333444444444455
012345678901234567890123456789012345678901
"filename/lockword.group.account:animal.inddcl.hpbcg"

items array		vectors array	
111111		1111111111222222222233	
0123456789012345		01234567890123456789012345678901	
+-----+		+-----+	
1	(0)	0	8
-----		-----	
5	(1)	32	19
-----		-----	
3	(2)	18	5
-----		-----	
4	(3)	24	7
-----		-----	
2	(4)	9	8
-----		-----	
0	(5)	0	51
+-----+		+-----+	

The items array, as illustrated above, can be listed in any order, or left unspecified if not required.

EXAMPLE 2:

In Example 2 the file string is `"*FILENAME:ANIMAL"`:

```
111111
0123456789012345
"*filename:animal "
```

items array			vectors array		
111111			111111111122222222233		
0123456789012345			01234567890123456789012345678901		
1	(0)		1	8	
2	(1)		0	0	
3	(2)		0	0	
4	(3)		0	0	
5	(4)		10	6	
0	(5)		0	15	

EXAMPLE 3:

In Example 3 the file string is "\$OLDPASS":

```
012345678
"$OLDPASS "
```

items array											vectors array										
111111											1111111111222222222233										
0123456789012345											01234567890123456789012345678901										
+-----+											+-----+										
1			(0)	0				8													
-----											-----										
2			(1)	0				0													
-----											-----										
3			(2)	0				0													
-----											-----										
4			(3)	0				0													
-----											-----										
5			(4)	0				0													
-----											-----										
0			(5)	3				8													
+-----+											+-----+										

Note that "\$" is a special exception to the rules of file names and is considered part of the file name unlike "*", which is not.

Opening a New Disk File

Figure 4-5 contains an SPL program which opens two files: a card reader file and a new disk file.

The second **FOPEN** call in Figure 4-5 (Statement 00019000):

```
OUT:=FOPEN(OUTPUT,%4,%101,128);
```

opens the new disk 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

The above bit pattern specifies the following file options:

Domain: New file. 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.

recsize

The logical record size is specified as 128 words.

All other parameters are omitted from the **FOPEN** intrinsic call.

Once the file is opened, the file number (used by other system intrinsics when referencing this file) is returned to the variable **OUT**.

The condition code for second **FOPEN** call is checked with the statement on line 00020000 of Figure 4-5. 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 on line 00036000 of Figure 4-5 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** is set to 0, where 0 specifies that the File Information Display will reflect the status of the file referenced in the last call to **FOPEN**. Refer to Appendix A for a discussion of the File Information Display.

The **QUIT** intrinsic call (statement 00023000 in Figure 4-5) 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, and old files are saved and assigned to the same domain to which they belonged previously. This may be overridden by a **:FILE** command.

```

PAGE 0001 HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
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 IN:=FOPEN(INPUT,%5,,40,DEV); <<CARD READER>>
00013000 00012 1 IF < THEN <<CHECK FOR ERROR>>
00014000 00013 1 BEGIN
00015000 00013 2 PRINT'FILE'INFO(IN); <<PRINT ERROR>>
00016000 00015 2 QUIT(1); <<ABORT>>
00017000 00017 2 END;
00019000 00017 1 OUT:=FOPEN(OUTPUT,%4,%101,128); <<NEW DISC FILE>>
00020000 00030 1 IF < THEN <<CHECK FOR ERROR>>
00021000 00031 1 BEGIN
00022000 00031 2 PRINT'FILE'INFO(OUT); <<PRINT ERROR>>
00023000 00033 2 QUIT(2); <<ABORT>>
00024000 00035 2 END;
00025000 00035 1 COPY'LOOP:
00026000 00035 1 LGTH:=FREAD(IN,BUFFER,40); <<READ A CARD>>
00027000 00043 1 IF < THEN <<CHECK FOR ERROR>>
00028000 00044 1 BEGIN
00029000 00044 2 PRINT'FILE'INFO(IN); <<PRINT ERROR>>
00030000 00046 2 QUIT(3); <<ABORT>>
00031000 00050 2 END;
00032000 00050 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00033000 00051 1 FWRITE(OUT,BUFFER,LGTH,0); <<COPY CARD TO DISC>>
00034000 00056 1 IF <> THEN <<CHECK FOR ERROR>>
00035000 00057 1 BEGIN
00036000 00057 2 PRINT'FILE'INFO(OUT); <<PRINT ERROR>>
00037000 00061 2 QUIT(4); <<ABORT>>
00038000 00063 2 END;
00039000 00063 1 GO COPY'LOOP; <<CONTINUE COPYING>>
00040000 00066 1 END'OF'FILE:
00041000 00066 2 FCLOSE(OUT,%1,0); <<MAKE PERMANENT>>
00042000 00072 1 IF < THEN <<CHECK FOR ERROR>>
00043000 00073 1 BEGIN
00044000 00073 2 PRINT'FILE'INFO(OUT); <<PRINT ERROR>>
00045000 00075 2 QUIT(5); <<ABORT>>
00046000 00077 2 END;
00047000 00077 1 END.

```

Figure 4-6. Opening a New Disk File

Opening an Old Disk File

Figure 4-6 contains an SPL program that opens three files: an old disk file, \$STDIN, and \$STDLIST.

Statement 00016000:

```
DFILE1:=FOPEN(DATA1,%5,%345,128);
```

opens the old disk file. The parameters specified are:

formaldesignator **DATAONE**, which is contained in the byte array **DATA1**.

foptions **%5**, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	BINARY
														5		OCTAL

The above bit pattern specifies the following file options:

Domain: Old permanent file. The system file directory should be searched. Bits (14:2) = 01.

ASCII/Binary: ASCII. Bit (13:1) = 1

aoptions **%345**, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	1	BINARY
								3		4			5			OCTAL

The above bit pattern specifies the following access options:

Access Type: Update access. (This file is updated later in the program with the **FUPDATE** intrinsic.)
Bits (12:4) = 0101.

Multirecord: Non-multirecord mode. Bit (11:1) = 0.

Dynamic Locking: Dynamic locking allowed. Bit (10:1) = 1.

Exclusive: Share access. Bits (8:2) = 11.

reclsize The logical record size is specified as 128 words.

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 file has a lockword, and the lockword is not included in the

formaldesignator parameter nor (for sessions only) supplied by the user on request.

Once the file is opened, the file number (used by other file system intrinsics when referencing this file) is returned to the variable `DFILE1`.


```

PAGE 0001  HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000  00000 0  $CONTROL USLINIT
00002000  00000 0  BEGIN
00003000  00000 1      BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000  00005 1      ARRAY BUFFER(0:127);
00005000  00005 1      INTEGER DFILE1,LGTH,DUMMY,IN,LIST;
00006000  00005 1      INTRINSIC FOPEN,FREAD,FUPDATE,FLOCK,FUNLOCK,FCLOSE,
00007000  00005 1          PRINT'FILE'INFO,QUIT,FWRITE,FREAD;
00008000  00005 1      PROCEDURE FILERROR(FILENO,QUITNO);
00009000  00000 1          VALUE QUITNO;
00010000  00000 1          INTEGER FILENO,QUITNO;
00011000  00000 1          BEGIN
00012000  00000 2              PRINT'FILE'INFO(FILENO);
00013000  00002 2              QUIT(QUITNO);
00014000  00004 2          END;
00015000  00000 1      <<END OF DECLARATIONS>>
00016000  00000 1          DFILE1:=FOPEN(DATA1,%5,%345,128);<<OLD DISC FILE>>
00017000  00011 1          IF < THEN FILERROR(DFILE1,1);    <<CHECK FOR ERROR>>
00018000  00015 1          IN:=FOPEN(,%244);                <<$STDIN>>
00019000  00024 1          IF < THEN FILERROR(IN,2);          <<CHECK FOR ERROR>>
00020000  00030 1          LIST:=FOPEN(,%614,%1);            <<$STDLIST>>
00021000  00040 1          IF < THEN FILERROR(LIST,3);        <<CHECK FOR ERROR>>
00022000  00044 1
00023000  00044 1      UPDATE'LOOP:
00024000  00044 1          FLOCK(DFILE1,1);                  <<LOCK FILE/SUSPEND>>
00025000  00047 1          IF < THEN FILERROR(DFILE1,4);      <<CHECK FOR ERROR>>
00026000  00053 1          LGTH:=FREAD(DFILE,BUFFER,128);    <<GET EMPLOYEE RECD>>
00027000  00061 1          IF < THEN FILERROR(DFILE1,5);      <<CHECK FOR ERROR>>
00028000  00065 1          IF > THEN GO END'OF'FILE;          <<CHECK FOR EOF>>
00029000  00070 1          FWRITE(LIST,BUFFER,-20,%320);      <<EMPLOYEE NAME>>
00030000  00075 1          IF <> THEN FILERROR(LIST,6);        <<CHECK FOR ERROR>>
00031000  00101 1          DUMMY:=FREAD(IN,BUFFER(30),5);    <<EMPLOYEE NUMBER>>
00032000  00110 1          IF < THEN FILERROR(IN,7);          <<CHECK FOR ERROR>>
00033000  00114 1          IF > THEN GO END'OF'FILE;
00034000  00115 1          FUPDATE(DFILE1,BUFFER,128);        <<EMPLOYEE RECORD>>
00035000  00121 1          IF <> THEN FILERROR(DFILE1,8);      <<CHECK FOR ERROR>>
00036000  00125 1          FUNLOCK(DFILE1);                  <<ALLOW OTHER ACCESS>>
00037000  00127 1          IF <> THEN FILERROR(DFILE1,9);      <<CHECK FOR ERROR>>
00038000  00133 1          GO UPDATE'LOOP;                    <<CONTINUE UPDATE>>
00039000  00140 1      END'OF'FILE;
00040000  00140 1          FUNLOCK(DFILE1);                  <<ALLOW OTHER ACCESS>>
00041000  00142 1          IF <> THEN FILERROR(DFILE1,10);     <<CHECK FOR ERROR>>
00042000  00146 1          FCLOSE(DFILE1,0,0);               <<DISP-NO CHANGE>>
00043000  00151 1          IF < THEN FILERROR(DFILE1,11);     <<CHECK FOR ERROR>>
00044000  00155 1      END.

```

Figure 4-7. Opening an Old Disk File

The condition code for this **FOPEN** is checked in line 00017000 of Figure 4-6. If the condition code is CCL, the error-check procedure **FILERROR** is called,



and two parameters, **DFILE1** and **1**, are passed to it for **FILENO** and **QUITNO**. (Refer to statements 00008000 through 00014000 in Figure 4-6.) **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** is set to **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**. Refer to Appendix A for a discussion of the File Information Display.

The **QUIT** intrinsic call (statement 00013000 of Figure 4-6) aborts the program's process. The value of **QUITNO** is **1** and this number is printed as a 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 **%100001** in this example.

Opening a File on a Device Other Than Disk

Figure 4-7 contains an SPL program that opens a card reader file and a disk file, reads the contents of a card deck and writes the records read from the card deck into the disk file and, finally, closes the disk file as a permanent file.

If the desired card deck has been read in by the spooler before the program which references the deck executes, or if the **:DATA** card of that deck has been read by an unspooled card reader, 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 System Operator to reply with the logical device number of the device on which the file resides.

In Figure 4-7, statement 00010000:

```
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 (13:3) = 1.
<i>aoptions</i>	Omitted. All bits are set to zero. Access defaults to READ only.
<i>recsize</i>	40 words.
<i>device</i>	CARD . The byte array DEV , containing the string "CARD", is specified for the <i>device</i> parameter.

All other parameters are omitted in the **FOPEN** 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 (line 00011000 of Figure 4-7) 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.

Line 00013000 in Figure 4-7, **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** is set to 0, where 0 specifies that the File Information Display will reflect the status of the file referenced in the last call to **FOPEN**. Refer to Appendix A for a discussion of the File Information Display.

The **QUIT** intrinsic call in line 00014000 of Figure 4-7:

```
QUIT(1);
```

aborts the process. The parameter 1 is an arbitrary user-specified number. When the program is terminated this number is displayed allowing you to determine, in the case of multiple **QUIT** intrinsic calls in a program, which specific **QUIT** call was executed.

Using **FREAD** and **FWRITE** with **\$STDIN** and **\$STDLIST**

If the standard input device (**\$STDIN**) and the standard list device (**\$STDLIST**) are opened with an **FOPEN** intrinsic call, the **FREAD** and **FWRITE** intrinsics 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 **FREAD** intrinsic can be used to transfer information from a buffer in the stack directly to the standard list device.

```

PAGE 0001  HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
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      INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT
00009000  00005 1      << END OF DECLARATIONS >>
00010000  00005 1          IN:=FOPEN(INPUT,%5,,40,DEV);      <<CARD READER>>
00011000  00012 1          IF < THEN                          <<CHECK FOR ERROR>>
00012000  00013 1              BEGIN
00013000  00013 2                  PRINT'FILE'INFO(IN);        <<PRINT ERROR>>
00014000  00015 2                  QUIT(1);                    <<ABORT>>
00015000  00017 2              END;
00016000  00017 1          OUT:=FOPEN(OUTPUT,%4,%101,128); <<NEW DISC FILE>>
00017000  00030 1          IF < THEN                          <<CHECK FOR ERROR>>
00018000  00031 1              BEGIN
00019000  00031 2                  PRINT'FILE'INFO(OUT);      <<PRINT ERROR>>
00020000  00033 2                  QUIT(2);                    <<ABORT>>
00021000  00035 2              END;
00022000  00035 1      COPY'LOOP:
00023000  00035 1          LGTH:=FREAD(IN,BUFFER,40);        <<READ A CARD>>
00024000  00043 1          IF < THEN                          <<CHECK FOR ERROR>>
00025000  00044 1              BEGIN
00026000  00044 2                  PRINT'FILE'INFO(IN);        <<PRINT ERROR>>
00027000  00046 2                  QUIT(3);                    <<ABORT>>
00028000  00050 2              END;
00029000  00050 1          IF > THEN GO END'OF'FILE;          <<CHECK FOR EOF>>
00030000  00051 1          FWRITE(OUT,BUFFER,LGTH,0);        <<COPY CARD TO DISC>>
00031000  00056 1          IF < THEN                          <<CHECK FOR ERROR>>
00032000  00057 1              BEGIN
00033000  00057 2                  PRINT'FILE'INFO(OUT);      <<PRINT ERROR>>
00034000  00061 2                  QUIT(4);                    <<ABORT>>
00035000  00063 2              END;
00036000  00063 1          GO COPY'LOOP;                      <<CONTINUE COPYING>>
00037000  00066 1      END'OF'FILE:
00038000  00066 1          FCLOSE(OUT,%1,0);                  <<MAKE PERMANENT>>
00039000  00072 1          IF < THEN                          << CHECK FOR ERROR>>
00040000  00073 1              BEGIN
00041000  00073 2                  PRINT'FILE'INFO(OUT);      <<PRINT ERROR>>
00042000  00075 2                  QUIT(5);                    <<ABORT>>
00043000  00077 2              END;
00044000  00077 1      END;

```

Figure 4-8. Opening a New Disk File

Opening \$STDIN

Figure 4-8 contains a program that opens **\$STDIN** so that **FREAD** intrinsic calls can be issued directly against the standard input device (in this case a terminal, since 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:

formaldesignator

Omitted.

Default: A temporary nameless file, that can be read but not saved, is assigned.

foptions

%244, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	BINARY
								2					4			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.

Since **\$STDIN** is specified (bits (10:3)=100), MPE knows the file exists and ignores the new file specification without reporting an error. 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 (00019000) 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 (statements 00008000 through 00014000) 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 00013000 in Figure 4-8) aborts the process.

Opening \$STDLIST

In Figure 4-8, statement 00020000:

```
LIST:=FOPEN(,%614,%1);
```

opens the standard list device so that the **FWRITE** intrinsic can be used to transfer information directly to the device.

formal designator

Omitted.

Default: A temporary nameless file, that can be written on but not saved, is assigned.

foptions

%614, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	BINARY
							6			1			4			OCTAL

The preceding 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 preceding 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 (00021000 of Figure 4-8) checks the condition code. If the condition code is CCL, signifying that the **FOPEN** request was denied, the error-check procedure **FILEERROR** is called.

The **FILEERROR** procedure (statements 00008000 through 00014000 in Figure 4-8) 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 00013000) aborts the process.

```

PAGE 0001  HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL $USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000 00005 1  ARRAY BUFFER(0:127);
00005000 00005 1  INTEGER DFILE1,LGTH,DUMMY,IN,LIST;
00006000 00005 1  INTRINSIC FOPEN,FREAD,FUPDATE,FUNLOCK,FCLOSE,
00007000 00005 1          PRINT'FILE'INFO,QUIT,FWRITE,FREAD;
00008000 00005 1  PROCEDURE FILERROR(FILENO,QUITNO);
00009000 00000 1      VALUE QUITNO;
00010000 00000 1      INTEGER FILENO,QUITNO;
00011000 00000 1      BEGIN
00012000 00000 2          PRINT'FILE'INFO(FILENO);
00013000 00002 2          QUIT(QUITNO);
00014000 00004 2      END;
00015000 00000 1  <<END OF DECLARATIONS>>
00016000 00000 1      DFILE1:=FOPEN(DATA1,%5,%345,128);<<OLD DISC FILE>>
00017000 00011 1      IF < THEN FILERROR(DFILE1,1);      <<CHECK FOR ERROR>>
00018000 00015 1      IN:=FOPEN(,%244);      <<$STDIN>>
00019000 00024 1      IF < THEN FILERROR(IN,2);      <<CHECK FOR ERROR>>
00020000 00030 1      LIST:=FOPEN(,%614,%1);      <<$STDLIST>>
00021000 00040 1      IF < THEN FILERROR(LIST,3);      <<CHECK FOR ERROR>>
00022000 00044 1  UPDATE'LOOP:
00023000 00044 1      FLOCK(DFILE1,1);      <<LOCK FILE/SUSPEND>>
00024000 00047 1      IF < THEN FILERROR(DFILE1,4);      <<CHECK FOR ERROR>>
00025000 00053 1      LGTH:=FREAD(DFILE1,BUFFER,128);      <<GET EMPLOYEE RECD>>
00026000 00061 1      IF < THEN FILERROR(DFILE1,5);      <<CHECK FOR ERROR>>
00027000 00065 1      IF > THEN GO END'OF'FILE;      <<CHECK FOR EOF>>
00028000 00070 1      FWRITE(LIST,BUFFER,-20,%320);      <<EMPLOYEE NAME>>
00029000 00075 1      IF <> THEN FILERROR(LIST,6);      <<CHECK FOR ERROR>>
00030000 00101 1      DUMMY:=FREAD(IN,BUFFER(30),5);      <<EMPLOYEE NUMBER>>
00031000 00110 1      IF < THEN FILERROR(IN,7);      <<CHECK FOR ERROR>>
00032000 00114 1      IF > THEN GO END'OF'FILE;
00034000 00115 1      FUPDATE(DFILE1,BUFFER,128);      <<EMPLOYEE RECORD>>
00035000 00121 1      IF <> THEN FILERROR(DFILE1,8);      <<CHECK FOR ERROR>>
00036000 00125 1      FUNLOCK(DFILE1);      <<ALLOW OTHER ACCESS>>
00037000 00127 1      IF <> THEN FILERROR(DFILE1,9);      <<CHECK FOR ERROR>>
00038000 00133 1      GO UPDATE'LOOP;      <<CONTINUE UPDATE>>
00039000 00140 1  END'OF'FILE:
00040000 00140 1      FUNLOCK(DFILE1);      <<ALLOW OTHER ACCESS>>
00041000 00142 1      IF <> THEN FILERROR(DFILE1,10);      <<CHECK FOR ERROR>>
00042000 00146 1      FCLOSE(DFILE1,0,0);      <<DISP-NO CHANGE>>
00043000 00151 1      IF < THEN FILERROR(DFILE1,11);      <<CHECK FOR ERROR>>
00044000 00155 1  END.

```



Figure 4-9. Opening \$STDIN and \$STDLIST

Closing Files

To terminate access to a file, you use the **FCLOSE** intrinsic. The **FCLOSE** intrinsic applies to files on all devices, and deallocates the device on which the file resides. If a file has several concurrent **FOPEN** calls issued by the same user, the device is not deallocated until the last “nested” **FCLOSE** intrinsic is executed.

The **FCLOSE** intrinsic may be used to change the disposition of a disk or magnetic tape file. For example, a file opened as a new file can be closed and saved as an old file with permanent or temporary disposition. Alternately, a disk file opened as a temporary file can be closed as temporary or saved as a permanent file.

When the **FOPEN** intrinsic opens a disk file specified as new in the *foptions* parameter (bits (14:2) = 00), neither the job temporary domain nor the system file domain is searched to ensure that a file of the same name does not exist already. However, if this file is saved with the **FCLOSE** intrinsic, 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 name as an existing file, but an error will occur 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 an old temporary file in the *foptions* parameter (bits (14:2) = 10), only the job temporary file domain (not the system file domain) is searched. Thus it is possible to have three files with the same name, a permanent file, a new file, and a temporary file. If a file opened as temporary 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 in which they belonged previously, either permanent or temporary. This may be altered, however, with a **:FILE** command. Further information on changing domains can be found in the *MPE File System Reference Manual* (30000-90236). The following examples illustrate how a new file can be closed as either a temporary file or a permanent file.

Closing a New File as a Temporary File

Figure 4-9 contains an **FCLOSE** intrinsic call that closes a new file as a temporary job file.

The **FCLOSE** intrinsic call in statement 00036000:

```
FCLOSE(DFILE2,%2,0);
```

closes the file specified by **DFILE2**. The parameters specified in the above intrinsic call are:

<i>filenum</i>	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 reopened 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. However, if this is the last FCLOSE on the device, then the tape is rewound and unloaded. Bits (13:3) = 010.

seccode 0. Unrestricted access.

A condition code of CCL is returned if the file is not closed successfully.
Statement 00037000:

```
IF < THEN FILEERROR(DFILE2,7);
```

checks the condition code. If the condition code is CCL, the error-check procedure FILEERROR (statements 00011000 through 00017000 in Figure 4-9) 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 (00016000) aborts the process.

The QUIT intrinsic causes MPE to close all files with no change. Thus new files are deleted, and old files are saved and assigned to the same domain to which they belonged previously, unless otherwise requested in a :FILE command.

```

PAGE 0001  HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLIMIT
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  INTRINSIC FOPEN,FWRITELABEL,FGETINFO,FREAD,WRITEDIR,FCLOSE
00010000 00011 1          PRINT'FILE'INFO,QUIT;
00011000 00011 1  PROCEDURE FILERROR(FILENO,QUITNO);
00012000 00000 1      VALUE QUITNO;
00013000 00000 1      INTEGER FILENO,QUITNO;
00014000 00000 1      BEGIN
00015000 00000 2          PRINT'FILE'INFO(FILENO);
00016000 00002 2          QUIT(QUITNO);
00017000 00004 2      END;
00018000 00000 1  <<END OF DECLARATIONS>>
00019000 00000 1      DFILE1:=FOPEN(DATA1,%5,%100);          <<OLD FILE-DATAONE>>
00020000 00010 1      IF < THEN FILERROR(DFILE1,1);          <<CHECK FOR ERROR>>
00021000 00014 1      DFILE2:=FOPEN(DATA2,%4,%4,128,,,1);    <<NEW FILE-DATATWO>>
00022000 00027 1      IF < THEN FILERROR(DFILE2,2);          <<CHECK FOR ERROR>>
00023000 00033 1      FWRITELABEL(DFILE2,LABL,9,0);          <<FILE ID>>
00024000 00041 1      IF <> THEN FILERROR(DFILE2,3);          <<CHECK FOR ERROR>>
00025000 00045 1      FGETINFO(DFILE1,,,,,,,,,REC);          <<LOCATE EOF>>
00026000 00053 1      IF < THEN FILERROR(DFILE1,4);          <<CHECK FOR ERROR>>
00027000 00057 1  INVERT'LOOP:
00028000 00057 1          DUMMY:=FREAD(DFILE1,BUFFER,128);    <<OLD FILE RECORD>>
00029000 00065 1          IF < THEN FILERROR(DFILE1,5);        <<CHECK FOR ERROR>>
00030000 00071 1          IF > THEN GO END'OF'FILE;            <<CHECK FOR EOF>>
00031000 00072 1          REC:=REC-1D                          <<LAST REDC NO>>
00032000 00076 1          FWRITEDIR(DFILE2,BUFFER,128,REC);    <<INVERT REC ORDER>>
00033000 00103 1          IF <> THEN FILERROR(DFILE2,6);        <<CHECK FOR ERROR>>
00034000 00107 1          GO INVERT'LOOP;                      <<CONTINUE OPERATION>>
00035000 00116 1  END'OF'FILE:
00036000 00116 1          FCLOSE(DFILE2,%2,0);                <<SAVE NEW AS TEMP>>
00037000 00122 1          IF < THEN FILERROR(DFILE2,7);        <<CHECK FOR ERROR>>
00038000 00126 1          FCLOSE(DFILE1,4,0);                  <<DELETE OLD FILE>>
00039000 00132 1          IF <> THEN FILERROR(DFILE1,8);        <<CHECK FOR ERROR>>
00040000 00136 1  END.

```

Figure 4-10. Closing a New File as a Temporary File

Closing a New File as a Permanent File

Figure 4-10 contains an **FCLOSE** intrinsic call that closes a new file as a permanent file. The **FCLOSE** intrinsic call in statement 00038000:

```
FCLOSE(OUT,%11,0);
```

closes the disk 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 disk, 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 disk, this disposition value has no effect. If the file is stored on magnetic tape, that tape is rewound and unloaded. Bits (13:3) = 001.

Disk Space Disposition: Unused disk space returned to the system. (Applicable to fixed and undefined length files only.) Bits (12:1) = 1.

seccode 0. Unrestricted access.

A condition code of CCL is returned if the file is not closed successfully. Statement 00039000 of Figure 4-10 checks the condition code and, if it is CCL, the next four statements, starting with the **BEGIN** statement, are executed.

The **PRINT'FILE'INFO** intrinsic which is called in statement 00041000 of Figure 4-10 prints a File Information Display on the standard list device, enabling you to determine the error number returned by **FCLOSE**. The **QUIT** intrinsic call (00042000, Figure 4-10) aborts the process.

```

PAGE 0001 HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
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 INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT;
00009000 00005 1 <<END OF DECLARATIONS>>
00010000 00005 1 IN:=FOPEN(INPUT,%5,,40,DEV); <<CARD READER>>
00011000 00012 1 IF < THEN <<CHECK FOR ERROR>>
00012000 00013 1 BEGIN
00013000 00013 2 PRINT'FILE'INFO(IN); <<PRINT ERROR>>
00014000 00015 2 QUIT(1); <<ABORT>>
00015000 00017 2 END;
00016000 00017 1 OUT:=FOPEN(OUTPUT,%4,%101,128);<<NEW DISC FILE>>
00017000 00030 1 IF < THEN <<CHECK FOR ERROR>>
00018000 00031 1 BEGIN
00019000 00031 2 PRINT'FILE'INFO(OUT); <<PRINT ERROR>>
00020000 00033 2 QUIT(2); <<ABORT>>
00021000 00035 2 END;
00022000 00035 1 COPY'LOOP:
00023000 00035 1 LGTH:=FREAD(IN,BUFFER,40); <<READ A CARD>>
00024000 00043 1 IF < THEN <<CHECK FOR ERROR>>
00025000 00044 1 BEGIN
00026000 00044 2 PRINT'FILE'INFO(IN); <<PRINT ERROR>>
00027000 00046 2 QUIT(3); <<ABORT>>
00028000 00050 2 END;
00029000 00050 1 IF > THEN GO END'OF'FILE; <<CHECK FOR EOF>>
00030000 00051 1 FWRITE(OUT,BUFFER,LGTH,0); <<COPY CARD TO DISC>>
00031000 00056 1 IF <> THEN <<CHECK FOR ERROR>>
00032000 00057 1 BEGIN
00033000 00057 2 PRINT'FILE'INFO(OUT); <<PRINT ERROR>>
00034000 00061 2 QUIT(4); <<ABORT>>
00035000 00063 2 END;
00036000 00063 1 GO COPY'LOOP; <<CONTINUE COPYING>>
00037000 00066 1 END'OF'FILE:
00038000 00066 1 FCLOSE(OUT,%1,0); <<MAKE PERMANENT>>
00039000 00072 1 IF < THEN <<CHECK FOR ERROR>>
00040000 00073 1 BEGIN
00041000 00073 2 PRINT'FILE'INFO(OUT); <<PRINT ERROR>>
00042000 00075 2 QUIT(5); <<ABORT>>
00043000 00077 2 END;
00044000 00077 1 END.

```

Figure 4-11. Closing a New File as a Permanent File

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 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 4-11 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 disk file, reads the card file, writes these records into the disk file, then closes the disk file.

The error check procedure (statements 00009000 through 00015000 in Figure 4-11) contains two parameters: **FILENO** (integer) and **QUITNO** (integer by value). **FILENO** is an identifier through which the file number is passed. 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.

Using FERRMSG

This intrinsic is usually called 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(FILENUM,1,0);
IF <
THEN BEGIN
  FCHECK(FILENUM,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
```

```

PAGE 0001  HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
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  INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,PRINT'FILE'INFO,QUIT;
00009000 00005 1  PROCEDURE FILEERROR(FILENO,QUITNO);
00010000 00000 1      VALUE QUITNO;
00011000 00000 1      INTEGER FILENO,QUITNO;
00012000 00000 1      BEGIN
00013000 00000 2          PRINT'FILE'INFO(FILENO);
00014000 00002 2          QUIT(QUITNO);
00015000 00004 2      END;
00016000 00000 1  <<END OF DECLARATIONS>>
00017000 00000 1      IN:=FOPEN(INPUT,%5,,40,DEV);      <<CARD READER>>
00018000 00012 1      IF < THEN FILEERROR(IN,1);      <<CHECK FOR ERROR>>
00019000 00016 1      OUT:=FOPEN(OUTPUT,%4,%101,128);  <<NEW DISC FILE>>
00020000 00027 1      IF < THEN FILEERROR(OUT,2);      <<CHECK FOR ERROR>>
00021000 00033 1  COPY'LOOP
00022000 00033 1      LGTH:=FREAD(IN,BUFFER,40);      <<READ A CARD>>
00023000 00041 1      IF < THEN FILEERROR(IN,3);      <<CHECK FOR ERROR>>
00024000 00045 1      IF > THE GO END'OF'FILE;      <<CHECK FOR EOF>>
00025000 00046 1      FWRITE(OUT,BUFFER,LGTH,0);      <<COPY CARD TO DISC>>
00026000 00053 1      IF <> THEN FILEERROR(OUT,4);      <<CHECK FOR ERROR>>
00027000 00057 1      GO COPY'LOOP;      <<CONTINUE COPYING>>
00028000 00062 1  END'OF'FILE:
00029000 00062 1      FCLOSE(OUT,%1,0);      <<MAKE PERMANENT>>
00030000 00066 1      IF < THEN FILEERROR(OUT,5);      <<CHECK FOR ERROR>>
00031000 00072 1  END.

```

Figure 4-12. Error-Check Procedure Example

Using the IOWAIT Intrinsic

Figure 4-12 shows a program that opens several terminals for input. Statement 00016000:

```
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 NOWAIT *aoptions* specified, the program must be running in Privileged Mode, and this program is switched to privileged mode with the GETPRIVMODE intrinsic call (statement 00019000, Figure 4-12).

Statement 00029000 of Figure 4-12:

```
FILE:=FOPEN(TNAM,%405,%4404,36,DEV(3));
```

opens a terminal. The parameters specified are:

formaldesignator 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. Bits (8:2) = 00.

Inhibit Buffering: Selected (**NOBUF**). Bit (7:1) = 1.
Multiaccess: No multiaccess. Bits (5:2) = 00.
NOWAIT: NOWAIT I/O selected. Bit (4:1) = 1.

recsize 36 words.

device **TERM** (terminal), specified in elements (3)-(7) of the byte array **DEV**.

Once the file is opened, the program is switched back to the non-privileged mode with the **GETUSERMODE** intrinsic call.

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.)

Statements 00029000/00030000 of Figure 4-12 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, the 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 4-12) operates the same way 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 4-12 would not have suspended if the `IODONTWAIT` intrinsic had been called, and control would have returned to the program.

```

PAGE 0001 HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
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 INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE,GETPRIVMODE,
00014000 00005 1 GETUSERODE,IOWAIT,QUIT;
00015000 00005 1 <<END OF DECLARATIONS>>
00016000 00005 1 OUT:=FOPEN(OUTPUT,4,1,,DEV);CCL(1);<<PRINTER OUTPUT&&>>
00017000 00015 1 WHILE (I:=I+1)<MAXTRM DO <<LOOP-SET UP TERMS>>
00018000 00023 1 BEGIN
00019000 00023 2 GETPRIVMODE; CCG(2); <<FOR NOWAIT FOPEN>>
00020000 00027 2 FILE:=FOPEN(TNAM,%405,%4404,36,DEV(3));<<INPUT&&>>
00021000 00042 2 CCL(3); <<CHECK FOR ERROR>>
00022000 00045 2 GETUSERMODE; CCG(4); <<FOR NOWAIT I/O>>
00023000 00051 2 OPEN(I):=FILE; <<SAVE FILE NUMBERS>>
00024000 00054 2 FWRITE(FILE,PROMPT,1,%320);CCNE(5);<<? PROMPT&&>>
00025000 00064 2 IOWAIT(FILE); CCNE(6); <<COMPLETE REQUEST>>
00026000 00075 2 FREAD(FILE,BUFR(I*36),-72);CCNE(7);<<INPUT NOWAIT>>
00027000 00111 2 END;
00028000 00116 1 WAIT;
00029000 00116 1 FILE:=IOWAIT(0,,LGTH); CCL(8); <<WAIT FOR 1ST DONE>>
00030000 00130 1 IF > THEN <<EOF ON TERM READ>>
00031000 00131 1 BEGIN
00032000 00131 2 FCLOSE(FILE,0,0); CCL(9); <<TERMINAL FILE&&>>
00033000 00137 2 IF(DONE:=DONE+1)>=MAXTRM THEN GO EXIT;<<TERMS CLSD?>>
00034000 00143 2 END
00035000 00143 1 ELSE
00036000 00145 1 BEGIN
00037000 00145 2 I:=-1; <<SET BUFFER INDEX>>
00038000 00147 2 DO I:=I+1 <<INCR BUFFER INDEX>>
00039000 00147 2 UNTIL OPEN(I)=FILE OR I=MAXTRM;<<SEARCH FOR FILE>>
00040000 00157 2 IF I=MAXTRM THEN QUIT(10); <<FILE NOT FOUND>>
00041000 00164 2 FWRITE(OUT,BUFR(I*36),-LGTH,0);<<COPY INPUT TO LP>>
00042000 00174 2 CCNE(11); <<CHECK FOR ERROR>>
00043000 00177 2 FWRITE(FILE,PROMPT,1,%320);CCNE(12);<<? PROMPT>>
00044000 00207 2 IOWAIT(FILE); CCNE(13); <<COMPLETE REQUEST>>
00045000 00220 2 FREAD(FILE,BUFR(I*36),-72);CCNE(14);<<IN-NOWAIT>>
00046000 00234 2 END;
00047000 00234 1 GO TO WAIT; <<CONTINUE>>
00048000 00235 1 EXIT;END.

```

Figure 4-13. Using the IOWAIT Intrinsic

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 modes that are established remain in effect until another **FSETMODE** call is issued or until the file is closed. The **FSETMODE** intrinsic applies to all files on all devices.

The following **FSETMODE** intrinsic call:

```
FSETMODE(FILE1,%2);
```

establishes access-mode options as outlined below. The parameters specified are:

filenum Designated by **FILE1**, which was assigned the file number by **FOPEN** when the file was opened. (It is a terminal in this example.)

modeflags **%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.

Terminal Control by User: MPE will issue carriage return/linefeed. Bit (13:1) = 0.

Section Divider

5. Other Applications of MPE Intrinsic



Other Applications of MPE Ininsics

MPE intrinsics allow you to perform the following utility functions:

- Manage library procedures with **LOADPROC** and **UNLOADPROC**.
- Convert numbers from ASCII to binary code with **BINARY** and **DBINARY**.
- Convert numbers from binary to ASCII code with **ASCII** and **DASCII**.
- Convert a string of characters between EBCDIC and ASCII, or between EBCDIK and JIS (KANJI) with **CTranslate**.
- Read input from job/session list devices with **READ** and **READX**.
- Write output to the job/session list device with **PRINT**.
- Write output to the System Console with **PRINTOP**, or write output to the System Console and solicit a reply with **PRINTOPREPLY**.
- Obtain system timer information with **TIMER**.
- Obtain the calendar date with **CALENDAR**.
- Obtain the time of day in terms of hour, minute, second, and tenth of second with **CLOCK**.
- Format the calendar date with **FMTCALENDAR**.
- Format the time of day with **FMTCLOCK**.
- Format the calendar date and time of day with **FMTDATE**.
- Obtain process run time (CPU time) with **PROCTIME**.
- Obtain information pertaining to your access mode and attributes with **WHO**.
- Obtain information about other jobs/sessions in the system with **JOBINFO**.
- Search an array for a specified name with **SEARCH**.
- Format the parameters of a non-MPE command with **MYCOMMAND**.
- Execute MPE commands programmatically with **COMMAND**.
- Enable or disable hardware arithmetic traps with **ARITRAP**.
- Enable or disable software arithmetic traps with **XARITRAP**.
- Enable or disable the software library trap with **XLIBTRAP**.
- Enable or disable the software system trap with **XSYSTRAP**.
- Enable/disable the CONTROL-Y trap with **XCONTRAP** and reset a terminal to accept a CONTROL-Y signal with **RESETCONTROL**.
- Change the size of the current DL to DB area with **DLSIZE**.
- Change the size of the current Z to DB area with **ZSIZE**.

- Suspend the calling process with **PAUSE**.
- Initiate a session break programmatically with **CAUSEBREAK**.
- Programmatically terminate a process (after successful execution) with **TERMINATE**.
- Programmatically abort any process within a user process structure with **QUIT**.
- Abort the entire process structure (program) with **QUITPROG**.
- Manage Interprocess Communication through the Job Control Words with **SETJCW**, **GETJCW**, **PUTJCW**, and **FINDJCW**.
- Access a message catalog in the MPE message facility, and insert parameters in a message, with the **GENMESSAGE** intrinsic.
- Control the functioning of an HP 2680/2688 page printer with **FDEVICECONTROL**.

Dynamic Loading and Unloading of Library Procedures

Normally, segments containing library procedures referenced by a program are linked to that program when the program is loaded. However, you can also dynamically link and unlink 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 entries since these segments are sharable. The procedures are loaded from segmented libraries, not from relocatable libraries (which are used only at program preparation time). Preparation and maintenance of segmented libraries and relocatable libraries is explained in the *MPE Segmenter Reference Manual* (30000-90011).

You do not need to dynamically load procedures 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**, enter the following intrinsic call:

```
PNUM:=LOADPROC(PNAME,0,LAB);
```

The parameters specified in the preceding intrinsic call are:

<i>procname</i>	Contained in the byte array PNAME . The contents of PNAME is the string "PROC1 ". Note that the string is terminated with a blank space.
<i>lib</i>	0, signifying that only the system library should be searched. If 2 were specified, library searching would proceed in this order: Logon Group Library

Logon Account Library
System Library

Specifying 1 for the *lib* parameter would cause the search to be conducted in this order:

Account Public Library
System Library

plabel **LAB**, a word to which the procedure's label ("*plabel*") is returned.

When the **LOADPROC** intrinsic executes, the procedure identity number will be returned as an integer to **PNUM**.

Dynamic Unloading

The **UNLOADPROC** intrinsic is used to unload a procedure and its referenced external procedures.

For example, to unload the procedure that was dynamically loaded in the previous example, enter 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-defined commands (UDCs) 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 subsequent byte areas.
- An integer specifying the length of the "name" (in bytes) in the entry.
- A byte string forming the name in the entry; this name and name length are checked against the search string for a match. In a command dictionary, for example, this would be a 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.

Each entry has an implied entry number in the dictionary. The entry number of the first entry in such a dictionary is one (1). If the entry is not found, zero (0) is returned by **SEARCH**.

In the examples below, the five rows all represent entries for the **SEARCH** intrinsic. The elements of each row (except the last) are the length of the entry, the length of the "name", the byte string, and the user-supplied definition. In the last row, the zero indicates the end of the array (but is not actually part of the array contents).

EXAMPLE 1:

```
BYTE ARRAY COMMANDTABLE (0:25) :=  
5,2,"IN",1,  
6,3,"OUT",1,  
0
```

```

7,4,"SKIP",2,
7,4,"EXIT",0,
0;

```

The main program might use the definitions as a cross-reference, to indicate the type of parameter which the user might enter after a prompt: 0 for no parameter, 1 for a filename, and 2 for a number.

EXAMPLE 2:

```

BYTE ARRAY SHORTCOMMANDS (0:29):=
5,1,"I","IN",
6,1,"O","OUT",
7,1,"S","SKIP",
7,1,"E","EXIT",
0;

```

The main program might use this array as a cross-reference table, to allow abbreviated commands to be entered by the user.

EXAMPLE 3:

```

BYTE ARRAY RESPONSETABLE (0:9):=
5,3,"YES",
4,2,"NO",
0;

```

This look-up table would allow the user to enter YES or NO as responses to prompts. No definitions were necessary in this example.

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 can also 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 two 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**.

Formatting Command Parameters

You can programmatically extract and format for execution the parameters of a command defined by you (that is, 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 5-1 and Figure 5-2 contains a program that determines whether the user is running the program in a session, and if so, performs the following:

- Prompts the user to enter a command name from the terminal.
- Reads the command name entered by the user.
- 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.
- Converts the parameter entered with the command to binary, then uses this operand to perform the calculation specified by the command.
- Converts the result to ASCII, then displays the result on the terminal.

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1982, 4:35 PM
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",
00011000 00010 1          5,3,"DIV",      5,3,"SET",      0;
00012000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00013000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00014000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00015000 00016 1          LGTH,      INDX,      PARMCNT,      TYPE;
00016000 00016 1
00017000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00018000 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 COMD>>
00026000 00023 1          IF <> THEN QUIT(1);                      <<CHECK FOR ERR>>
00027000 00026 1          IF COMMAND="END" THEN GO EXIT;          <<DONE - EXIT>>
00028000 00040 1          COMMAND(LGTH):=%15;                      <<CARRIAGE RETN>>
00029000 00043 1
00030000 00043 1          TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT,      <<TAKE APART CMD>>
00031000 00050 1          PARMINFO,TABLE);
00032000 00056 1          IF < THEN GO ERROR;                      <<NO CMD MATCH>>
00033000 00057 1          IF PARMCNT<>1 THEN GO ERROR;            <<NO PARAMETERS>>
00034000 00062 1          INDX:=PARMINFO-@COMMAND;                <<SUBSCR OF PARM>>
00035000 00065 1          OPERAND:=BINARY(COMMAND(INDX),          <<CONVERT PARM>>
00036000 00070 1          PARMINFO(1).(0:8));
00037000 00075 1          IF <> THEN GO ERROR;                      <<CHECK FOR ERR>>
00038100 00076 1
00039000 00076 1          CASE (TYPE-1) OF                          <<SELECT OPERATN>>
00040000 00100 1              BEGIN
00041000 00106 2                  ACCUM:=ACCUM+OPERAND;          <<ADD COMD>>
00042000 00116 2                  ACCUM:=ACCUM-OPERAND;          <<SUB COMD>>
00043000 00122 2                  ACCUM:=ACCUM*OPERAND;          <<MUL COMD>>
00044000 00126 2                  ACCUM:=ACCUM/OPERAND;          <<DIV COMMAND>>
00045000 00133 2                  ACCUM:=OPERAND;                <<SET COMMAND>>
00046000 00136 2              END;

```

Figure 5-1. Using the MYCOMMAND Intrinsic (Program UTILY) (1 of 2)

```

00047000 00143 1  RESULT:
00048000 00143 1      MOVE ANSWER(8):="    ";          <<RESET OLD ANSWER>>
00049000 00155 1      ASCII(ACCUM,10,ANSWER(8));        <<CONVERT ACCUM>>
00050000 00163 1      PRINT(OUTPUT,7,0);                <<OUTPUT NEW ANSWER>>
00051000 00170 1      GO LOOP;                          <<CONTINUE CALC>>
00052000 00171 1  ERROR:
00053000 00171 1      PRINT(ERRMSG,7,0);                <<ERROR MESSAGE>>
00054000 00175 1      IF NOT INTERACTIVE THEN QUIT(2);  <<NO LIVE USER-QUIT>>
00055000 00201 1      GO LOOP;                          <<CONTINUE CALC>>
00056000 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 5-2. Using the MYCOMMAND Intrinsic (Program UTILY) (2 of 2)

Statement 00025000:

```
LGTH:=READ(INPUT,-72);
```

reads the command entered by the user. (The arrays **INPUT** and **COMMAND** have been set as equivalents in statement 00006000 in Figure 5-1.)

Statements 00026000 and 00027000:

```
IF <> THEN QUIT(1);
IF COMMAND="END" THEN GO EXIT;
```

perform the following:

- Check for a condition code error and execute the **QUIT** intrinsic (causing the process to abort if a condition code error is returned).
- Cause the program control to transfer to the statement **EXIT** if "END" is entered by the user.

Statement 00028000:

```
COMMAND(LGTH):=%15;
```

adds a carriage return character as the last character of the *comimage* parameter for the command entered. 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. This is because in SPL, 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 five characters long. Therefore, the carriage return character is added to position 5 of the array (which is actually the 6th position).

Statement 00030000:

```
TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT,PARMINFO,TABLE);
```

calls the **MYCOMMAND** intrinsic to parse the command entered by the user. The parameters specified are:

<i>comimage</i>	Specified by the array COMMAND . It 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: <pre>COMMAND(LGTH):=%15;</pre>
<i>delimiters</i>	Omitted. The default delimiter array " COMMA, EQUAL, SEMICOLON, CARRIAGE RETURN " is used.
<i>maxparms</i>	1, specifying that one parameter is expected in <i>comimage</i> .
<i>numparms</i>	Specified by PARMCNT . It contains the actual number of parameters entered with the command.
<i>parms</i>	Specified by PARMINFO . An integer array to which the byte address of the parameter, entered as part of <i>comimage</i> , is returned.

Note



Although this parameter is listed as a double array in the specifications for the **MYCOMMAND** intrinsic in Chapter 2, 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 setting the two as equivalents.

<i>dict</i>	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. When a match is found, the number of the entry in the table containing the matching name is returned to **TYPE**. The *dict* parameter may specify a specially formatted array, or table. Each entry in the table will contain:

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

Statement 00032000:

```
IF < THEN GO ERROR;
```

checks the condition code, and if it is CCL, transfers program control to statement label **ERROR**.

Statement 00033000:

```
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**.

Statements 00034000 and 00035000:

```
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 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 offset of the parameter.

The following statement 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**:

```
OPERAND:=BINARY(COMMAND(INDX),PARMINFO(1).(0:8));
```

The following statement 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 statement following **BEGIN**) but the value assigned to **TYPE** by **MYCOMMAND** contains the range 1 to 5:

```
CASE(TYPE-1)OF
```

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 directory 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	:DSCONTROL	:LOG
:ABORTJOB	:DSLNE	:MPLINE
:ACCEPT	:DSTAT	:MRJECONTROL
:ALLOW	:FILE	:NEWACCT
:ALTACCT	:FOREIGN	:NEWGROUP
:ALTGROUP	:GETLOG	:NEWUSER
:ALTJOB	:GETRIN	:NEWVSET
:ALTLOG	:GIVE	:OUTFENCE
:ALTSEC	:HEADOFF	:PTAPE
:ALTPOOLFILE	:HEADON	:PURGE
:ALTUSER	:HELP	:PURGEACCT
:ALTVSET	:IMLCONTROL	:PURGEGROUP
:ASSOCIATE	:JOBFENCE	:PURGEUSER
:BREAKJOB	:JOBPRI	:PURGEVSET
:BUILD	:JOBSECURITY	:RECALL
:CACHECONTROL	:LDISMOUNT	:REFUSE
:CHANGELOG	:LIMIT	:RELEASE
:COMMENT	:LISTACCT	:RELLOG
:CONSOLE	:LISTEQ	:REMOTE
:DEALLOCATE	:LISTF	:REMOTE HELLO
:DELETESPOOLFILE	:LISTFTEMP	:RENAME
:DISALLOW	:LISTGROUP	:REPLY
:DISASSOCIATE	:LISTLOG	:REPORT
:DICSRPS	:LISTUSER	:RESET
:DOWN	:LISTVS	:RESETACCT
:DOWNLOAD	:LMOUNT	:RESETDUMP
:RESUMEJOB	:SHOWIN	:STOPSPPOOL
:RESUMELOG	:SHOWJCW	:STREAM
:RESUMESPOOL	:SHOWJOB	:SUSPENDSPOOL
:SAVE	:SHOWLOG	:SWITCHLOG
:SECURE	:SHOWME	:TAKE
:SETDUMP	:SHOWOUT	:TELL
:SETJCW	:SHOWQ	:TELLOP
:SETMSG	:SHOWTIME	:TUNE
:SHOWALLOW	:SPEED	:UP
:SHOWCACHE	:STARTCACHE	:VMOUNT
:SHOWCOM	:STARTSPOOL	:WARN
:SHOWDEV	:STOPCACHE	:WELCOME

Refer to the *MPE V Commands Reference Manual* (32033-90006) for a discussion of these 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 but including a terminating carriage return are contained in the byte array **COMD**. Any error code is returned to **ECODE**. The **:SHOWTIME** command has no parameters, therefore, no information is returned to **EPARM**.

When the intrinsic executes, the date and time are printed on the job/session list device.

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 5-3 and Figure 5-4 contains a program which must determine if the user is running the program in an interactive session. The statement:

```
WHO(INTERACTIVE);
```

calls the **WHO** intrinsic to make this determination. If the logical identifier **INTERACTIVE** is **TRUE** (bit (15:1) = 1) after the **WHO** intrinsic executes, and the job/session input file and job/session list file form an interactive pair, then the user is running the program interactively.

Statement 00024000:

```
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.

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1982, 4:35 PM
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",
00011000 00010 1           5,3,"DIV",      5,3,"SET",      0;
00012000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00013000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00014000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00015000 00016 1           LGTH,      INDX,      PARMCNT,      TYPE;
00016000 00016 1
00017000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00018000 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 COMD>>
00026000 00023 1          IF <> THEN QUIT(1);                        <<CHECK FOR ERR>>
00027000 00026 1          IF COMMAND ="END" THEN GO EXIT;           <<DONE - EXIT>>
00028000 00040 1          COMMAND(LGTH):=%15;                        <<CARRIAGE RETN>>
00029000 00043 1
00030000 00043 1          TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT,      <<TAKE APART CMD>>
00031000 00050 1          PARMINFO,TABLE);
00032000 00056 1          IF < THEN GO ERROR;                        <<NO CMD MATCH>>
00033000 00057 1          IF PARMCNT<>1 THEN GO ERROR;               <<NO PARAMETERS>>
00034000 00062 1          INDX:=PARMINFO-@COMMAND;                  <<SUBSCR OF PARM>>
00035000 00065 1          OPERAND:=BINARY(COMMAND(INDX),           <<CONVERT PARM>>
00036000 00070 1          PARMINFO(1).(0:8));
00037000 00075 1          IF <> THEN GO ERROR;                        <<CHECK FOR ERR>>
00038000 00076 1
00039000 00076 1          CASE (TYPE-1) OF                            <<SELECT OPERATN>>
00040000 00100 1              BEGIN
00041000 00106 2                  ACCUM:=ACCUM+OPERAND;             <<ADD COMD>>
00042000 00116 2                  ACCUM:=ACCUM-OPERAND;             <<SUB COMD>>
00043000 00122 2                  ACCUM:=ACCUM*OPERAND;             <<MUL COMD>>
00044000 00126 2                  ACCUM:=ACCUM/OPERAND;             <<DIV COMMAND>>
00045000 00133 2                  ACCUM:=OPERAND;                   <<SET COMMAND>>
00046000 00136 2              END;

```

Figure 5-3. Using the WHO Intrinsic (1 of 2)

```

00047000 00143 1  RESULT:
00048000 00143 1      MOVE ANSWER(8):="    ";          <<RESET OLD ANSWER>>
00049000 00155 1      ASCII(ACCUM,10,ANSWER(8));      <<CONVERT ACCUM>>
00050000 00163 1      PRINT(OUTPUT,7,0);              <<OUTPUT NEW ANSWER>>
00051000 00170 1      GO LOOP;                      <<CONTINUE CALC>>
00052000 00171 1  ERROR:
00053000 00171 1      PRINT(ERRMSG,7,0);              <<ERROR MESSAGE>>
00054000 00175 1      IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVE USER-QUIT>>
00055000 00201 1      GO LOOP;                      <<CONTINUE CALC>>
00056000 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 5-4. Using the WHO Intrinsic (2 of 2)

Identifying a Job or Session with JOBINFO

JOBINFO provides the user with three options for identifying a job or session which is on the system. The three options are the following:

- Taking the caller's job/session as default.
- Specifying a specific job/session number.
- Specifying a logon ID.

To retrieve information about the job/session executing JOBINFO, the user must specify the appropriate *jsind* (1 for session, 2 for job) and a *jsnnn* of " (double-word zero). Further, the *itemnum* of the first optional triple must not be a one (1) as this invokes option number three (described below). For example, to retrieve the logon ID for the caller, from a session, a JOBINFO call might look like this:

```
JOBINFO( 1, jsnnn, STATUS,,,1,jsname,ERROR1);
```

Where:

- *Jsnnn* must be ".
- *Jsind* is 1 since JOBINFO is executed from a session, and default to the caller's session for the logon ID.
- *Jsname* must be a logical array of 13 words.
- The session number will be returned through *jsnnn*.
- The second triple contains the *itemnum* of 1. When using the default job or session, the first triple can not contain an *itemnum* equaling 1.

It is not possible to attempt a default call with *jsind* equaling 2 if JOBINFO is executed from a session. Likewise, it is not possible to call JOBINFO, attempting a default call, with *jsind* equaling 1 when JOBINFO is executed from a job.

The user may identify a job or session by specifying the appropriate job/session number through *jsnnn*, along with the appropriate *jsind*. By supplying a non-zero *jsnnn*, the other two job/session identification options

are over-ridden. There is no restriction on the use of any of the optional triples, or *itemnums*. An example of specifying a specific job/session number is as follows:

```
JOBINFO( 2, jsnnn, STATUS, 1, jsname, ERROR1 );
```

Where:

- *Jsind* equals 2 and specifies that *jsnnn* is a job number.
- *Jsnnn* is a job number.
- *Itemnum* equals 1 denoting that the logon ID of job number *jsnnn* is to be retrieved.

The user may identify a job or session by specifying the appropriate job/session by supplying the appropriate logon ID through the first optional triple. The user must supply the appropriate *jsind* and *jsnnn* must be '. The logon ID is specified through the first triple with *itemnum* equal to 1, and *item* being a logical array containing the logon ID (a character string). The logon ID must be terminated by a binary zero (0). The maximum length of the logon ID is twenty-six (26) characters, plus one (1) for the binary zero terminator. An example of this is as follows:

```
JOBINFO( 2, jsnnn, STATUS, 1, LOGON'ID, ERROR1 );
```

Where:

- *Jsind* equals 2, denoting the LOGON'ID supplied is that of a job.
- *Jsnnn* equals ' '.
- The job number of the job denoted by LOGON'ID will be returned through *jsnnn*.
- The first triple is specified with an *itemnum* equal to 1, a logical array LOGON'ID containing the logon id of a job (terminated by a binary 0), and an integer error return for the item.

An example of initializing LOGON'ID might be as follows:

```
MOVE LOGON'ID(0) := "TESTJOB,LARRY.OSE",0;
```

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. **BINARY** will not convert numbers which are greater than 32,767 in value.

The **ASCII** intrinsic call is illustrated in Figure 5-6, statement 00049000:

```
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 ninth element (0 is the first element) of the byte array **ANSWER**. For convenience, *ACCUM* is declared as an integer instead of a logical value (as required by the intrinsic). SPL is not strict in checking types. In other languages, you would need to use equivalence statements in the program. 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 statement 00015000 in Figure 5-5.

```
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 starting with the 21st 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.

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1982, 4:35 PM
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",
00011000 00010 1          5,3,"DIV",      5,3,"SET",      0;
00012000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00013000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00014000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00015000 00016 1          LGTH,      INDX,      PARMCNT,      TYPE;
00016000 00016 1
00017000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00018000 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 COMD>>
00026000 00023 1          IF <> THEN QUIT(1);                                <<CHECK FOR ERR>>
00027000 00026 1          IF COMMAND ="END" THEN GO EXIT;                    <<DONE - EXIT>>
00028000 00040 1          COMMAND(LGTH):=%15;                                <<CARRIAGE RETN>>
00029000 00043 1
00030000 00043 1          TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART CMD>>
00031000 00050 1          PARMINFO,TABLE);
00032000 00056 1          IF < THEN GO ERROR;                                <<NO CMD MATCH>>
00033000 00057 1          IF PARMCNT<>1 THEN GO ERROR;                    <<NO PARAMETERS>>
00034000 00062 1          INDX:=PARMINFO-@COMMAND;                        <<SUBSCR OF PARM>>
00035000 00065 1          OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARM>>
00036000 00070 1          PARMINFO(1).(0:8));
00037000 00075 1          IF <> THEN GO ERROR;                                <<CHECK FOR ERR>>
00038000 00076 1
00039000 00076 1          CASE (TYPE-1) OF                                <<SELECT OPERATN>>
00040000 00100 1              BEGIN
00041000 0106 2                  ACCUM:=ACCUM+OPERAND;                    <<ADD COMD>>
00042000 00116 2                  ACCUM:=ACCUM-OPERAND;                    <<SUB COMD>>
00043000 00122 2                  ACCUM:=ACCUM*OPERAND;                    <<MUL COMD>>
00044000 00126 2                  ACCUM:=ACCUM/OPERAND;                    <<DIV COMMAND>>
00045000 00133 2                  ACCUM:=OPERAND;                        <<SET COMMAND>>
00046000 00136 2              END;

```

Figure 5-5. Using the ASCII Intrinsic (1 of 2)


```

00047000 00143 1  RESULT:
00048000 00143 1      MOVE ANSWER(8):="  ";          <<RESET OLD ANSWER>>
00049000 00155 1      ASCII(ACCUM,10,ANSWER(8));      <<CONVERTACCUM>>
00050000 00163 1      PRINT(OUTPUT,7,0);              <<OUTPUT NEW ANSWER>>
00051000 00170 1      GO LOOP;                      <<CONTINUE CALC>>
00052000 00171 1  ERROR:
00053000 00171 1      PRINT(ERRMSG,7,0);              <<ERROR MESSAGE>>
00054000 00175 1      IF NOT INTERACTIVE THEN QUIT(2); <<NO LIVEUSER-QUIT>>
00055000 00201 1      GO LOOP;                      <<CONTINUE CALC>>
00056000 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 5-6. Using the ASCII Intrinsic (2 of 2)

```

PAGE 0001    HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1982, 4:35
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            PRINT(MSG,16,0);                <<OUTPUT COUNTER>>
00017000 00013 2            RESETCONTROL;                    <<REARM CNTL Y TRAP>>
00018000 00014 2            TOS:=%31400+SDEC;                <<BUILD EXIT INSTRN>>
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 CNTL 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<30000000D 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 5-7. Using the DASCII Intrinsic

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 5-8, statement 00035000:

```
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 eight bits of the first word of the array **PARMINFO**. The resulting binary value is stored in the word **OPERAND**.

To convert a number from an ASCII string to a double-word (32-bit) binary value, the **DBINARY** intrinsic is used.

A **DBINARY** intrinsic call could be of the form:

```
DVAL:=DBINARY(STRING,LENGTH);
```

where **STRING** contains the octal or decimal number to be converted and **LENGTH** is an integer representing the length of the string containing the ASCII-coded value. The converted double-word value is returned to **DVAL**.

PAGE 0001 HEWLETT-PACKARD 32100A.08.1 SPL[4W] TUE, OCT 28, 1982, 4:35 PM

```

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",
00011000 00010 1            5,3,"DIV",        5,3,"SET",        0;
00012000 00016 1    INTEGER ARRAY PARMINFO(0:1);
00013000 00016 1    LOGICAL INTERACTIVE:=FALSE;
00014000 00016 1    INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ";
00015000 00016 1            LGTH,        INDX,        PARMCNT,        TYPE;
00016000 00016 1
00017000 00016 1    INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00018000 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 COMD>>
00026000 00023 1            IF <> THEN QUIT(1);                    <<CHECK FOR ERR>>
00027000 00026 1            IF COMMAND ="END" THEN GO EXIT;            <<DONE - EXIT>>
00028000 00040 1            COMMAND(LGTH):=%15;                    <<CARRIAGE RETN>>
00029000 00043 1
00030000 00043 1            TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART CMD>>
00031000 00050 1                            PARMINFO,TABLE);
00032000 00056 1            IF < THEN GO ERROR;                    <<NO CMD MATCH>>
00033000 00057 1            IF PARMCNT<>1 THEN GO ERROR;            <<NO PARAMETERS>>
00034000 00062 1            INDX:=PARMINFO-@COMMAND;            <<SUBSCR OF PARM>>
00035000 00065 1            OPERAND:=BINARY(COMMAND(INDX),            <<CONVERT PARM>>
00036000 00070 1                            PARMINFO(1).(0:8));
00037000 00075 1            IF <> THEN GO ERROR;                    <<CHECK FOR ERR>>
00038000 00076 1
00039000 00076 1            CASE (TYPE-1) OF                    <<SELECT OPERATN>>
00040000 00100 1                BEGIN
00041000 00106 2                    ACCUM:=ACCUM+OPERAND;            <<ADD COMD>>
00042000 00116 2                    ACCUM:=ACCUM-OPERAND;            <<SUB COMD>>
00043000 00122 2                    ACCUM:=ACCUM*OPERAND;            <<MUL COMD>>
00044000 00126 2                    ACCUM:=ACCUM/OPERAND;            <<DIV COMMAND>>
00045000 00133 2                    ACCUM:=OPERAND;                <<SET COMMAND>>
00046000 00136 2                END;

```



Figure 5-8. Using the BINARY Intrinsic (1 of 2)

```

00047000 00143 1  RESULT:
00048000 00143 1      MOVE ANSWER(8):="    ";          <<RESET OLD ANSWER>>
00049000 00155 1      ASCII(ACCUM,10,ANSWER(8));        <<CONVERT ACCUM>>
00050000 00163 1      PRINT(OUTPUT,7,0);                <<OUTPUT NEW ANSWER>>
00051000 00170 1      GO LOOP;                          <<CONTINUE CALC>>
00052000 00171 1  ERROR:
00053000 00171 1      PRINT(ERRMSG,7,0);                <<ERROR MESSAGE>>
00054000 00175 1      IF NOT INTERACTIVE THEN QUIT(2);   <<NO LIVE USER-QUIT>>
00055000 00201 1      GO LOOP;                          <<CONTINUE CALC>>
00056000 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 5-9. Using the BINARY Intrinsic (2 of 2)

Translating Characters with the CTRANSLATE Intrinsic

The CTRANSLATE intrinsic is used for character code translating, whether between the standard computer character codes, or 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 <i>table</i> parameter.
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.

As an example of converting from EBCDIC to ASCII, suppose the byte array **ESTRING** contains the EBCDIC characters "JOB 2". To convert this string to its ASCII equivalent and store it in the byte array **ASTRING** use the following intrinsic call:

```
CTRANSLATE(1,ESTRING,ASTRING,5);
```

The parameters specified in the previous intrinsic call are:

<i>code</i>	1, which specifies the EBCDIC-to-ASCII table. An 0 for this parameter specifies a user-defined translation table, and a 2 specifies the ASCII-to- EBCDIC table.
<i>instring</i>	ESTRING , a byte array containing the string to be converted.
<i>outstring</i>	ASTRING , 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, MPE provides other intrinsics that allow you to read information from the job/session input device (**READ** and **READX**) or write information to the job/session list device (**PRINT**). You can transmit a message to the System Console with the **PRINTOP** intrinsic or transmit a message to the System Console and solicit a reply with the **PRINTOPREPLY** intrinsic.

Note



The **READ**, **READX**, and **PRINT** intrinsics are limited in their usefulness in that **:FILE** commands are not allowed. Also, the *filenum* parameter, obtained from the **FOPEN** intrinsic, is not available for use with these intrinsics. Therefore, if an error occurs, it is not possible to determine what it is because the **FCHECK** intrinsic requires the *filenum* parameter. You may find it more convenient (and a better programming practice) to use the **FOPEN** intrinsic to open the files **\$STDIN** and **\$STDLIST**, then issue **FREAD** and **FWRITE** 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. 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-file to **\$STDIN**, and only the commands **:EOD**, and **:EOF** indicate the end of file for **\$STDINX**. The **READ** intrinsic call is illustrated in Figure 5-10 and Figure 5-11.

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]
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",
00011000 00010 1          5,3,"DIV",      5,3,"SET",      0;
00012000 00016 1  INTEGER ARRAY PARMINFO(0:1);
00013000 00016 1  LOGICAL INTERACTIVE:=FALSE;
00014000 00016 1  INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00015000 00016 1          LGTH,      INDX,      PARMCNT,      TYPE;
00016000 00016 1
00017000 00016 1  INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00018000 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 COMD>>
00026000 00023 1          IF <> THEN QUIT(1);                                <<CHECK FOR ERR>>
00027000 00026 1          IF COMMAND ="END" THEN GO EXIT;                    <<DONE - EXIT>>
00028000 00040 1          COMMAND(LGTH):=%15;                                <<CARRIAGE RETN>>
00029000 00043 1
00030000 00043 1          TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART CMD>>
00031000 00050 1          PARMINFO,TABLE);
00032000 00056 1          IF < THEN GO ERROR;                                <<NO CMD MATCH>>
00033000 00057 1          IF PARMCNT<>1 THEN GO ERROR;                        <<NO PARAMETERS>>
00034000 00062 1          INDX:=PARMINFO-@COMMAND;                          <<SUBSCR OF PARM>>
00035000 00065 1          OPERAND:=BINARY(COMMAND(INDX),                    <<CONVERT PARM>>
00036000 00070 1          PARMINFO(1).(0:8));
00037000 00075 1          IF <> THEN GO ERROR;                                <<CHECK FOR ERR>>
00038000 00076 1
00039000 00076 1          CASE (TYPE-1) OF                                <<SELECT OPERATN>>
00040000 00100 1              BEGIN
00041000 00106 2              ACCUM:=ACCUM+OPERAND;                        <<ADD COMD>>
00042000 00116 2              ACCUM:=ACCUM-OPERAND;                        <<SUB COMD>>
00043000 00122 2              ACCUM:=ACCUM*OPERAND;                        <<MUL COMD>>
00044000 00126 2              ACCUM:=ACCUM/OPERAND;                        <<DIV COMMAND>>
00045000 00133 2              ACCUM:=OPERAND;                              <<SET COMMAND>>
00046000 00136 2              END;

```

Figure 5-10. Using the PRINT and READ Intrinsics (1 of 2)

```

00047000 00143 1  RESULT:
00048000 00143 1      MOVE ANSWER(8):="    ";          <<RESET OLD ANSWER>>
00049000 00155 1      ASCII(ACCUM,10,ANSWER(8));        <<CONVERT ACCUM>>
00050000 00163 1      PRINT(OUTPUT,7,0);                <<OUTPUT NEW ANSWER>>
00051000 00170 1      GO LOOP;                          <<CONTINUE CALC>>
00052000 00171 1  ERROR:
00053000 00171 1      PRINT(ERRMSG,7,0);                <<ERROR MESSAGE>>
00054000 00175 1      IF NOT INTERACTIVE THEN QUIT(2);   <<NO LIVE USER-QUIT>>
00055000 00201 1      GO LOOP;                          <<CONTINUE CALC>>
00056000 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 5-11. Using the PRINT and READ Intrinsic (2 of 2)

Statement 00025000:

```
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 number of bytes read is returned and stored in the word LGTH when the intrinsic executes.

Statement 00026000:

```
IF <> THEN QUIT(1);
```

checks for a CCG or CCL condition code and, if either is found 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 for jobs is a line printer and for sessions a terminal. You can write a string of ASCII characters from an array in your program to this list device with the PRINT intrinsic.

In Figure 5-10, statement 00021000:

```
PRINT(HEADING,9,0);
```

transmits the string "INTEGER CALCULATOR" from the array HEADING (refer to statement 00003000). 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 System Console

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 '1' signifies that the message is 10 words long (a negative value would specify bytes). If zero is specified for the *length* parameter, only the standard message prefix is written on the System Console; the string contained in **MESSAGE** would not be transmitted.

Writing Output to the System Console and Requesting a Reply

The PRINTOPREPLY intrinsic can be used to transmit an ASCII string from an array in your program to the System Console and to request that a reply be returned. For example, a program could ask the System Operator if the line printer contains a certain type of form. If the response is affirmative, the program could then write information on these forms.

A PRINTOPREPLY intrinsic call could be as follows:

```
REPLGTH:=PRINTOPREPLY(MESSAGE,18,0,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>	'1' words. A negative value would specify bytes. If zero is specified for the length parameter, only the standard message prefix is written on the System Console; the string contained in MESSAGE would not be transmitted.
<i>control</i>	0 (MPE ignores this parameter).
<i>reply</i>	The System 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 System 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. The maximum value for *expectedl* is 31 bytes.

Suspending the Calling Process

The calling process can be suspended with the PAUSE intrinsic. The maximum interval allowed is approximately 2,147,484 seconds (almost 25 days). A PAUSE intrinsic call could be as shown below, where **INT** is a real variable that specifies the amount of time, in seconds, that the process is suspended: **PAUSE(INT);**

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. The CAUSEBREAK intrinsic is the programmatic equivalent of using the **BREAK** key in a session. It 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	:HELP	:RENAME	:SHOWIN
:ALTSEC	:IF	:REPORT	:SHOWJCW
:ALTVSET	:LISTEQ	:RESET	:SHOWJOB
:BUILD	:LISTF	:RESETDUMP	:SHOWME
:COMMENT	:LISTFTEMP	:RESUME	:SHOWOUT
:DEBUG	:LISTVS	:SAVE	:SHOWTIME
:DISCRPS	:MOUNT	:SECURE	:SPEED
:DISMOUNT	:NEWVSET	:SETCATALOG	:STARTSESS
:DSLNE	:PTAPE	:SETDUMP	:STREAM
:DSTAT	:PURGE	:SETJCW	:TELL
:ELSE	:REDO	:SETMSG	:TELLOP
:ENDIF	:RELEASE	:SHOWCACHE	:VSUSER
:FILE	:REMOTE	:SHOWCATALOG	
:GETRIN	:REMOTE HELLO	:SHOWDEV	

(Refer to the *MPE V Commands Reference Manual* (32033-90006) for discussions of the above commands.)

The CAUSEBREAK intrinsic is not valid in a job. A program containing the CAUSEBREAK intrinsic will not break if it is executed from a UDC which specifies the NOBREAK option.

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



Terminating a Process

You can programmatically 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 with a disposition of 0. (For more details, refer to the **FCLOSE** intrinsic discussion in Chapter 2.)

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. All process files still open are closed with a disposition of 0. (Refer to the **FCLOSE** intrinsic discussion in Chapter 2 for more details.)

The **QUIT** intrinsic transmits an abort message to the calling process calling device and terminates the process in an error state. (Refer to “Run-Time Messages” in Appendix A.) 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 is in effect. (Refer to the *MPE V Commands Reference Manual* (32033-90006) for more information on the **:CONTINUE** command.)

Figure 5-12 and Figure 5-13 shows the **QUIT** intrinsic being called if a **READ** statement did not execute properly. The abort message resulting from the **QUIT** intrinsic is shown below:

```
ABORT :program.group.account %0.%26  
PROGRAM ERROR #18 :PROCESS QUIT . PARAM=1
```

Statement 00026000:

```
IF <> THEN QUIT(1);
```

checks for a CCG or CCL condition code and, if one is returned, the **QUIT** intrinsic is called. The abort message is printed and the process is aborted. 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 (PARAM=1). The system Job Control Word, **JCW**, is set to the value %100000, with the **QUIT** parameter as a modifier. In this example, **JCW** would be set to %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. All files still open by any user process are closed with a disposition of 0. (Further details can be found under the `FCLOSE` intrinsic in Chapter 2.)

```

PAGE 0001 HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
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",
00011000 00010 1 5,3,"DIV", 5,3,"SET", 0;
00012000 00016 1 INTEGER ARRAY PARMINFO(0:1);
00013000 00016 1 LOGICAL INTERACTIVE:=FALSE;
00014000 00016 1 INTEGER ACCUM:=0, OPERAND:=0, REQ:="? ",
00015000 00016 1 LGTH, INDX, PARMCNT, TYPE;
00016000 00016 1
00017000 00016 1 INTRINSIC ASCII,BINARY,READ,PRINT,MYCOMMAND,QUIT,WHO;
00018000 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 COMD>>
00026000 00023 1 IF <> THEN QUIT(1); <<CHECK FOR ERR>>
00027000 00026 1 IF COMMAND="END" THEN GO EXIT; <<DONE - EXIT>>
00028000 00040 1 COMMAND(LGTH):=%15; <<CARRIAGE RETN>>
00029000 00043 1
00030000 00043 1 TYPE:=MYCOMMAND(COMMAND,,1,PARMCNT, <<TAKE APART CMD>>
00031000 00050 1 PARMINFO,TABLE);
00032000 00056 1 IF < THEN GO ERROR; <<NO CMD MATCH>>
00033000 00057 1 IF PARMCNT<>1 THEN GO ERROR; <<NO PARAMETERS>>
00034000 00062 1 INDX:=PARMINFO-@COMMAND; <<SUBSCR OF PARM>>
00035000 00065 1 OPERAND:=BINARY(COMMAND(INDX), <<CONVERT PARM>>
00036000 00070 1 PARMINFO(1).(0:8));
00037000 00075 1 IF <> THEN GO ERROR; <<CHECK FOR ERR>>
00038000 00076 1
00037000 00076 1 CASE (TYPE-1) OF <<SELECT OPERATN>>
00040000 00100 1 BEGIN
00041000 00106 2 ACCUM:=ACCUM+OPERAND; <<ADD COMD>>
00042000 00116 2 ACCUM:=ACCUM-OPERAND; <<SUB COMD>>
00043000 00122 2 ACCUM:=ACCUM*OPERAND; <<MUL COMD>>
00044000 00126 2 ACCUM:=ACCUM/OPERAND; <<DIV COMD>>
00045000 00133 2 ACCUM:=OPERAND; <<SET COMD>>
00046000 00136 2 END;

```

Figure 5-12. Using the QUIT Intrinsic (1 of 2)

```

00047000 00143 1  RESULT:
00048000 00143 1      MOVE ANSWER(8):="    ";          <<RESET OLD ANSWER>>
00049000 00155 1      ASCII(ACCUM,10,ANSWER(8));        <<CONVERT ACCUM>>
00050000 00163 1      PRINT(OUTPUT,7,0);                <<OUTPUT NEW ANSWER>>
00051000 00170 1      GO LOOP;                          <<CONTINUE CALC>>
00052000 00171 1  ERROR:
00053000 00171 1      PRINT(ERRMSG,7,0);                <<ERROR MESSAGE>>
00054000 00175      IF NOT INTERACTIVE THEN QUIT(2);    <<NO LIVE USER-QUIT>>
00055000 00201 1      GO LOOP;                          <<CONTINUE CALC>>
00056000 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 5-13. Using the QUIT Intrinsic (2 of 2)

In batch jobs not containing the :CONTINUE command this terminates the job. For more information on the :CONTINUE command refer to the *MPE V Commands Reference Manual* (32033-90006).

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 %100000, with the QUITPROG parameter as a modifier. Thus, in this example, JCW would be set to %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 as 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 disk storage. The maximum stack size value is specified at preparation or run time with the *segsz* 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 of a process running in Privileged Mode is frozen in main memory, as during an input/output operation, either implicitly or explicitly, intrinsics to change the register offsets DL to DB or Z to DB cannot be executed. The stack segment is frozen in main memory implicitly when a user's process interfaces directly with the input/output system. It is frozen in main memory explicitly by a direct call to system intrinsics. 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 reissuing 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. (Refer to Figure 5-14.)

A request for contraction to 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 to the entire 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 10 words of the DL area located adjacent to DB 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 5-15 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. SPL is the only language which can access this area for you.

The program in Figure 5-15 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 (LIFO) 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 that the program was prepared (:PREP command) with a MAXDATA=2000 parameter.

Statement 00018000 of Figure 5-15 sets the DL to DB area to the original area assigned when the process was created (initial DL):

```
TOTALDL:=DLSIZE(0);
```

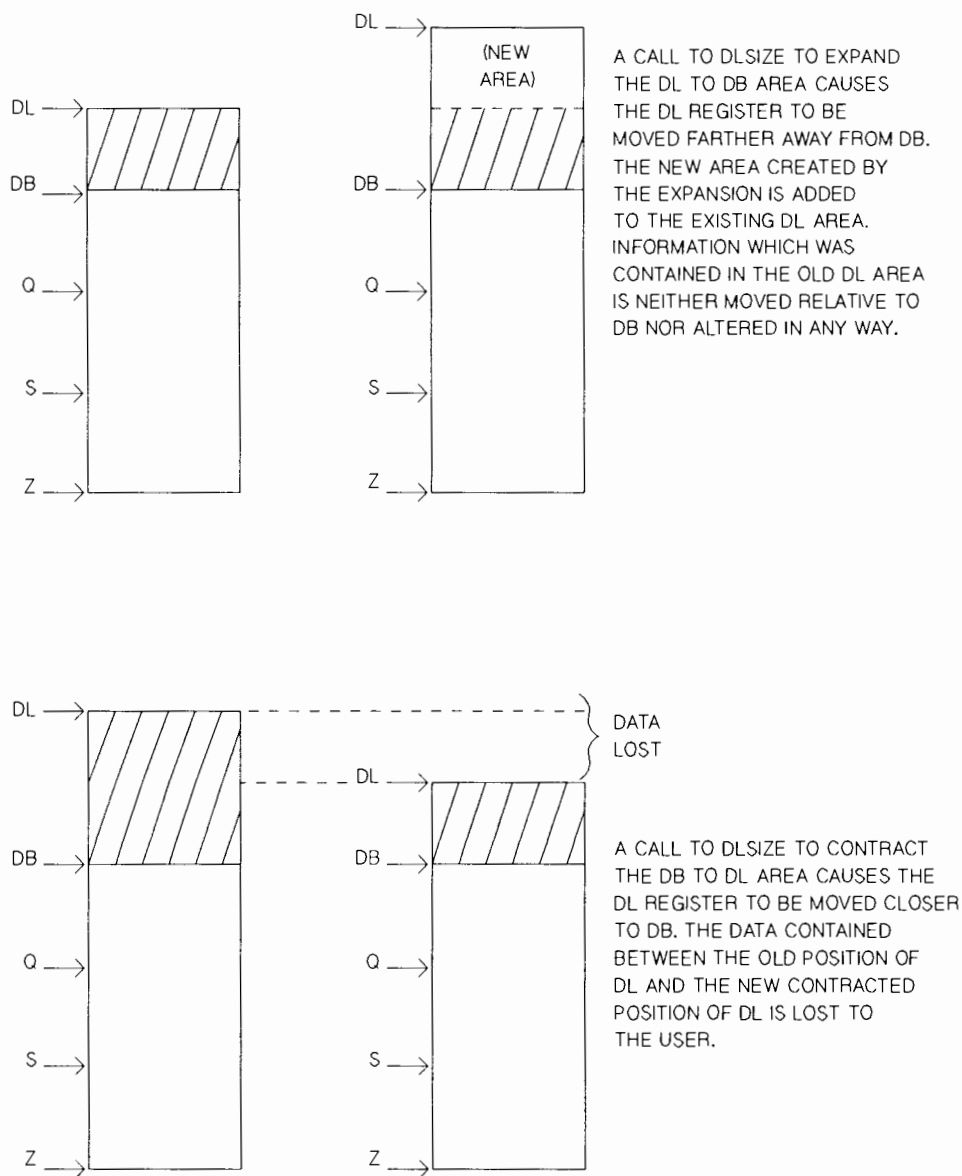


Figure 5-14. Expanding and Contracting the DL to DB Area

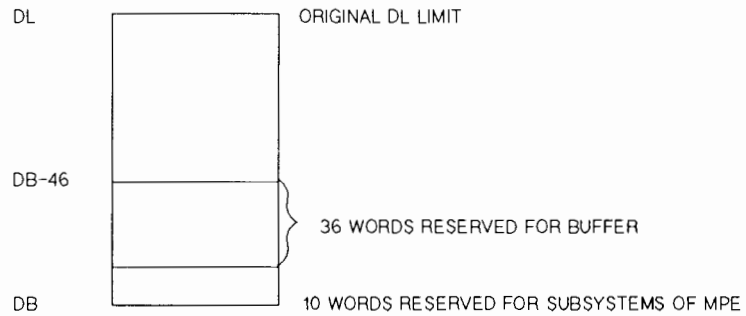

```

PAGE 0001 HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 INTEGER IN,OUT,LGTH,
00004000 00000 1 TOTALDL:=0;
00005000 00000 1 LOGICAL PROMPT:="? ";
00006000 00000 1 LOGICAL POINTER BUFFER:=-46; <<BUFFER:36 ; RESERV:10>>
00007000 00000 1
00008000 00000 1 INTRINSIC FOPEN,DLSIZE,FREAD,FWRITE,QUIT;
00009000 00000 1
00010000 00000 1 <<END OF DECLARATIONS>>
00011000 00000 1
00012000 00000 1 IN:=FOPEN(,%44); <<$STDIN>>
00013000 00007 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00014000 00012 1
00015000 00012 1 OUT:=FOPEN(,%414,1); <<$STDLIST>>
00016000 00022 1 IF < THEN QUIT(2); <<CHECK FOR ERROR>>
00017000 00025 1 RESTART:
00018000 00025 1 TOTALDL:=DLSIZE(0) <<SET DL TO INITL SIZE>>
00019000 00030 1 IF <> THEN QUIT(3); <<CHECK FOR ERROR>>
00020000 00033 1 LINEIN:
00021000 00033 1 PUSH(DL); <<GET CRNT DL SETTING>>
00022000 00034 1 IF TOS>@BUFFER THEN <<NEXT BFR OUTSIDE DL?>>
00023000 00036 1 BEGIN
00024000 00036 2 TOTALDL:=DLSIZE(TOTALDL-128);<<GET MORE DL AREA>>
00025000 00043 2 IF <> THEN QUIT(4); <<CHECK FOR ERROR>>
00026000 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 BUFR>>
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 BUFRS OUT:RESTR>>
00045000 00132 1 @BUFFER:=@BUFFER+36; <<ADDRESS PREVIOUS BUFR>>
00046000 00135 1 GO LINEOUT; <<CONTINUE OUTPUT PHASE>>
00047000 00137 1 EXIT:END.

```

Figure 5-15. Using the DLSIZE Intrinsic (Program DLAREA)

Consider the following illustration of the DL to DB area:



Statement 00006000 in Figure 5-15:

```
LOGICAL POINTER BUFFER:=-46;
```

sets a pointer to DB-46, which is the DB-relative address of the first word in **BUFFER**.

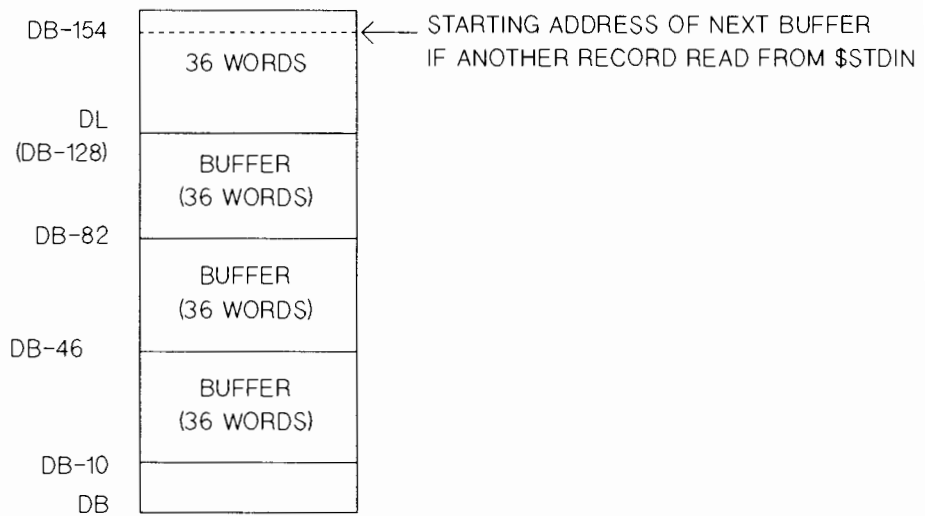
Statement 00021000:

```
PUSH(DL);
```

pushes the DL register contents onto the top of the stack, and statement 00022000:

```
IF TOS>@BUFFER THEN
```

checks to determine whether 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) the address of the first word in **BUFFER**, then **BUFFER** is outside the DL to DB area. The following diagram illustrates this point:



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 four statements in the program beginning on line 00023000 add 128 words to the DL to DB area.

Statements 00027000 and 00028000:

```
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** in statement 00035000.

If **LGTH** = 0, signaling 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.

Statement 00024000:

```
TOTALDL:=DLSIZE(0);
```

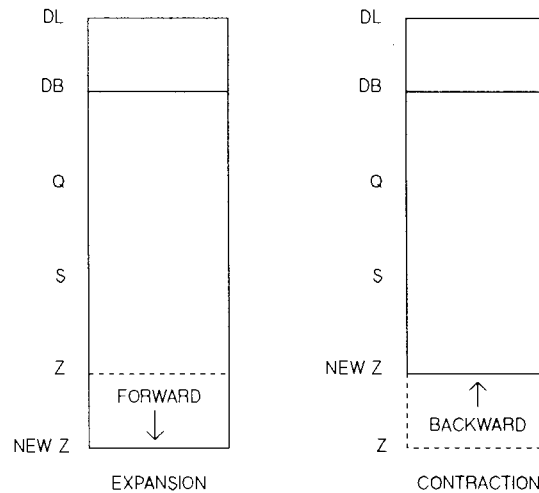
contracts the DL to DB area back to its original size, destroying the contents of all buffers. An **:EOD** entry terminates program execution.

Figure 5-16 shows the results when the program is run.

Changing the Z to DB Area Size

You can alter the size of the current Z to DB area by adjusting the register offset of the Z address from the DB address with the **ZSIZE** intrinsic.

The **ZSIZE** intrinsic moves the Z address forward (expansion) or backward (contraction) as shown below:



If the Z to DB area size requested exceeds the maximum size permitted for the Z to DL (stack) area, only the maximum size allowed is granted.

All changes to the Z to DB area are made in increments or decrements of 128 words; thus the size actually granted may differ from the size requested. For example, if the present Z to DB area size is 128 words, a request for 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.

The validity of a trap procedure, specified by the external-type label of the user trap procedure (*plabels*), depends on the code domain of the caller’s code and executing mode (privileged or nonprivileged), and on the code domain of the *plabel* and its mode (privileged or nonprivileged). The code domains are:

PROG	(User Program)	SSL	(System SL, non-MPE segments)
GSL	(Group SL)	MPSSL	(System SL, MPE segments)
PSL	(Public SL)		

If, at the time of enabling a trap procedure, the code of the caller is:

- Nonprivileged in PROG, GSL, or PSL, *plabel* must be nonprivileged in PROG, GSL, or PSL.
- Privileged in PROG, GSL, or PSL, *plabel* may be privileged or nonprivileged in PROG, GSL, or PSL.
- Privileged or nonprivileged in non-MPE SSL, *plabel* cannot be in any 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);
```

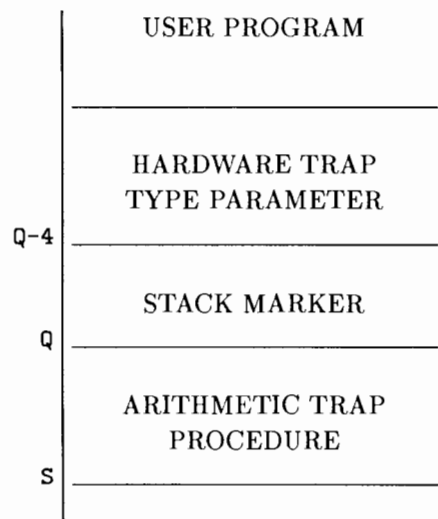
where **STATE** is true (bit (15:1) = 1) to enable the signals from all hardware traps, or false (bit (15:1) = 0) to disable these signals.

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 the right stackop bit in the status word at Q-1 of the Stack Marker, 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

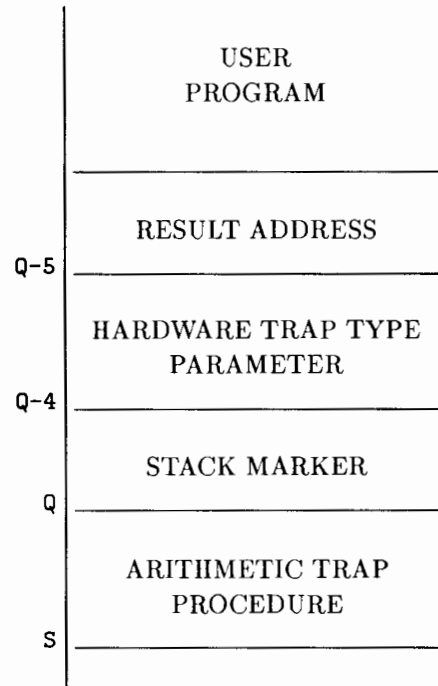
A return from the trap procedure (through an **EXIT 1** instruction) will resume execution in the user code domain at the instruction following that which activated the trap procedure. The condition of the stack when the trap procedure is invoked is:



Extended Precision Floating Point Traps

Bit 10 = Extended Precision Overflow
 9 = Extended Precision Underflow
 8 = Extended Precision Divide By 0

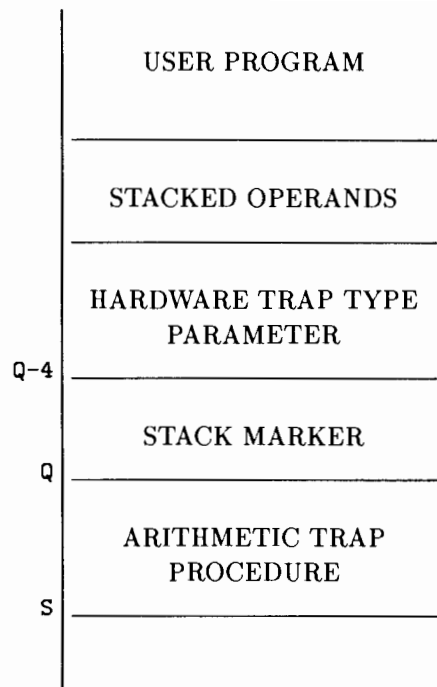
The address of the result operand is left on the stack in Q-5. An **EXIT 2** return will resume execution in the user code domain at the instruction following the one which caused the trap. The condition of the stack when the trap procedure is invoked is:



Commercial Instruction Traps

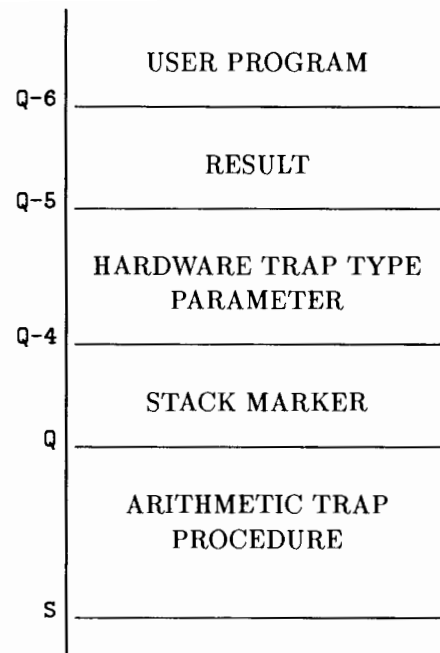
- Bit 7 = Decimal Overflow
- 6 = Invalid ASCII Digit (CVAD)
- 5 = Invalid Decimal Digit
- 4 = Invalid Source Word Count (CVBD)
- 3 = Invalid Decimal Operand Length
- 2 = Decimal Divide By 0

The parameters 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 5-17. The procedure `FDIVZRO` is a trap procedure to which control is passed if a divide by zero operation is attempted while a Floating Point Divide By 0 software trap is enabled. Statement 00027000 of Figure 5-17 enables the Floating Point Divide By 0 trap. The parameter `%1` (bit (15:1) = 1) enables only the Floating Point Divide By 0. The `@FDIVZRO` passes the trap procedure as a parameter, and `DUMMY1` and `DUMMY2` are dummy parameters.

Statement 00031000 of Figure 5-17 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:



The two-word floating point value of **RESULT** has been left at Q-5/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.

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1982, 4:35
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 ANS>>
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;                          <<DEL 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/0. TRAP>>
00028000 00005 1      IF < THEN QUIT(1);                  <<CHECK FOR ERROR>>
00029000 00010 1
00030000 00010 1      PRINT(DIVMSG,8,0);                    <<DIVIDE HEADING>>
00031000 00014 1      RESULT:=NUM/DENOM;                    <<DIVIDE>>
00032000 00020 1
00033000 00020 1      PRINT(ADDMSG,7,0);                    <<ADD HEADING>>
00034000 00024 1      RESULT:=RESULT+RESULT;                <<ADD>>
00035000 00030 1 END.

```

Figure 5-17. Using the SARITRAP Intrinsic (Program ATRAP)

Statement 00020000 of Figure 5-17 deletes one word from the stack (the TRAP parameter at Q-4) and returns to the program leaving QUOTIENT (whose value is either %3777777777D or %1777777777D) on the top of stack (once the stack marker for FDIVZRD and the trap parameter are deleted).

When statement 00034000:

```
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**). In a practical program 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 #2 :FLOATING POINT OVERFLOW
```

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

```
:
```

Library Trap

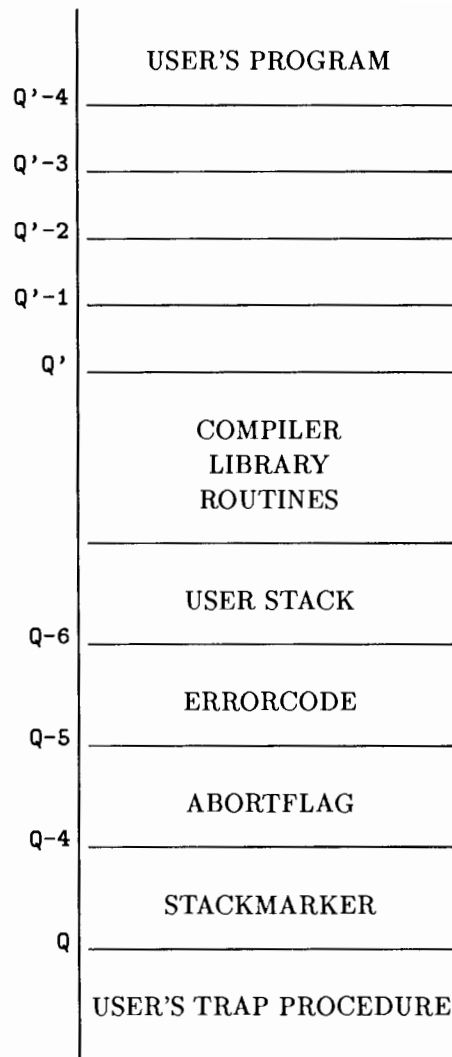
The software library trap reacts to major errors that occur during execution of procedures from the System Library. When a user program begins execution, this trap is disabled automatically. You can enable (or disable) it with the **XLIBTRAP** intrinsic. When enabled, the library trap passes control to a trap procedure in the event of an error. This procedure, in turn, returns four words to the user program which contain 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 Reference Manual* (30000-90028).

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 user program. If no *plabel* existed previously, 0 is returned.

When a library trap procedure is invoked, the condition of the stack is:



where the variables shown in Q-4 to Q-6 represent the following:

USERSTACK	A word pointer to Q' 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 <i>HP 3000 Compiler Library Reference Manual</i> (30000-90028).
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:

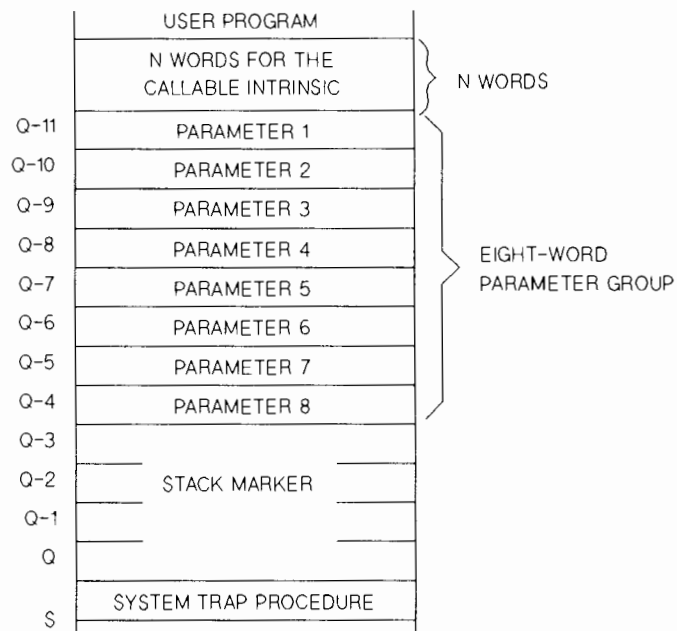
- **Illegal access.** An attempt by a user to access an intrinsic for which he does not have access capability.
- **Illegal parameters.** The passing to an intrinsic of parameters that are not defined for the user's environment.
- **Illegal environment.** The DB register is not currently pointing to 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(NEWPLABEL,OLDPLABEL);
```

where **NEWPLABEL** 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:

BITS	0	9	10	15	
Q-11	I			N	PARAMETER 1
Q-10	P				PARAMETER 2
Q-9	P-1		E-1		PARAMETER 3
Q-8	P-2		E-2		PARAMETER 4
Q-7	P-3		E-3		PARAMETER 5
Q-6	P-4		E-4		PARAMETER 6
Q-5	P-5		E-5		PARAMETER 7
Q-4	P-6		E-6		PARAMETER 8
	7		8		

where the variables represent the following:

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. 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. For example, it may 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, is:

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 CONTROL key and the Y key simultaneously.

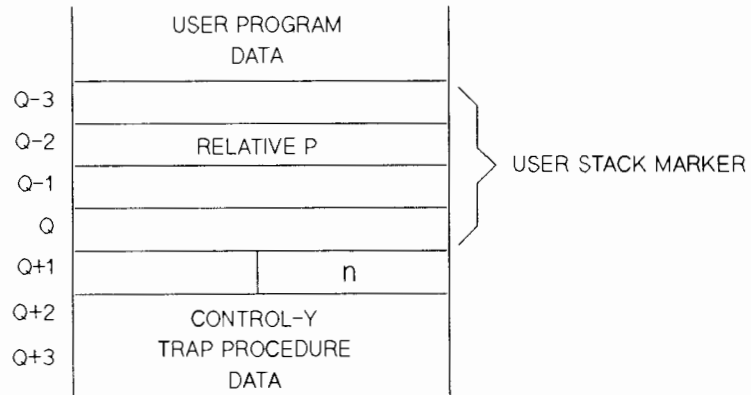
When more than one process is currently running within your process tree structure, the CONTROL-Y signal interrupts the last process to enable the trap.

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 for interaction. This procedure executes in the same mode (either privileged or nonprivileged) as the interrupted user program. The restrictions for operating the trap procedure are: you cannot trap from Privileged to non-Privileged Mode; the program Group SL/Public SL can't trap into System SL; the System SL can trap into System SL only; if you are in MPE, you can trap into either MPE or a non-MPE mode; if you are not in MPE, you can only trap into a non-MPE mode.
3. Control returns from the trap procedure to the interrupted program or procedure. If the interrupted program or procedure was waiting for 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 entered 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 to be deleted from the stack (in addition to the stack marker) 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 0** instruction.

Note



If you are a user with Privileged Mode capability, you should be aware of the following:

- If your interrupted code was executing in Privileged Mode, your trap procedure must also be executed in Privileged Mode, and therefore must have Privileged Mode capability.
- 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 5-18 shows a program containing a CONTROL-Y trap procedure. Statements 00022000 through 00024000:



```
LOOP:
    CNTR:=CNTR+1D;
    IF CNTR<3000000D THEN GO LOOP;
```

increment a double-word counter by 1 each time the loop is executed. When the counter reaches the value 3000000 (decimal), program execution terminates.

The CONTROL-Y trap procedure, beginning with statement 00010000:

```
PROCEDURE CONTROLY;
```

assumes control whenever CONTROL-Y is entered from the terminal. The trap procedure executes, then control passes back to the loop.

Statement 00012000:

```
INTEGER SDEC=Q+1;
```

equates SDEC to Q+1. The rightmost byte of Q+1 contains the number of words on the stack to be deleted (in addition to the stack marker) 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 may be called at any time from any procedure after the CONTROL-Y trap has been invoked. An EXIT instruction must be built and statement 00016000:

```
TOS:=%31400+SDEC;
```

accomplishes this, loading the octal value %31400 plus the value of SDEC (Q+1) onto the top of the stack. Statement 00017000:

```
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 statement 00020000:

```
XCONTRAP(@CONTROLY,DUMMY);
```

The @CONTROLY parameter informs the system that a procedure (CONTROLY) is being passed as a parameter.

```

PAGE 0001  HP32100A.08.01  [4W]  (C)  HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLINIT
00002000 00000 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 INTRINSIC PRINT,XCONTRAP,QUIT,DASCII,RESETCONTROL;
00009000 00020 1
00010000 00020 1 PROCEDURE CONTROLY;
00011000 00000 1 BEGIN
00012000 00000 2 INTEGER SDEC=Q+1;
00013000 00000 2 LGTH:=DASCII(CNTR,10,BMSG(20));<<CONVERT COUNTER>>
00014000 00007 2 PRINT(MSG,16,0); <<OUTPUT COUNTER>>
00015000 00013 2 RESETCONTROL; <<REARM CONTROL Y TRAP>>
00016000 00014 2 TOS:=%31400+SDEC; <<BUILD EXIT INSTRUCTION>>
00017000 00016 2 ASSEMBLE(XEQ 0); <<EXECUTE EXIT>>
00018000 00017 2 END;
00019000 00000 1 PRINT(HEADING,11,0); <<PROGRAM ID>>
00020000 00004 1 XCONTRAP(@CONTROLY,DUMMY); <<ARM CONTROL Y TRAP>>
00021000 00007 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00022000 00012 1 LOOP:
00023000 00012 1 CNTR:=CNTR+1D; <<DOUBLE INCREMENT>>
00024000 00023 1 IF CNTR<3000000D THEN GO LOOP;<<CONTINUOUS LOOP>>
00025000 00027 1 END.

```

Figure 5-18. Using the XCONTRAP Intrinsic (Program CONTY)

The results of executing the program are shown below:

```
:RUN CONTY__
```

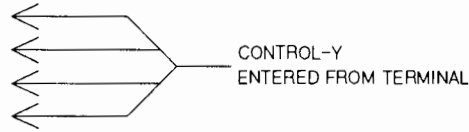
```
CONTROL Y TRAP EXAMPLE
```

```
COUNTER CURRENTLY = 125153
```

```
COUNTER CURRENTLY = 1093923
```

```
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.

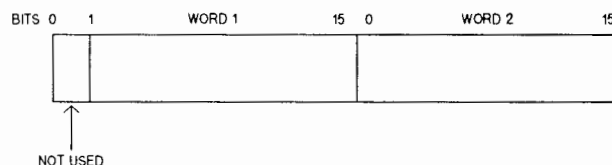
This quantity is reset to zero on 24-day intervals at 12 o'clock midnight. Detection and correction of this case between two calls to **TIMER** (less than 24 days apart) for computing an elapsed time interval can be done by adding 2,073,600,000 (the number of milliseconds in 24 days) to the result when subtracting a current **TIMER** count from a previous count with a negative result.

Figure 5-19 and Figure 5-20 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.)

Statement 00027000:

```
CBUF(5) := INTEGER(TIMER) . (11:5) + %73;
```

calls the **TIMER** intrinsic to obtain the timer count. A double-word quantity is returned as follows:

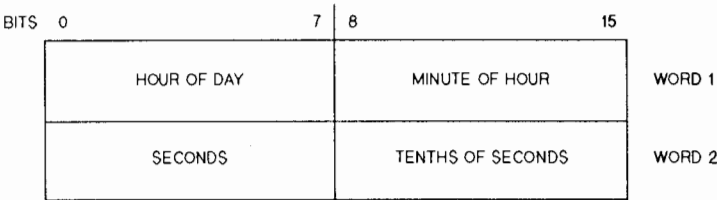


The **INTEGER** function strips Word 1 from this quantity, leaving a 16-bit integer value. The low-order five bits change value most rapidly, and are used to obtain a decimal number from 0 to 32.

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 sixth position of byte array **CBUF** (**CBUF(5)**). The **FWRITE** intrinsic displays this character, and the string "TYPE" on the terminal. (**CBUF** and **BUFR** have been equated in statements 00006000 and 00007000.)

Obtaining the Current Time

The **CLOCK** intrinsic returns the actual time as a double-word. The first word contains the hour and minute of the hour, and the second word contains seconds and tenths of seconds, as follows:



By using the intrinsic call **TIME:=CLOCK;**, the above information would be returned to the double-word identifier **TIME**.

```

PAGE 0001 HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00001 0 BYTE ARRAY INNAME(0:5):="INPUT ";
00004000 00004 1 BYTE ARRAY OUTNAME(0:6):="OUTPUT ";
00005000 00005 1 INTEGER IN,OUT,LGTH,DUMMY,TIME,TIMEOUT:=10;
00006000 00005 1 ARRAY BUFR(0:3):="TYPE X",0;
00007000 00004 1 BYTE ARRAY CBUF(*)=BUFR;
00008000 00004 1 ARRAY INSTRUCTIONS(0:34):="REACTION TIMER: "%6412,
00009000 00011 1 "TYPE THE REQUESTED CHARACTER AS QUICKLY AS YOU CAN. ";
00010000 00043 1 ARRAY MSG(0:24):="TRY AGAIN? (Y/N)","WRONG CHARACTER.",
00011000 00020 1 "%6412,"YOU'RE TOO SLOW!";
00012000 00031 1 ARRAY RESPONSE(0:16):="REACTION TIME: MILLISECONDS";
00013000 00021 1 BYTE ARRAY CRESP(*)=RESPONSE;
00014000 00021 1 INTRINSIC FOPEN,FREAD,FWRITE,FCONTROL,ASCII,TIMER,QUIT;
00015000 00021 1 <<END OF DECLARATIONS>>
00016000 00021 1 IN:=FOPEN(INNAME,%45); <<$STDIN>>
00017000 00007 1 IF < THEN QUIT(1); <<CHECK FOR ERROR>>
00018000 00012 1 OUT:=FOPEN(OUTNAME,%414,%1); <<$STDLIST>>
00019000 00022 1 IF < THEN QUIT(2); <<CHECK FOR ERROR>>
00020000 00025 1 FWRITE(OUT,INSTRUCTIONS,35,0); <<USER DIRECTIONS>>
00021000 00032 1 IF < THEN QUIT(3); <<CHECK FOR ERROR>>
00022000 00035 1 LOOP:
00023000 00035 1 FCONTROL (IN,21,DUMMY); <<ENABLE TIMER READ>>
00024000 00041 1 IF < THEN QUIT(4); <<CHECK FOR ERROR>>
00025000 00044 1 FCONTROL(IN,4,TIMEOUT); <<ENABLE TIMEOUT>>
00026000 00050 1 IF < THEN QUIT(5); <<CHECK FOR ERROR>>
00027000 00053 1 CBUF(5):=INTEGER(TIMER).(11:5)+%73;<<GENERATE CHARACTER>>
00028000 00062 1 FWRITE (OUT,BUFR,3,%320); <<REQUEST USER INPUT>>
00029000 00067 1 IF < THEN QUIT(6); <<CHECK FOR ERROR>>
00030000 00072 1 LGTH:=FREAD(IN,BUFR(3),-1); <<READ CHARACTER>>
00031000 00101 1 IF < THEN <<TIMEOUT OCCURRED>>
00032000 00102 1 BEGIN
00033000 00102 2 FWRITE(OUT,MSG(16),9,0); <<TOO SLOW MESSAGE>>
00034000 00110 2 IF < THEN QUIT(7) ELSE GO NEXT;<<CHECK FOR ERROR>>
00035000 00120 2 END;
00036000 00120 1 IF CBUF(5)<>CBUF(6) THEN <<INCORRECT CHARACTER>>
00037000 00126 1 BEGIN
00038000 00126 2 FWRITE(OUT,MSG(8),8,0); <<WRONG CHARACTER MSG>>
00039000 00134 2 IF < THEN QUIT(8) ELSE GO NEXT;<<CHECK FOR ERROR>>
00040000 00141 2 END;
00041000 00141 1 MOVE RESPONSE(7):=" "; <<RESET RESPONSE TIME>>
00042000 00153 1 FCONTROL(IN,22,TIME); <<READ INPUT TIME>>
00043000 00157 1 IF <> THEN QUIT(9); <<CHECK FOR ERROR>>
00044000 00162 1 ASCII(TIME*10,10,CRESP(15)); <<CONVERT TIME>>
00045000 00171 1 FWRITE(OUT,RESPONSE,17,0); <<REACTION TIME>>
00046000 00177 1 IF < THEN QUIT(10); <<CHECK FOR ERROR>>

```

Figure 5-19. Using the TIMER Intrinsic (1 of 2)

```

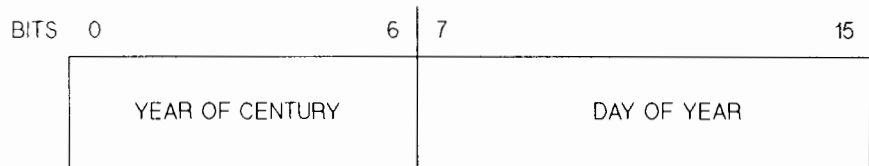
00047000 00202 1  NEXT:
00048000 00202 1      FWRITE(OUT,MSG,8,%320);          <<CONTINUE TEST?>>
00049000 00207 1      IF < THEN QUIT(11);              <<CHECK FOR ERROR>>
00050000 00212 1      FREAD(IN,BUFR(3),-1);            <<GET Y/N ANSWER>>
00051000 00220 1      IF < THEN QUIT(12);              <<CHECK FOR ERROR>>
00052000 00224 1      IF CBUF(6)="Y" THEN GO LOOP;      <<Y-CONTINUE TEST>>
00053000 00232 1  END.

```

Figure 5-20. Using the TIMER Intrinsic (2 of 2)

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

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 program shown in Figure 5-21 illustrates the use of these intrinsics.

```

PAGE 0001 HP32100A.08.01 [4W] HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00004000 00000 1   ARRAY F'DATE(0:8);
00005000 00000 1       BYTE ARRAY FDATE(*)=F'DATE;
00006000 00000 1   ARRAY F'TIME(0:4);
00007000 00000 1       BYTE ARRAY FTIME(*)=F'TIME;
00008000 00000 1   ARRAY DATE'TIME(0:13);
00009000 00000 1       BYTE ARRAY DATETIME(*)=DATE'TIME;
00010000 00000 1
00011000 00000 1   LOGICAL DATE;
00012000 00000 1   DOUBLE TIME;
00013000 00000 1
00014000 00000 1   INTRINSIC CALENDAR,CLOCK,FMTCALENDAR,
00015000 00000 1       FMTCLOCK,FMTDATE,PRINT;
00016000 00000 1
00017000 00000 1   DATE:=CALENDAR;
00018000 00003 1   TIME:=CLOCK;
00019000 00006 1
00020000 00006 1   FMTCALENDAR(DATE,FDATE);
00021000 00011 1   PRINT(F'DATE,-17,%60);
00022000 00015 1
00023000 00015 1   FMTCLOCK(TIME,FTIME);
00024000 00020 1   PRINT(F'TIME,-8,%60);
00025000 00024 1
00026000 00024 1   FMTDATE(DATE,TIME,DATETIME);
00027000 00030 1   PRINT(DATE'TIME,-27,%0);
00028000 00034 1
00029000 00034 1   END.

```

Figure 5-21. Using the FMTCALENDAR, FMTCLOCK, and FMTDATE Intrinsics

Job Control Words

There are three classes of Job Control Words (JCWs) in the system. User-defined JCWs are named and assigned values exclusively by the user. System-defined JCWs are named by the system and can be assigned values either by the system (under certain circumstances) or by the user. System-reserved JCWs are assigned values exclusively by the system.

Any attempt by a user to assign a value to a system-reserved JCW (with either SETJCW or PUTJCW) will result in an error. The values of system-reserved JCWs can be returned with FINDJCW. The values of these JCWs can be used in assignments to other JCWs, but the user cannot assign values to a system-reserved JCW. System-reserved JCWs are available on MPE V releases G.01.00 or later.

The name of the system-reserved JCWs are reserved words, that is, a user cannot have a JCW of the same name. The following is a list of the system-reserved JCWs and their meanings.

HPDATE	Indicates the day of the month. The possible range of values for HPDATE is 1 through 31, inclusive.
HPDAY	Indicates the day of the week. The possible range of values for HPDAY is 1 through 7, inclusive, with 1 indicating Sunday and 7 indicating Saturday.
HPHOUR	Indicates the hour of the day on a 24-hour basis. The possible range of values for HPHOUR is 0 through 23, inclusive.
HPMINUTE	Indicates the minute of the hour. The possible range of values for HPMINUTE is 0 through 59, inclusive.
HPMONTH	Indicates the month of the year. The possible range of values for HPMONTH is 1 through 12, inclusive, with 1 indicating January.
HPYEAR	Contains the year of the century.

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 job or session 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 (0:1)=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 (1:15) 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 %140000, (bits (0:2) only), the "CIERR 976" message is replaced by "CIERR 989, PROGRAM ABORTED PER USER REQUEST". Refer to the *MPE V Commands Reference Manual* (32033-90006) 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.

As an example, consider a job where two processes in the same process tree pass information to each other through the Job Control Word. In one

process, you transmit the contents of the word **PROCLNK** to **JCW**. Process A sets the Job Control Word to **PROCLNK** as follows:

```
SETJCW(PROCLNK);
```

When process B is executed, it obtains the value of **JCW** through the **GETJCW** intrinsic. In this case, the contents of **JCW** is 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** because MPE uses **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 **JCWs**, is shared by all processes in a job or session; 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 value of the **JCW** is updated to the value passed by the **PUTJCW** intrinsic call. If a **JCW** of that name 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 of **PUTJCW** 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. |
| 4 | Error. Attempted to assign a value to an MPE-defined JCW value mnemonic (OK , WARN , FATAL , or SYSTEM). |
| 5 | Error. Cannot assign a value to a system-reserved JCW . |

The **FINDJCW** intrinsic is used to scan the **JCW Table** for a given **JCW** 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 of **FINDJCW** 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 Facility

The MPE message facility consists of a message catalog (**CATALOG.PUB.SYS**), the HELP subsystem catalog (**CICAT.PUB.SYS**, containing descriptions of all MPE commands), many user message catalogs, a program (**MAKECAT.PUB.SYS**) for building message catalogs, and an intrinsic (**GENMESSAGE**) used to insert parameters in messages from a catalog and print the resulting message.

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 format. The message 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 nonblank 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 that need not be contiguous but must be in the file in ascending order. After processing by the program **MAKECAT**, the catalog file contains records of 80 bytes with a blocking factor of 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      (Comment)
$FIRST LINE, AND THE SECOND LINE IS "HP32002"              (Comment)
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:

Default entry point	Reads from the input file and builds a temporary file with the formal designator CATALOG . Also renames any old temporary CATALOG to CATnn , using an incremental numbering scheme (for example, CAT1 , CAT2).
BUILD	To use BUILD as the entry point, you must logon as MANAGER.SYS . Reads from the input file, builds the system message catalog (formal designator CATALOG), and installs the message system. Existing catalog is renamed CATnn using the incremental numbering scheme as for the default entry point. 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 disk address of CATALOG are placed in the system global area. The message system may be installed while the system is running.
DIR	To use DIR as the entry point you must logon as MANAGER.SYS . 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 disk 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 Used to build the HELP catalog. Reads input file and builds a HELP catalog (formal designator **HELPCAT**).

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

```
:FILE INPUT=CAT15
:RUN MAKECAT.PUB.SYS      ** No entry point **
**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. Logon 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 GENMESSAGE to Insert Parameters in Messages

The GENMESSAGE intrinsic can be used to access the MPE message facility. GENMESSAGE is called with a set number, a message number, and any values to be substituted in the message.

The message facility fetches the message from a message catalog, inserts parameters, and then routes the message to a file. It then returns the message in a buffer to the calling program, and/or prints the message on \$STDLIST.

GENMESSAGE expects the catalog file to be a standard 80-byte Editor file with valid data, including continuation characters "&" and "%", to be in positions 1 through 72. Data found in positions 73 and above will not be included when the message is formed to go to the file or message buffer.

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.

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 *param1*, *param2*, *param3*, *param4*, and *param5* parameters in the **GENMESSAGE** intrinsic call and are inserted in the message wherever a “!” is found. Parameters are inserted in the following order: *param1* substitutes for the leftmost “!” in the message, *param2* for the next leftmost, and so forth. If *param(n)* is present, *param(n-1)* must be present (that is, you cannot specify *param3* unless *param1* and *param2* are specified.)

Figure 5-22 contains a simple program that inserts the value 95 into message number 201 in message set 1 in the message catalog **CATALOG.PUB.SYS**. The complete message is then 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 (201) are included as parameters in the **GENMESSAGE** call. The parameter *paramask* is set to **%10000** and *param1* (**NUMBER**) has the value 95. The complete message is returned in **BUFF**, which is then printed on the terminal with the **PRINT** intrinsic.

Application Message Facility

Native Language Support provides the capability for programmers to produce localized applications. Translated program messages may be printed in the language of a user, and data manipulation may be done according to the rules of a particular language.

Information on **GENCAT**, the application message facility, appears in the *Native Language Support Reference Manual* (32414-90001).

```

PAGE 0001  HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLINIT
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,PRINTFILEINFO,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,1,201,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
LOG FILE NUMBER 95 IS ON
END OF PROGRAM

```

Figure 5-22. GENMESSAGE Intrinsic Example

Section Divider

A. MPE Diagnostic Messages



MPE Diagnostic Messages

Programs running under MPE at any batch input device or terminal may return the following types of error messages:

- Command Interpreter Error Messages which report fatal errors that occur during the interpretation or execution of an MPE command.
- Command Interpreter Warning Messages which report unusual conditions that occur during command interpretation or execution but that may not necessarily be detrimental to the processing of the job or session.
- Run-Time Messages which denote conditions that abort the running program, unless an appropriate error trap has been enabled.
- User Messages which are sent to you by other users currently running jobs or sessions.
- Operator Messages which are sent to you by the System Operator.
- System Messages which denote miscellaneous conditions that terminate or otherwise affect the job/session, such as an abort requested by the System Operator.

Other messages may be received only at the System Console, such as System Operator messages and System Failure messages:

- System Operator Messages
 - Status Messages indicate the current status of jobs/sessions or input/output devices.
 - Input/Output Messages request service for, and report errors on, input/output devices.
 - User Messages, sent by users to the System Operator.
- System Failure Messages
 - 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 by the MPE internal interrupt structure (arithmetic traps, bounds violations, stack overflow, etc) are “program errors” and are described in Table A-1>.
- Explicit calls to the `QUIT` intrinsic.
- Explicit calls to the `QUITPROG` intrinsic.
- Violations of other callable intrinsics (listed in Table A-2), such as passing of illegal parameters or the invoking of an intrinsic without having the required capability class.

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 transmits a run-time (abort) message to the user’s output device and terminates the user’s process. 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 aborted or terminated (if no `:CONTINUE` command overrides termination).

If the aborted program 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 (for example, 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 error messages is:

ABORT:*pname.segment.location:sname.segment.location*
|<—————>||<—————>|
 p-field *s-field*

 {=}
msgtype#msgno:<message>[.param {#} number]

|<—————>|
 m-field (from 1 to 7 lines)

Where:

<i>p-field</i>	Is the location of the last instruction executed.
<i>s-field</i>	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:

<i>pname</i>	The name of the program file containing the user's program, 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.
<i>sname</i>	The symbolic name of the SL in which the segment exists: SYSL - System SL PUSL - Public SL GRSL - Group SL
<i>segment</i>	The logical number of the code segment relating to either the program or SL, whichever is appropriate.
<i>location</i>	The location in the code segment. This is expressed in terms of the displacement (P-PB), where PB is 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 subfields previously described will be output containing a question mark (?).

The *m-field* contains the error message text. The parameters within the *m-field* are:

msgtype is one of:

PROGRAM TYPE
ERROR:INTRINSIC
RUN-TIME ERROR
CREATE ERROR
ACTIVATE ERROR
SUSPEND ERROR
MYCOMMAND ERROR
LOCKGLORIN ERROR
LOADER ERROR
FILESYSTEM ERROR

and corresponds to the names listed in Table A-1 through Table A-9.

For a listing of File System Errors refer to the discussion on the FCHECK intrinsic in Section 2.

<i>msgno</i>	A message number in the appropriate message table.
<i>message</i>	The text of the message.
<i>number</i>	The number of the invalid parameter passed to an intrinsic (the message will read: PARAM=).

Some examples of run-time messages are:

BINARY was called with an invalid byte address:

```
ABORT :BIN.ED.MPE.%0.%12
ERROR :INTRINSIC#62:BINARY
RUN-TIME ERROR#5 :PARAMETER ADDRESS VIOLATION.PARAM #1
```

The program was in an infinite loop doing a DUP instruction:

```
ABORT :OV.ED.MPE%0.%177777
PROGRAM ERROR#20 :STACK OVERFLOW
```

A return was made from a nonprivileged segment to a privileged segment:

```
ABORT :PRIV.ED.MPE%0.%3
PROGRAM ERROR #6 :PRIVILEGED INSTRUCTION
```

The program called the QUIT intrinsic with a parameter of 15:

```
ABORT :QUIT.ED.MPE.%0.%1
PROGRAM ERROR #18 :PROCESS QUIT.PARAM=15
```

Nearly all CST entries were allocated and the program tried to create a process which required more CSTs than were available:

```
ABORT :EDITOR.PUB.SYS.%2.%7
ERROR :INTRINSIC #100:CREATE
CREATE ERROR #30 :LOAD ERROR
LOADER ERROR#65 :UNABLE TO OBTAIN CST ENTRIES
```

The program tried to activate a nonexistent process:

```
ABORT :EDITOR.PUB.SYS.%2.%13
ERROR :INTRINSIC #104: ACTIVATE
ACTIVATE ERROR #21 :ACTIVATION OF MAIN PROCESS NOT ALLOWED
```

Table A-1. PROGRAM TYPE 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
18	PROCESS QUIT=<number>
19	PROGRAM QUIT=<number>
20	STACK OVERFLOW
21	PROGRAM KILLED
22	INVALID STACK MARKER
23	ADDRESS VIOLATION
24	BOUNDS VIOLATION
25	NONRESPONDING MODULE
26	DATA PARITY
27	MEMORY PARITY
28	SYSTEM PARITY
29	STACK UNDERFLOW
30	CST VIOLATION
31	STT VIOLATION

LOGIC ERROR IN THE PROGRAM

<number> is the value passed to the QUITPROG or QUIT intrinsic by the terminating process. (This value is output only if it is not zero.)

Logic error in the program. Probably looping and adding to the stack. May require larger MAXDATA when preparing program.

Program aborted from an external source.

Possible logic error in program.

Possible hardware problem.

Logic error in program. Probably invalid CST or STT discovered by hardware. Explicit PCAL from TOS may have referenced nonexistent CST or STT. May be bad program file.

Table A-2. "ERROR:INTRINSIC" Message Numbers

MESSAGE NO.	INTRINSIC	MESSAGE NO.	INTRINSIC
1	FOPEN	6	FPOINT
2	FREAD	7	FREADDIR
3	FWRITE	8	FCLOSE
4	FUPDATE	10	FCHECK
5	FSPACE	11	FGETINFO
12	FREADSEEK	74	DBINARY
13	FCONTROL	75	DASCII
14	FSETMODE	76	QUIT
15	FLOCK	77	STACKDUMP
16	FUNLOCK	78	SETDUMP
17	FRENAME	79	RESETDUMP
18	FRELATE	80	LOADPROC
19	FREADLABEL	81	UNLOADPROC
20	FWRITELABEL	82	INITUSLF
21	PRINTFILEINFO	83	ADJUSTUSLF
22	IOWAIT	84	EXPANDUSLF
23	FINTEXT	85	PUTJCW
24	FLABELINFO	86	FINDJCW
25	FINSTATE	87	GETINFO
30	GETLOCRIN	99	DEBUG
31	FRELOCRIN	100	CREATE
32	LOCKLOCRIN	101	CREATPROCESS
33	UNLOCKLOCRIN	102	KILL
34	LOCKGLORIN	103	SUSPEND
35	UNLOCKGLORIN	104	ACTIVATE
36	LOCRINOWNER	105	GETORIGIN
40	TIMER	106	MAIL
42	PROCTIME	107	SENDMAIL
43	CALENDAR	108	RECEIVEMAIL
44	CLOCK	109	FATHER
45	PAUSE	110	GETPROCINFO
50	XARITRAP	111	PROCINFO
51	ARITRAP	112	GETPROCID
52	XLIBTRAP	120	GETPRIORITY
53	XSYSTRAP	130	GETDSEG
54	XCONTRAP	131	FREEDSEG
55	RESETCONTROL	132	DMOVEOUT
56	CAUSEBREAK	133	DMOVIN
60	TERMINATE	134	ALTDSEG
61	CTranslate	135	DLSIZE
62	BINARY	136	ZSIZE
63	ASCII	139	SWITCHDB
64	READ	180	JOBINFO

Table A-2.
"ERROR:INTRINSIC" Message Numbers (continued)

MESSAGE NO.	INTRINSIC	MESSAGE NO.	INTRINSIC
65	PRINT	191	PTAPE
66	PRINTOP	200	GETPRIVMODE
67	PRINTOPREPLY	201	GETUSERMODE
68	COMMAND	210	OPENLOG
69	WHO	211	WRITELOG
70	SEARCH	212	CLOSELOG
71	MYCOMMAND	214	LOGSTATUS
72	SETJCW	215	LOGINFO
73	GETJCW	305	FERRMSG

Table A-3. "RUN-TIME" Error Messages

MESSAGE NO.	MESSAGE
1	ILLEGAL DB REGISTER
2	ILLEGAL CAPABILITY
3	OMITTED PARAMETER
4	INCORRECT 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. They are caused by a logic error in the program.

Table A-4. "CREATE" Error Messages

UNKNOWN SUBQUEUE NAME (CREATE ERROR 20)
SUBQUEUE 'A' REQUESTED WITHOUT FROZEN STACK (CREATE ERROR 21)
INSUFFICIENT CAPABILITY FOR NONSTANDARD 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 ERROR 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 A-5. "ACTIVATE" Error Messages

ACTIVATION OF SYSTEM PROCESS NOT ALLOWED (ACTIVATE ERROR 20)
ACTIVATION OF MAIN PROCESS NOT ALLOWED (ACTIVATE ERROR 21)

Table A-6. "SUSPEND" Error Messages

INSUFFICIENT CAPABILITY (SUSPEND ERROR 20)
--

Table A-7. "MYCOMMAND" Error Messages

PARSED PARAM OF COMIMAGE > 255 CHARACTERS

Table A-8. "LOCKGLORIN" Error Messages

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 A-9. "LOADER" Error and Warning Messages

ILLEGAL 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 FILE'S EXTENT MAXIMUM MUST BE ONE (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)
TEMPORARY PROGRAM FILE ILLEGAL (LOAD ERR 46)
UNABLE TO OPEN SYSTEM SL FILE (LOAD ERR 50)
UNABLE TO OPEN PUBLIC SL FILE (LOAD ERR 51)

Table A-10.
"LOADER" Error and Warning Messages (Continued)

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)
UNABLE TO OBTAIN CSTX ENTRIES (LOAD ERR 69)
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)
ILLEGAL MAXDATA (LOAD ERR 77)
PROGRAM ALREADY ALLOCATED (LOAD ERR 80)
ILLEGAL PROGRAM ALLOCATION (LOAD ERR 81)
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 = S (LOAD WARN 89)
PROGRAM LOADED WITH LIB = P (LOAD WARN 90)
PROGRAM LOADED WITH LIB = G (LOAD WARN 91)
ATTEMPTING TO ALLOCATE OR DEALLOCATE PROGRAM FROM NON-SYSTEM & DISC (LOAD ERR 92)
UNABLE TO MOUNT PROGRAM FILE'S HOME VOLUME SET (LOAD ERR 93)
UNABLE TO MOUNT SYSTEM SL'S HOME VOLUME SET (LOAD ERR 94)
UNABLE TO MOUNT PRIVATE SL'S HOME VOLUME SET (LOAD ERR 95)
UNABLE TO MOUNT GROUP SL'S HOME VOLUME SET (LOAD ERR 96)
UNABLE TO LOAD REMOTE PROGRAM FILE (LOAD ERR 97)
UNABLE TO CONVERT OLD FORMAT (LOAD ERR 98)
UNABLE TO OBTAIN DST FOR LOGICAL MAP (LOAD ERR 99)
TOO MANY MAPPED SEGMENTS (LOAD ERR 100)
SEGMAP TOO BIG (LOAD ERR 101)
UNABLE TO EXPAND SEGMAP (LOAD ERR 102)
UNABLE MANY DYNAMIC LOADS ON PROCEDURE (LOAD ERR 103)

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:

J
FROM/{S}*num,username.acctname/message*

The parameters have the following meanings:

<i>num</i>	The job/session number.
<i>username</i>	The names of the transmitting job/session and user, and
<i>acctname</i>	the name of the account under which it is running.
<i>message</i>	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 frequently used 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 Operator, that message appears in one of two formats, depending on its degree of urgency. Urgent messages (Operator Warnings) which pre-empt any form of input/output being conducted on the standard list device, appear in the same format as user messages:

OPERATOR WARNING/*message*

where *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/S3,OPERATOR.SYS/*message*

Once again *message* is the message text.

System Messages

Miscellaneous conditions that terminate or otherwise affect your job/session are reported through system messages. These messages may appear, asynchronously, during the course of running a job/session on the standard list device. Table A-10 lists the system messages and their meanings.

Table A-11. System Messages

CAN'T INITIATE NEW SESSIONS NOW	
A new session 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.	
NOTE: System Operators can bypass rejections due to 2 and 3 by supplying ;HIPRI on the :HELLO and :JOB commands.	
<i>SESSION</i> *{JOB }	ABORTED BY SYSTEM MANAGEMENT*
The job/session has been aborted by the System Operator through the appropriate command. An immediate logoff takes place.	
<i>SESSION</i> *{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 logoff 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	
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.	

File Information Display

In addition to Command Interpreter and run-time (abort) error messages, certain file input/output errors result in the output of a file information display if the caller of the file system intrinsic subsequently calls PRINTFILEINFO. For files not yet opened, or for which the FOPEN intrinsic failed, this display appears as in the example below.

```
+--F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
!  FILE NUMBER 5      IS UNDEFINED.                ! (Line 1)
!  ERROR NUMBER: 2    RESIDUE: 0      (WORDS) ! (Line 2)
!  BLOCK NUMBER: 0    NUMREC: 0      ! (Line 3)
+-----+
```

In this display, the lines indicated show the following information:

Line	Content
1	A warning that there is no corresponding file open.
2	"ERROR NUMBER" indicates the last FOPEN error for the calling program. "RESIDUE" is the number of words not transferred in an input/output request; since no such request applies in this case, this is zero.
3	The "BLOCK NUMBER" and "NUMREC" fields will always be zero in this short form.

For files that were open when a CCG (end-of-file error) or CCL (irrecoverable file error) was returned, the file information display appears as shown in this example:

```
+--F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
!  FILE NAME IS IN.VOLLMER.CLIFTON                ! (Line 1)
!  FOPTIONS: NEW,ASCII,FORMAL,F,NOCCTL,FEQ,      ! (Line 2)
!  NOLABEL                                         ! (Line 3)
!  AOPTIONS: INPUT,NOMR,NOLOCK,DEF,BUF,NOMULTI,  ! (Line 4)
!  WAIT,NOCOPY                                    ! (Line 5)
!  DEVICE TYPE: 0    DEVICE SUBTYPE: 9           ! (Line 6)
!  LDEV: 2          DRT: 4          UNIT: 1       ! (Line 7)
!  RECORD SIZE: 256  BLOCK SIZE: 256  (BYTES) ! (Line 8)
!  EXTENT SIZE: 128  MAX EXTENTS: 8             ! (Line 9)
!  RECPTR: 0        RECLIMIT: 1023             ! (Line 10)
!  LOGCOUNT: 0     PHYSCOUNT: 0               ! (Line 11)
!  EOF AT: 0        LABEL ADDR: %00201327630    ! (Line 12)
!  FILE CODE: 0     ID IS JOE      ULABELS: 0    ! (Line 13)
!  PHYSICAL STATUS: 10000000000000001           ! (Line 14)
!  NUMBER WRITERS: 0  NUMBER READERS: 1         ! (Line 15)
!  ERROR NUMBER: 0    RESIDUE: 0               ! (Line 16)
!  BLOCK NUMBER: 0    NUMREC: 1               ! (Line 17)
+-----+
```

The lines indicated show the following information:

Line	Content
1	The filename (IN.VOLLMER.CLIFTON).
2,3	The <i>foptions</i> in effect, including: Domain: NEW - A new file. SYS - System file domain. JOB - Job temporary file domain. File Type: ASCII BINARY Default file Designator: FORMAL - Actual file designator is the same as the formal file designator. \$STDIN \$STDLIST \$STDINX \$NEWPASS \$OLDPASS \$NULL Record Format: F - Fixed length. V - Variable length. U - Undefined length. ? - Unknown format. Carriage Control: NOCCTL - None. CCTL - Carriage control character expected. File Equation Option: FEQ - :FILE allowed. DEQ - :FILE not allowed. Labeled Tape Option: NOLABEL - Not a labeled tape. LABEL - Labeled tape.

4,5	The <i>aoptions</i> in effect, including: Access Type: INPUT - Read access. OUTPUT - Write access. OUTKEEP - Write-only access, without deleting. APPEND - Append access. IN/OUT - Input and output access. UPDATE - Update access. Multi-record Option: NOMR - Single record access. MR - Multirecord access. Dynamic Locking Option: NOLOCK - No locking permitted. LOCK - Locking permitted. Exclusive Access Option: DEF - Default specification. EXC - Exclusive access allowed. SEA - Semi-exclusive access. SHR - Sharable file. Buffering: BUF - Automatic Buffering. NOBUF - Inhibit buffering. Multi-access Option: NOMULTI - Multi-access not allowed. MULTI - Intra-job multi-access allowed. GMULTI - Inter-job multi-access allowed (any job/session process tree). Wait Option: WAIT - I/O with wait. NOWAIT - I/O without wait. Copy Option: NOCOPY - No special treatment of MSG, CIR files. COPY - Treat MSG, CIR files as regular variable length files.
6,7	The device type, device subtype, Logical Device Number (LDEV), Device Reference Table (DRT), and unit of the device on which the file resides. If the file is a spoolfile, the <i>ldev</i> will be "virtual" rather than a physical device number. (Refer to <i>ldev</i> under FGETINFO .)
8	The record and block size of the offending record, in bytes or words as noted.
9	The extent size of the current extent and the maximum number of extents allowed this file.
10	The current record pointer and limit on number of records in the file.

- 11 The present count of logical and physical records.
- 12 The locations of the current end-of-file and header label of the file.
- 13 The file code, name of the file's creator, and number of user-created labels
- 14 The physical (hardware) status of the device on which the file resides.
- 15 "NUMBER WRITERS" is the number of **FOPEN** calls of the file with some type of write access to the file. "NUMBER READERS" is the number of **FOPEN** calls to the file with read access to the file. This is only for message files, in all other files it will not appear.
- 16 The error number and residue; same as for the abbreviated file information display format.
- 17 The block number and number of records (**NUMREC**) for the file.

Section Divider

B. Device Characteristics



Device Characteristics

MPE intrinsics can be used to alter certain aspects of device operation. Before any of these intrinsics can be issued against a device, the devicefile must be opened with the **FOPEN** intrinsic (refer to the **FOPEN** discussion in Section 2).

Characteristics

With the **FCONTROL** intrinsic, you can:

- Change terminal speed.
- Change input echo facility.
- Enable and disable the system **BREAK** function.
- Enable and disable subsystem **BREAK** requests.
- Enable and disable parity checking.
- Enable and disable tape-mode option.
- Enable and disable the terminal timer.
- Read the result from the terminal input timer.
- Define line-termination characters for terminal input.

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. A carriage control code of %320 will append data to the current contents of the line printer buffer, whether prespace or postspace is selected.

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 undefined-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. An end of tape indication is returned whenever a write operation completes with the tape positioned beyond the EOT marker.

Line Printer and Terminal Carriage Control Codes

Line printer and terminal carriage control codes are shown in Table 2-5 (refer to Section 2, **FWRITE**). All of the Carriage Control Codes shown in Table 2-5 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. The device must be nonspooled. When the file is opened with **CCTL**, the Carriage Control Codes may be used via **FWRITE** either as the value of the *control* parameter, or, when *control* is specified as 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. If the word count is less than 3 or the byte count is less than 6, Type B reads are converted to Type A reads.
E	Hardware-sensed end-of-file.

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.

Any subsequent requests to the device after an end-of-file condition is detected are rejected with an end-of-file indication, except as described in the next paragraph.

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

Refer to the *MPE File System Reference Manual* (30000-90236) for detailed information on terminals and terminal characteristics.

Using the FCARD Intrinsic With an HP 7260A Optical Mark Reader

The FCARD intrinsic enables you to programmatically control the operation of the HP 7960A Optical Mark Reader (OMR). This is achieved through passing a parameter value (*recode*), corresponding to the function of FCARD desired, from a program to FCARD. FCARD returns parameter values to the calling program which indicate the success or the cause of failure of execution, the status of the HP 7260A, the file number of the HP 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 B-1, Figure B-2 and Figure B-3 performs the following:

- Opens the HP 7260A/terminal file.
- Displays Operator instructions.
- Temporarily suspends program operation until the HP 7260A READY switch is pressed.
- Reads ten cards in the ASCII reading format.
- Displays the number of columns read from each card.
- Examines STATUS for empty input hopper status.
- Examines output RECODE values for each request.
- Closes the HP 7260A terminal file.

Under the label **OPENFILE**, the program requests that an HP 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 "CANNOT 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** to 4 and calling **FCARD** as illustrated.

The program, upon regaining process control, checks for unsuccessful execution of the request (**RECODE**<>0). If the execution was unsuccessful, the program goes to the label **FINIS** and terminates. The program could also have branched to an instruction set to correct or display an error at

this point. 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** 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 the procedure **DISPCOUNT**, which displays the number of columns read from the previous card. After the **DISPCOUNT** procedure is completed, the program goes to the label **CLOSE'F**.

Under the label **CLOSE'F**, the program requests that the HP 7260A be put in the NOT READY state and that the HP 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 in Figure B-3.

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:

Bits	0	7	8	15
Word 1	1st column data	2nd column data		
Word 2	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 in the following program example:

BUFFER WORD	-	-	-	-	12	11	0	1	2	3	4	5	6	7	8	9	COLUMN ROW NO.
	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.

```

$CONTROL USLINIT
BEGIN
INTEGER ARRAY BUFADR(0:99);
BYTE ARRAY TOO(0:72);
POINTER HERE;
INTEGER RECODE,A,I;
INTRINSIC QUIT,PRINT;
INTEGER COUNT,FILENUM,STATUS;
INTRINSIC PRINT'FILE'INFO;

PROCEDURE DISP'COUNT(COUNT);
INTEGER COUNT;

BEGIN
    ARRAY OUT(0:11);
    BYTE ARRAY ROUT(*)=OUT;
    INTRINSIC PRINT,ASCII;
    INTEGER A1,A2;

    MOVE ROUT:="NO. OF COLUMNS READ= ";
    A1:=ASCII(COUNT,10,ROUT(21));
    A2:= -21-A1;
    PRINT(OUT,A2,%401);

END;

PROCEDURE FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
INTEGER ARRAY BUFADR;
INTEGER RECODE,FILENUM,COUNT,STATUS;
OPTION EXTERNAL;

@HERE:=@TOO & LSR(1);
OPENFILE:
    <<GET FILE NUMBER FOR LOGICAL DEV EQUAL TO THE TERMINAL>>
    RECODE:=0;
    FCARD(RECODE,FILENUM,BUFADR,COUNT,STATUS);
    IF RECODE =0 THEN GO DISPINST;
    MOVE TOO:="CANNOT OPEN FILE - PROGRAM WILL TERMINATE";
    PRINT (HERE,-40,0);
    GO FINIS;

```

Figure B-1. FCARD Intrinsic Example (1 of 3)

```

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);

RDYWAIT:
  MOVE TOO:="NOW, PRESS THE READY SWITCH.";
  PRINT(HERE,-28,0);
  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 TOO:="INPUT HOPPER EMPTY OR OUTPUT HOPPER FULL";
  PRINT(HERE,-40,0);
  MOVE TOO:="CORRECT HOPPER CONDITION AND PRESS READY";
  PRINT(HERE,-40,0);
  IF RECODE <> 0 THEN GO FINIS ELSE
  GO READ';

```



Figure B-2. FCARD Intrinsic Example (2 of 3)

```

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 TOO:="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
    MOVE TOO:="FREAD OR FWRITE ERROR-PROGRAM WILL ABORT";
    PRINT(HERE,-40,0);
    QUIT(RECODE);  END;

IF RECODE =6 THEN BEGIN
  MOVE TOO:=":EOJ, :EOD, :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 B-3. FCARD Intrinsic Example (3 of 3)

Section Divider

Index



Index

A

- Abort calling process
 - QUIT, 2-283
- Abort entire user process structure
 - QUITPROG, 2-284
- Aborting
 - A Process, 5-28
 - A Program, 5-28
- Abort stack analysis facility, disable
 - RESETDUMP, 2-292
- Accept another subsystem break signal
 - RESETCONTROL, 2-291
- Access
 - Non-Sharable Device, 4-2
- Access, close user logging facility
 - CLOSELOG, 2-28
- Access file, predefine characteristics
 - FOPEN, 2-141
- Access file, terminate
 - FCLOSE, 2-78
- Access information returned, file
 - FGETINFO, 2-111
- Access job and session information
 - JOBINFO, 2-237
- Access message catalogs
 - GENMESSAGE, 2-202
- Access mode, control of files/devices
 - FSETMODE, 2-183
- Access-Mode Options
 - Declaring, 4-47
- Access to process information
 - PROCINFO, 2-273
- Access to User Logging facility
 - OPENLOG, 2-263
- Acquiring
 - Global RINs, 3-44
 - RINs, Local, 3-49
- Activating
 - Processes, 3-29
- Activation call for process, return source
 - GETORIGIN, 2-214
- Active Process Substates, 3-28
- Allocation to job, free local RINs
 - FREELOCRIN, 2-178
- ALTDSEG
 - Alter size of extra data segment, 2-15
- Intrinsic, 2-15
- Alter DB to Z area of stack
 - ZSIZE, 2-325
- Alter size of extra data segment
 - ALTDSEG, 2-15
- Arithmetic trap, enable/disable
 - ARITRAP, 2-17
- Arithmetic trap handling procedure, arm/disarm
 - XARITRAP, 2-318
- Arithmetic Traps, 5-39
- ARITRAP
 - Collectively enables/disables arithmetic traps, 2-17
 - Intrinsic, 2-17
- Arm/disarm arithmetic trap handling procedure
 - XARITRAP, 2-318
- Arm/disarm software library trap handling procedure
 - XLIBTRAP, 2-323
- Arm/disarm subsystem break (Control-Y) trap handling procedure
 - XCONTRAP, 2-321
- Arm/disarm system trap handling procedure
 - XSYSTRAP, 2-324
- Arm system debugger from process abort
 - SETDUMP, 2-296
- Arrays
 - Searching, 5-3
- Array, search for specified entry/name
 - SEARCH, 2-293
- ASCII
 - Convert binary to numeric ASCII string, 2-19
 - Intrinsic, 2-19
- ASCII Numeric String
 - Converting to Binary Coded Value, 5-19
- ASCII Reading Formats, B-4
- ASCII string conversion to EBCDIC
 - CTranslate, 2-44
- ASCII string, convert from binary number
 - DASCII, 2-46
- ASCII string, convert to binary
 - BINARY, 2-23
- ASCII string number, convert to binary
 - DBINARY, 2-49
- ASCII string, read \$STDIN into array
 - READ, 2-285

- ASCII string, read \$STDINX into array
 READX, 2-287
- Assign system-defined files and create process
 CREATEPROCESS, 2-39
- Assign value of JCW in JCW table
 PUTJCW, 2-281
- Avoiding Deadlocks, 3-39

B

- BEGINLOG
 - Intrinsic, 2-21
 - Mark beginning of user logging transaction, 2-21
- BINARY
 - Convert number from ASCII string to binary, 2-23
 - Intrinsic, 2-23
- Binary Code
 - Converting Numbers to ASCII Strings, 5-15
- Binary conversion, to numeric ASCII string
 ASCII, 2-19
- binary, convert from ASCII string number
 DBINARY, 2-49
- binary number, convert to ASCII string
 DASCII, 2-46
- Break mode, place session in
 CAUSEBREAK, 2-25

C

- CALENDAR
 - Intrinsic, 2-24
 - Return calendar date, 2-24
- Calendar Date
 - Formatting, 5-56
- Calendar date, format
 FMTCALENDAR, 2-138
 FMTDATE, 2-140
- Calling Intrinsics, 1-1
- Calling Intrinsics From Languages, 1-5
- Calling Process
 - Suspending, 5-26
- Calling process, abort
 QUIT, 2-283
- Calling process, delete
 TERMINATE, 2-308
- Calling process, suspend for specified time
 PAUSE, 2-266
- Capabilities
 - Data Segment Management, 3-10
 - Optional, 1-9
 - Privileged Mode, 3-1
 - Process Handling, 3-26
 - Required, 2-5
- Carriage Control Codes, B-2
- Catalog

- Message, 5-60
- CAUSEBREAK
 - Intrinsic, 2-25
 - Place a session in break mode, 2-25
- Cause return from interrupt procedure
 FINTEXT, 2-126
- Change length of USL file
 EXPANDUSLF, 2-63
- Change priority of process
 GETPRIORITY, 2-215
- Changing
 - DL to DB Area Size, 5-32
 - Size of Extra Data Segment, 3-25
 - Z to DB Area Size, 5-37
- Changing Stack Sizes, 5-31
- Character string, print on listing device
 PRINT, 2-267
- Character string, print on system console
 PRINTOP, 2-270
- Character string, print on system console/reply
 PRINTOPREPLY, 2-271
- Child of calling process
 CREATE, 2-33
- Child process, delete
 KILL, 2-244
- Child process, return PIN
 GETPROCID, 2-218
- Child process, return status
 GETPROCINFO, 2-219
- CLEANUSL
 - Delete inactive entries from USL file, 2-26
 - Intrinsic, 2-26
- CLOCK
 - Intrinsic, 2-27
 - Return system time, 2-27
- Close access to user logging facility
 CLOSELOG, 2-28
- Close a file
 FCLOSE, 2-78
- CLOSELOG
 - Close access to user logging facility, 2-28
 - Intrinsic, 2-28
- Closing
 - Files, 4-35
 - New File as Permanent File, 4-38
 - New File as Temporary File, 4-36
- Collectively enables/disables arithmetic traps
 ARITRAP, 2-17
- Column Image Reading Formats, B-4
- COMMAND
 - Execute command programmatically, 2-31
 - Intrinsic, 2-31
- Command image, parse user defined
 MYCOMMAND, 2-259
- Command Parameters

- Formatting, 5-4
- Command, programmatic execution
 - COMMAND, 2-31
- Commercial Instruction Traps, 5-41
- Communication
 - Interprocess, 3-36
- Condition Codes, 1-6, 2-5
- Contract/expand stack
 - DLSIZE, 2-51
- Control access mode of files/devices
 - FSETMODE, 2-183
- Control operation, on a file or device
 - FCONTROL, 2-82
- Control operations for a printer
 - FDEVICECONTROL, 2-90
- Control operations for a spooled devicefile
 - FDEVICECONTROL, 2-90
- Control-Y trap handling procedure, arm/disarm
 - XCONTRAP, 2-321
- CONTROL-Y Traps, 5-49
- Conversion, binary to numeric ASCII string
 - ASCII, 2-19
- Convert ASCII string number to binary
 - DBINARY, 2-49
- Convert binary number to ASCII string
 - DASCII, 2-46
- Convert binary to numeric ASCII string
 - ASCII, 2-19
- Converting
 - Numbers from ASCII Numeric String to Binary Value, 5-19
 - Numbers from Binary Code to ASCII String, 5-15
- Convert number from ASCII string to binary
 - BINARY, 2-23
- Convert string from ASCII to EBCDIC
 - CTranslate, 2-44
- Convert string from EBCDIC to ASCII
 - CTranslate, 2-44
- Convert string from EBCDIK to JIS
 - CTranslate, 2-44
- Convert string from JIS to EBCDIK
 - CTranslate, 2-44
- Copy data area to extra data segment
 - DMOVOUT, 2-55
- Copy data from extra data segment
 - DMOVIN, 2-53
- CPU time, return for process
 - PROCTIME, 2-278
- CREATE
 - Create a process, 2-33
 - Intrinsic, 2-33
- Create a process
 - CREATE, 2-33
- Create extra data segment

- GETDSEG, 2-207
- CREATEPROCESS
 - Create process and assign system-defined files, 2-39
 - Intrinsic, 2-39
- Create process and assign system-defined files
 - CREATEPROCESS, 2-39
- Create Volumes Capability, 1-10
- Creating
 - Extra Data Segment, 3-10
 - Processes, 3-29
- Creating an Extra Data Segment, 3-11
- CTranslate
 - Convert string from ASCII to EBCDIC, 2-44
 - Convert string from EBCDIC to ASCII, 2-44
 - Convert string from EBCDIK to JIS, 2-44
 - Convert string from JIS to EBCDIK, 2-44
 - Intrinsic, 2-44
- CTranslate Intrinsic
 - Translating Characters With, 5-22
- Current date and time returned
 - DATELINE, 2-48

D

- DASCII
 - Convert binary number to ASCII string, 2-46
 - Intrinsic, 2-46
- Data area, copy to extra data segment
 - DMOVOUT, 2-55
- Data segment, alter size
 - ALTDSEG, 2-15
- Data segment, copy data from
 - DMOVIN, 2-53
- Data segment, create extra
 - GETDSEG, 2-207
- Data Segment Management Capability, 1-9, 3-10
- Data segment, release extra
 - FREEDSEG, 2-176
- Data stack, read from direct access disk file
 - FREADDIR, 2-171
- Date
 - Calendar, Obtaining, 5-56
- Date and time, return
 - DATELINE, 2-48
- Date, format
 - FMTDATE, 2-140
- Date, format calendar
 - FMTCALENDAR, 2-138
- DATELINE
 - Intrinsic, 2-48
 - Return current date and time, 2-48
- Date, return calendar
 - CALENDAR, 2-24
- Day, format time
 - FMTCLOCK, 2-139

DBINARY

- Convert ASCII string number to binary, 2-49

- Intrinsic, 2-49

DB Pointer

- Moving, 3-5

Deadlocks, 3-39

DEBUG

- Intrinsic, 2-50

- Invoke Debug facility, 2-50

Debug facility, invoke

- DEBUG, 2-50

Declarations

- Intrinsic, 1-2

- Procedure, 1-2

Declaring

- Access-Mode Options, 4-47

Decrease length of USL file

- EXPANDUSLF, 2-63

Define file and establish access

- FOPEN, 2-141

Delete calling process

- TERMINATE, 2-308

Delete child process

- KILL, 2-244

Delete inactive entries from USL file

- CLEANUSL, 2-26

Deleting

- Extra Data Segment, 3-25

- Processes, 3-34

Determine if file pair is inactive/duplicative

- FRELATE, 2-179

Determine process PIN that locked local RIN

- LOCRINOWNER, 2-251

Determining

- Father Process, 3-40

- User Access Mode/Attributes, 5-12

Determining Son Process, 3-40

Determining Source of Process Activation, 3-40

Device Characteristics

- Line Printer, B-1

- Magnetic Tape, B-1

- Terminals, B-3

Device, perform control operation

- FCONTROL, 2-82

Devices/files, control access mode

- FSETMODE, 2-183

Direct access disk file, read logical record

- FREADDIR, 2-171

Disable abort stack analysis facility

- RESETDUMP, 2-292

Disable arithmetic trap

- ARITRAP, 2-17

Disable/enable software interrupts

- FINTSTATE, 2-127

Disarm/arm arithmetic trap handling procedure

XARITRAP, 2-318

Disarm/arm software library trap handling procedure

XLIBTRAP, 2-323

Disarm/arm subsystem break (Control-Y) trap handling procedure

XCONTRAP, 2-321

Disarm/arm system trap handling procedure

XSYSTRAP, 2-324

Disk file label information returned

- FLABELINFO, 2-129

disk file, move record pointer

- FSPACE, 2-186

disk file record, move to buffer

- FREADSEEK, 2-175

disk file, remove

- FRENAME, 2-181

Disk Files

- Opening, 4-25

disk file, set logical record pointer

- FPOINT, 2-165

disk file, update logical record

- FUPDATE, 2-189

Display File Information, A-11

DLSIZE

- Expand/Contract mode stack, 2-51

- Intrinsic, 2-51

DL to DB Area Size

- Changing, 5-32

DMOVIN

- Copy data from extra data segment, 2-53

- Intrinsic, 2-53

DMOVOUT

- Copy stack to extra data segment, 2-55

- Intrinsic, 2-55

Domains

- File, 4-2

Duplicative/inactive, determine file pair

- FRELATE, 2-179

Dynamically enter privileged mode

- GETPRIVMODE, 2-217

Dynamically return program to nonprivileged mode

- GETUSERMODE, 2-221

Dynamically unloads segmented library (SL) procedure

- UNLOADPROC, 2-310

Dynamically unlock file

- FUNLOCK, 2-188

Dynamic Loading, 5-2

Dynamic Loading/Unloading, Library Procedures, 5-2

Dynamic Unloading, 5-3

E

- EBCDIC string conversion to ASCII
 - CTranslate, 2-44
- EBCDIK string conversion to JIS
 - CTranslate, 2-44
- Enable arithmetic trap
 - ARITRAP, 2-17
- Enable/disable software interrupts
 - FINTSTATE, 2-127
- Enabling/Disabling Traps, 5-38
- ENDLOG
 - Intrinsic, 2-60
 - Mark end of user logging transaction, 2-60
- End-of-File Indication, B-2
- Entering
 - Nonprivileged Mode, 3-5
 - Privileged Mode, 3-3
- Enter privileged mode, dynamically
 - GETPRIVMODE, 2-217
- Entries, clean inactive from USL file
 - CLEANUSL, 2-26
- Entry, search array for specified
 - SEARCH, 2-293
- Error information for file I/O errors, return
 - FCHECK, 2-71
- Error Messages, Type of, A-1
- Errors
 - Intrinsic Call, 1-6
- Exclusive use, lock file
 - FLOCK, 2-134
- Execute command programmatically
 - COMMAND, 2-31
- Execute full stack trace
 - STACKDUMP, 2-298
- Executing
 - MPE Commands Programmatically, 5-10
- Expand/contract stack
 - DLSIZE, 2-51
- EXPANDUSLF
 - Change length of USL file, 2-63
 - Intrinsic, 2-63
- Extended Precision Floating Point Traps, 5-40
- Extra Data Segment
 - Changing Size Of, 3-25
 - Creating, 3-10
 - Deleting, 3-25
 - Transferring Data From Stack, 3-25
 - Transferring Data to Stack, 3-25
- Extra data segment, copy data from
 - DMOVIN, 2-53
- Extra data segment, copy from data area
 - DMOVOUT, 2-55
- Extra data segment, create
 - GETDSEG, 2-207

- Extra data segment, release
 - FREEDSEG, 2-176

F

- Facility access, close user logging
 - CLOSELOG, 2-28
- FATHER
 - Intrinsic, 2-65
 - Request PIN of parent process, 2-65
- Father Process
 - Determining, 3-40
- FCARD Intrinsic
 - Using to Operate Optical Mark Reader, B-3
- FCHECK
 - Intrinsic, 2-71
 - Return error information for file I/O errors, 2-71
- FCHECK error number returned
 - FERRMSG, 2-98
- FCLOSE
 - Close a file, 2-78
 - Intrinsic, 2-78
- FCONTROL
 - Intrinsic, 2-82
 - Perform control operation on a file or device, 2-82
- FDEVICECONTROL
 - Control operations for printer or spooled device-file, 2-90
 - Intrinsic, 2-90
- FERRMSG
 - Intrinsic, 2-98
 - Returns message corresponding to FCHECK error number, 2-98
- FERRMSG Intrinsic
 - Using, 4-41
- FFILEINFO
 - Intrinsic, 2-99
 - Returns information about a file, 2-99
- FGETINFO
 - Intrinsic, 2-111
 - Return access and status information for file, 2-111
- File access
 - FOPEN, 2-141
- File access, terminate
 - FCLOSE, 2-78
- File designator, parse and validate
 - FPARSE, 2-162
- File Device Relationships, 4-2
- File display, print on list device
 - PRINTFILEINFO, 2-269
- File Domains, 4-2
- File Information Display, A-11
- File information returned

- FFILEINFO, 2-99
- File information returned, access
 - FGETINFO, 2-111
- File information returned, status
 - FGETINFO, 2-111
- File I/O errors, error information returned
 - FCHECK, 2-71
- File label information, disk file returned
 - FLABELINFO, 2-129
- File label, read
 - FREADLABEL, 2-173
- File, lock for exclusive use
 - FLOCK, 2-134
- File pair, determine if inactive/duplicative
 - FRELATE, 2-179
- File, perform control operation
 - FCONTROL, 2-82
- File, read logical record to stack
 - FREAD, 2-166
- Files
 - Accessing Internal Operations For, 4-7
 - Closing, 4-35
 - Closing New as Temporary, 4-36
 - Closing New File as Permanent File, 4-38
 - Device Relationships, 4-2
 - How to Use, 4-6
 - Non-Sharable Devices, 4-5
 - Opening, 4-4
 - Opening Device Other Than Disk, 4-29
 - Opening New Disk Files, 4-23
- Files/devices, control access mode
 - FSETMODE, 2-183
- File System Error Check Procedure
 - Writing, 4-40
- File, unlock dynamically
 - FUNLOCK, 2-188
- FINDJCW
 - Intrinsic, 2-120
- FINDJCW Search JCW table and return value, 2-120
- FINTEXT
 - Cause return from interrupt procedure, 2-126
 - Intrinsic, 2-126
- FINTSTATE
 - Enable/disable software interrupts, 2-127
 - Intrinsic, 2-127
- FLABELINFO
 - Intrinsic, 2-129
 - Return information from file label, disk file, 2-129
- FLOCK
 - Intrinsic, 2-134
 - Lock file for exclusive use, 2-134
- Flush contents, User Logging memory buffer
 - FLUSHLOG, 2-136
- FLUSHLOG
 - Flush contents, User Logging memory buffer, 2-136
 - Intrinsic, 2-136
- FMTCALENDAR
 - Format calendar date, 2-138
 - Intrinsic, 2-138
- FMTCLOCK
 - Format time of day, 2-139
 - Intrinsic, 2-139
- FMTDATE
 - Format calendar date and time of day, 2-140
 - Intrinsic, 2-140
- FOPEN
 - Define file and establish access, 2-141
 - Intrinsic, 2-141
- Format calendar date
 - FMTCALENDAR, 2-138
- Format calendar date and time of day
 - FMTDATE, 2-140
- Format time of day
 - FMTCLOCK, 2-139
- Formatting Calendar Date & Time Information, 5-56
- Formatting Command Parameters, 5-4
- FPARSE
 - Intrinsic, 2-162
 - Parse and validate file designator, 2-162
- FPOINT
 - Intrinsic, 2-165
 - Set logical record pointer for disk file, 2-165
- FREAD
 - Intrinsic, 2-166
 - Read logical record from file to stack, 2-166
- FREADBACKWARD
 - Intrinsic, 2-169
 - Read reverse capability, 2-169
- FREADDIR
 - Intrinsic, 2-171
 - Read logical record from direct access disk file, 2-171
- FREADLABEL
 - Intrinsic, 2-173
 - Read user file label, 2-173
- FREADSEEK
 - Intrinsic, 2-175
 - Move record from disk file to buffer, 2-175
- FREEDSEG
 - Intrinsic, 2-176
 - Release extra data segment, 2-176
- Freeing Local RINs, 3-51
- Free local RINs from allocation to job
 - FREELOCRIN, 2-178
- FREELOCRIN
 - Free local RINs from allocation to job, 2-178

- Intrinsic, 2-178
- FRELATE**
 - Determine if file pair is inactive/duplicative, 2-179
 - Intrinsic, 2-179
- FRENAME**
 - Intrinsic, 2-181
 - Remove disk file, 2-181
- FSETMODE**
 - Control access modes of files/devices, 2-183
 - Intrinsic, 2-183
- FSPACE**
 - Intrinsic, 2-186
 - Move record pointer on tape/disk file, 2-186
- Full stack trace, execute
 - STACKDUMP**, 2-298
- Functional Return, 2-3
- FUNLOCK**
 - Dynamically unlock file, 2-188
 - Intrinsic, 2-188
- FUPDATE**
 - Intrinsic, 2-189
 - Update logical record in disk file, 2-189
- FWRITE**
 - Intrinsic, 2-191
 - Write logical/physical record from stack to file, 2-191
- FWRITEDIR**
 - Intrinsic, 2-199
 - Write logical record from stack to disk file, 2-199
- FWRITELABEL**
 - Intrinsic, 2-201
 - Write user file label, 2-201

G

- GENMESSAGE**
 - Access message catalogs, 2-202
 - Intrinsic, 2-202
- GETDSEG**
 - Create extra data segment, 2-207
 - Intrinsic, 2-207
- GETINFO**
 - Intrinsic, 2-210
 - Return user-supplied information, 2-210
- GETJCW**
 - Intrinsic, 2-212
 - Return system defined JCW value, 2-212
- GETORIGIN**
 - Intrinsic, 2-214
 - Return source of activation call for process, 2-214
- GETPRIORITY**
 - Change priority of process, 2-215
 - Intrinsic, 2-215

- GETPRIVMODE**
 - Dynamically enter privileged mode, 2-217
 - Intrinsic, 2-217
- GETPROCID**
 - Intrinsic, 2-218
 - Return PIN of child process, 2-218
- GETPROCINFO**
 - Intrinsic, 2-219
 - Return status for parent/child process, 2-219
- GETUSERMODE**
 - Dynamically return program to nonprivileged mode, 2-221
 - Intrinsic, 2-221
- Global RINs
 - Acquiring, 3-44
 - Inter-Job Level, 3-43
 - Locking/Unlocking, 3-45
 - Releasing, 3-44

H

- Hand-Shaking Arrangement, 3-42
- HPACDINFO**
 - Intrinsic, 2-222
- HPACDPUT**
 - Intrinsic, 2-225

I

- Identifying
 - Local RIN Owners, 3-51
- Identifying Job/Session, with **JOBINFO**, 5-14
- Implementing Intrinsic Calls, 1-3
- Inactive/duplicative, determine file pair
 - FRELATE**, 2-179
- Inactive entries, clean from USL file
 - CLEANUSL**, 2-26
- Increase length of USL file
 - EXPANDUSLF**, 2-63
- Information about open User Logging file
 - LOGINFO**, 2-252
 - LOGSTATUS**, 2-255
- Information, return user-supplied
 - GETINFO**, 2-210
- Initialize USL file to empty state
 - INITUSLF**, 2-232
- Initiate completion for I/O request
 - IODONTWAIT**, 2-233
 - IOWAIT**, 2-235
- Initiate session on specified terminal
 - STARTSESS**, 2-300
- INITUSLF**
 - Initialize USL file to empty state, 2-232
 - Intrinsic, 2-232
- Inter-Job Level Global RINs, 3-43
- Internal Operations For File Accessing, 4-7

Interprocess Communication, 3-28, 3-36

Interprocess Local Level RINs, 3-49

Interrupt procedure, cause return from

 FINTEXT, 2-126

Interrupts, enable/disable software

 FINTSTATE, 2-127

Intrinsic

 ALTDSEG, 2-15

 ARITRAP, 2-17

 ASCII, 2-19

 BEGINLOG, 2-21

 BINARY, 2-23

 CALENDAR, 2-24

 CAUSEBREAK, 2-25

 CLEANUSL, 2-26

 CLOCK, 2-27

 CLOSELOG, 2-28

 COMMAND, 2-31

 CREATE, 2-33

 CREATEPROCESS, 2-39

 CTRANSLATE, 2-44

 DASCH, 2-46

 DATELINE, 2-48

 DBINARY, 2-49

 DEBUG, 2-50

 DLsize, 2-51

 DMOVIN, 2-53

 DMOVOUT, 2-55

 ENDLOG, 2-60

 EXPANDUSLF, 2-63

 FATHER, 2-65

 FCHECK, 2-71

 FCLOSE, 2-78

 FCONTROL, 2-82

 FDEVICECONTROL, 2-90

 FERRMSG, 2-98

 FFILEINFO, 2-99

 FGETINFO, 2-111

 FINDJCW, 2-120

 FINTEXT, 2-126

 FINTSTATE, 2-127

 FLABELINFO, 2-129

 FLOCK, 2-134

 FLUSHLOG, 2-136

 FMTCALENDAR, 2-138

 FMTCLOCK, 2-139

 FMTDATE, 2-140

 FOPEN, 2-141

 FPARSE, 2-162

 FPOINT, 2-165

 FREAD, 2-166

 FREADBACKWARD, 2-169

 FREADDIR, 2-171

 FREADLABEL, 2-173

 FREADSEEK, 2-175

 FREEDSEG, 2-176

 FREELOCRIN, 2-178

 FRELATE, 2-179

 FRENAME, 2-181

 FSETMODE, 2-183

 FSPACE, 2-186

 FUNLOCK, 2-188

 FUPDATE, 2-189

 FWRITE, 2-191

 FWRITEDIR, 2-199

 FWRITELABEL, 2-201

 GENMESSAGE, 2-202

 GETDSEG, 2-207

 GETINFO, 2-210

 GETJCW, 2-212

 GETORIGIN, 2-214

 GETPRIORITY, 2-215

 GETPRIVMODE, 2-217

 GETPROCID, 2-218

 GETPROCINFO, 2-219

 GETUSERMODE, 2-221

 HPACDINFO, 2-222

 HPACDPUT, 2-225

 INITUSLF, 2-232

 IODONTWAIT, 2-233

 IOWAIT, 2-235

 JOBINFO, 2-237

 KILL, 2-244

 LOADPROC, 2-245

 LOCKGLORIN, 2-247

 LOCKLOCRIN, 2-249

 LOCRINOWNER, 2-251

 LOGINFO, 2-252

 LOGSTATUS, 2-255

 MAIL, 2-257

 MYCOMMAND, 2-259

 OPENLOG, 2-263

 PAUSE, 2-266

 PRINT, 2-267

 PRINTFILEINFO, 2-269

 PRINTOP, 2-270

 PRINTOPREPLY, 2-271

 PROCINFO, 2-273

 PROCTIME, 2-278

 PTAPE, 2-278

 PUTJCW, 2-281

 QUIT, 2-283

 QUITPROG, 2-284

 READ, 2-285

 READX, 2-287

 RECEIVEMAIL, 2-289

 RESETCONTROL, 2-291

 RESETDUMP, 2-292

 SEARCH, 2-293

 SENDMAIL, 2-294

SETDUMP, 2-296
 SETJCW, 2-297
 STACKDUMP, 2-298
 STARTSESS, 2-300
 SUSPEND, 2-305
 SWITCHDB, 2-306
 TERMINATE, 2-308
 TIMER, 2-309
 UNLOADPROC, 2-310
 UNLOCKGLORIN, 2-311
 UNLOCKLOCRIN, 2-312
 WHO, 2-313
 WRITELOG, 2-316
 XARITRAP, 2-318
 XCONTRAP, 2-321
 XLIBTRAP, 2-323
 XSYSTRAP, 2-324
 ZSIZE, 2-325
 Intrinsic Calls
 Implementing, 1-3
 Intrinsic Declarations, 1-2
 Intrinsic Names, 2-1
 Intrinsic
 Call Errors, 1-6
 Calling, 1-1
 Calling From SPL, 1-2
 Functions, 1-10
 Procedure Declarations, 1-2
 Time and Date, 5-53
 Invoke Debug facility
 DEBUG, 2-50
 IODONTWAIT
 Initiate completion for I/O request, 2-233
 Intrinsic, 2-233
 I/O request, initiate completion for
 IODONTWAIT, 2-233
 IOWAIT, 2-235
 IOWAIT
 Initiate completion for I/O request, 2-235
 Intrinsic, 2-235
 IOWAIT Intrinsic
 Using, 4-42
J
 JCW, set bits in stream
 SETJCW, 2-297
 JCWs (See Job Control Words), 5-57
 JCW table search, return value FINDJCW, 2-120
 JCW value, assign to JCW table
 PUTJCW, 2-281
 JCW value, return system defined
 GETJCW, 2-212
 JIS string conversion to EBCDIK
 CTranslate, 2-44
 Job

Identifying with JOBINFO, 5-14
 Job Control Words
 User-Defined, 5-59
 Job Control Words (JCWs), 5-57
 JOBINFO
 Access job and session information, 2-237
 Intrinsic, 2-237
 Job information, access
 JOBINFO, 2-237
 Job/Session Input Devices
 Reading Input From, 5-23
 Job/Session Input/Output Devices
 Transmitting Programs From, 5-23
 Job/Session List Device
 Writing Output to, 5-25

K
 KILL
 Delete child process, 2-244
 Intrinsic, 2-244

L
 Label, write user file
 FWRITE LABEL, 2-201
 Languages
 Calling Intrinsic From, 1-5
 Library Procedures
 Dynamic Loading/Unloading, 5-2
 Library Trap, 5-45
 Line Printer, B-1
 Line Printer Carriage Control Codes, B-2
 Loading
 Dynamic, 5-2
 LOADPROC
 Intrinsic, 2-245
 Load SL procedure, 2-245
 Load SL procedure
 LOADPROC, 2-245
 Local RINs, free from allocation to job
 FREELOCRIN, 2-178
 Lock file for exclusive use
 FLOCK, 2-134
 Lock global RIN
 LOCKGLORIN, 2-247
 LOCKGLORIN
 Intrinsic, 2-247
 Lock global RIN, 2-247
 Locking
 Global RINs, 3-45
 Local RINs, 3-49
 Lock local RIN
 LOCKLOCRIN, 2-249
 LOCKLOCRIN
 Intrinsic, 2-249



- Lock local RIN, 2-249
- LOCRIOWNER
 - Determine process PIN that locked local RIN, 2-251
 - Intrinsic, 2-251
- Log File Uses, 3-59
- Logging
 - User Procedures, 3-59
- Logging, User, 3-52
- Logical record pointer for disk file, set
 - FPOINT, 2-165
- Logical record, read from current record pointer
 - FREADBACKWARD, 2-169
- Logical record, read from file to stack
 - FREAD, 2-166
- Logical record, update in disk file
 - FUPDATE, 2-189
- Logical record, write from stack to disk file
 - FWRITEDIR, 2-199
- Logical record, write from stack to file
 - FWRITE, 2-191
- LOGINFO
 - Information about open User Logging file, 2-252
 - Intrinsic, 2-252
- LOGSTATUS
 - Information about open User Logging file, 2-255
 - Intrinsic, 2-255

M

- Magnetic Tape, B-1
- Mail
 - Receiving, 3-38
 - Sending, 3-37
- MAIL
 - Intrinsic, 2-257
 - Test mailbox status, 2-257
- Mailbox, 3-36
 - Testing Status Of, 3-36
- Mailbox status, test
 - MAIL, 2-257
- Mail, send to another process
 - SENDMAIL, 2-294
- MAKECAT Program, 5-61
- Mark beginning of user logging transaction
 - BEGINLOG, 2-21
- Mark end of user logging transaction
 - ENDLOG, 2-60
- Memory buffer, flush contents
 - FLUSHLOG, 2-136
- Message Catalog, 5-60
- Message catalogs, access
 - GENMESSAGE, 2-202
- Message Facility, MPE, 5-60
- Message returned, corresponds to FCHECK
 - FERRMSG, 2-98

- Messages
 - Operator, A-10
 - Run-Time, A-1
 - System, A-10
 - System Console, A-1
 - System Failure, A-1
 - System Operator, A-1
 - User, A-9
- Mnemonics, Definitions, 2-1
- Move record from disk file to buffer
 - FREADSEEK, 2-175
- Move record pointer on tape/disk file
 - FSPACE, 2-186
- Moving
 - DB Pointer, 3-5
- MPE Commands
 - Executing Programmatically, 5-10
- MPE Message Facility, 5-60
- Multiple Resource Identification Number Capability, 1-9
- MYCOMMAND
 - Intrinsic, 2-259
 - Parse user defined command image, 2-259

N

- Name, search array for specified
 - SEARCH, 2-293
- Non-privileged Mode
 - Entering, 3-5
- NonPrivileged mode, return program
 - GETUSERMODE, 2-221
- Non-Sharable Device Access, 4-2
- Non-Sharable Devices
 - Files On, 4-5

O

- Obtaining
 - Calendar Date, 5-56
- Obtaining Process Run Time, 5-56
- Obtaining System Timer Information, 5-53
- Obtaining the Current Time, 5-54
- Opening
 - Files, 4-4
 - STDIN, 4-31
 - STDLIST, 4-32
- Opening Files On Devices Other Than Disk, 4-29
- Opening New Disk Files, 4-23
- Opening Old Disk Files, 4-25
- OPENLOG
 - Access to User Logging facility, 2-263
 - Intrinsic, 2-263
- Open User Logging file, information
 - LOGINFO, 2-252
 - LOGSTATUS, 2-255

- Operator Messages, A-10
- Optical Mark Reader, B-3
- Optional Capabilities, 1-9
 - Create Volumes, 1-10
 - Data Segment Management, 1-9
 - Multiple Resource Identification Number, 1-9
 - Privileged Mode, 1-9
 - Process Handling, 1-9
 - User Logging, 1-10
 - Volume Set Usage, 1-10

P

- Parameters
 - Definition, 2-4
 - Using GENMESSAGE To Insert in Messages, 5-62
- Parent process, request PIN of
 - FATHER, 2-65
- Parent process, return status
 - GETPROCINFO, 2-219
- Parse and validate file designator
 - FPARSE, 2-162
- Parse user defined command image
 - MYCOMMAND, 2-259
- PAUSE
 - Intrinsic, 2-266
 - Suspend calling process for time specified, 2-266
- Perform control operation on a file or device
 - FCONTROL, 2-82
- Permanently Privileged Programs, 3-1
- Physical record, write from stack to file
 - FWRITE, 2-191
- PIN, return for child process
 - GETPROCID, 2-218
- Place a session in break mode
 - CAUSEBREAK, 2-25
- Predefine characteristics of file and access
 - FOPEN, 2-141
- PRINT
 - Intrinsic, 2-267
 - Print character string on listing device, 2-267
- Print character sting on system console
 - PRINTOP, 2-270
- Print character string on listing device
 - PRINT, 2-267
- Print character string on system console/reply
 - PRINTOPREPLY, 2-271
- Printer, control operations
 - FDEVICECONTROL, 2-90
- Print file display on list device
 - PRINTFILEINFO, 2-269
- PRINTFILEINFO
 - Intrinsic, 2-269
 - Print file display on list device, 2-269
- PRINTOP

- Intrinsic, 2-270
- Print character string on system console, 2-270
- PRINTOPREPLY
 - Intrinsic, 2-271
 - Print character string on system console/reply, 2-271
- Priority of process, change
 - GETPRIORITY, 2-215
- Privileged Mode
 - Entering, 3-3
- Privileged Mode Capability, 1-9, 3-1
- Privileged mode (PM), enter dynamically
 - GETPRIVMODE, 2-217
- Privileged Programs
 - Permanent, 3-1
 - Temporary, 3-2
- Procedure, arm/disarm system trap handling
 - XSYSTRAP, 2-324
- Procedure, arm/disarm trap handling
 - XARITRAP, 2-318
 - XCONTRAP, 2-321
 - XLIBTRAP, 2-323
- Procedure Declarations, 1-2
- Process
 - Aborting, 5-28
 - Determining Source of Activation, 3-40
 - Obtaining Run Time, 5-56
 - Terminating, 5-27
- Process abort, arm system debugger
 - SETDUMP, 2-296
- Process, accept another subsystem break signal
 - RESETCONTROL, 2-291
- Process Break
 - Requesting, 5-26
- Process calling, abort
 - QUIT, 2-283
- Process, change priority
 - GETPRIORITY, 2-215
- Process creation
 - CREATE, 2-33
 - CREATEPROCESS, 2-39
- Process, delete calling
 - TERMINATE, 2-308
- Processes, 3-27
 - Active/Suspended Process Substates, 3-28
 - Creating/Activating, 3-29
 - Deleting, 3-34
 - Organization Of, 3-28
 - Rescheduling, 3-39
 - Scheduling, 3-6
 - Suspending, 3-34
- Process Handling Capability, 1-9, 3-26
- Process information, provide access
 - PROCINFO, 2-273
- Process Priority

- Determining, 3-40
- Process, receive mail from another
 - RECEIVEMAIL, 2-289
- Process, request PIN of parent
 - FATHER, 2-65
- Process, return CPU time
 - PROCTIME, 2-278
- Process, return source of activation
 - GETORIGIN, 2-214
- Process, send mail to another
 - SENDMAIL, 2-294
- Process State
 - Determining, 3-40
- Process structure, abort entire user
 - QUITPROG, 2-284
- Process, suspend
 - SUSPEND, 2-305
- Process, suspend for specified time
 - PAUSE, 2-266
- PROCINFO
 - Access to process information, 2-273
 - Intrinsic, 2-273
- PROCTIME
 - Intrinsic, 2-278
 - Return CPU time for process, 2-278
- Program
 - Aborting, 5-28
- Programmatic execution of command
 - COMMAND, 2-31
- Program, return to nonprivileged mode
 - GETUSERMODE, 2-221
- PTAPE
 - Intrinsic, 2-278
- PUTJCW
 - Assign value of JCW in JCW table, 2-281
 - Intrinsic, 2-281

Q

- QUIT
 - Abort calling process, 2-283
 - Intrinsic, 2-283
- QUITPROG
 - Abort entire user process structure, 2-284
 - Intrinsic, 2-284

R

- READ
 - Intrinsic, 2-285
 - Read \$STDIN ASCII string into array, 2-285
- Reading Input from Job/Session Input Device, 5-23
- Read logical record from current record pointer
 - FREADBACKWARD, 2-169
- Read logical record from direct access disk file
 - FREADDIR, 2-171

- Read logical record from file to stack
 - FREAD, 2-166
- Read reverse capability
 - FREADBACKWARD, 2-169
- Read \$STDIN ASCII string into array
 - READ, 2-285
- Read \$STDINX ASCII string into array
 - READX, 2-287
- Read user file label
 - FREADLABEL, 2-173
- READX
 - Intrinsic, 2-287
 - Read \$STDINX ASCII string into array, 2-287
- RECEIVEMAIL
 - Intrinsic, 2-289
 - Receive mail from another process, 2-289
- Receive mail from another process
 - RECEIVEMAIL, 2-289
- Receiving Mail, 3-38
- Record, move from disk file to buffer
 - FREADSEEK, 2-175
- Record pointer, move on tape/disk file
 - FSPACE, 2-186
- Record, write to User Logging
 - WRITELOG, 2-316
- Release extra data segment
 - FREEDSEG, 2-176
- Releasing
 - Global RINs, 3-44
- Remove disk file
 - FRENAME, 2-181
- Requesting a Process Break, 5-26
- Request PIN of parent process
 - FATHER, 2-65
- Rescheduling
 - Processes, 3-39
- RESETCONTROL
 - Accept another subsystem break signal, 2-291
 - Intrinsic, 2-291
- RESETDUMP
 - Disable abort stack analysis facility, 2-292
 - Intrinsic, 2-292
- Resource Management, 3-42
- Resources, 3-42
- Return access and status information for file
 - FGETINFO, 2-111
- Return calendar date
 - CALENDAR, 2-24
- Return CPU time for process
 - PROCTIME, 2-278
- Return current date and time
 - DATETIME, 2-48
- Return error information for file I/O errors
 - FCHECK, 2-71
- Return file label information of disk file

- FLABELINFO, 2-129
- Return from interrupt procedure
 - FINTEXT, 2-126
- Return information about user
 - WHO, 2-313
- Return PIN of child process
 - GETPROCID, 2-218
- Return program to nonprivileged mode
 - GETUSERMODE, 2-221
- Returns information about a file
 - FFILEINFO, 2-99
- Returns message corresponding to FCHECK error number
 - FERRMSG, 2-98
- Return source of activation call for process
 - GETORIGIN, 2-214
- Return status for parent/child process
 - GETPROCINFO, 2-219
- Return system defined JCW value
 - GETJCW, 2-212
- Return system time
 - CLOCK, 2-27
- Return system timer information
 - TIMER, 2-309
- Return user-supplied information
 - GETINFO, 2-210
- RIN, 3-43
- RIN, determine process PIN that locked local
 - LOCRINOWNER, 2-251
- RIN, lock global
 - LOCKGLORIN, 2-247
- RIN, lock local
 - LOCKLOCRIN, 2-249
- RINs
 - Acquiring Local, 3-49
 - Freeing Local, 3-51
 - Identifying Local Owners, 3-51
 - Interprocess Local Level, 3-49
 - Locking/Unlocking Local, 3-49
- RINs, free from allocation to job
 - FREELOCRIN, 2-178
- RIN, unlock global
 - UNLOCKGLORIN, 2-311
- RIN, unlock local
 - UNLOCKLOCRIN, 2-312
- Run-Time Messages, A-1

S

- Scheduling
 - Processes, 3-6
- SEARCH
 - Intrinsic, 2-293
 - Search array for specified entry/name, 2-293
- Search array for specified entry/name
 - SEARCH, 2-293

- Searching Arrays, 5-3
- Search JCW table and return value FINDJCW, 2-120
- Sending Mail, 3-37
- SENDMAIL
 - Intrinsic, 2-294
 - Send mail to another process, 2-294
- Send mail to another process
 - SENDMAIL, 2-294
- Session
 - Identifying with JOBINF, 5-14
- Session information, access
 - JOBINF, 2-237
- Session, initiate on specified terminal
 - STARTSESS, 2-300
- Session, place in break mode
 - CAUSEBREAK, 2-25
- Set bits in stream JCW
 - SETJCW, 2-297
- SETDUMP
 - Arm system debugger from process abort, 2-296
 - Intrinsic, 2-296
- SETJCW
 - Intrinsic, 2-297
 - Set bits in stream JCW, 2-297
- Set logical record pointer for disk file
 - FPOINT, 2-165
- Size, alter data segment
 - ALTDSEG, 2-15
- SL procedure, load
 - LOADPROC, 2-245
- SL procedure, unloads
 - UNLOADPROC, 2-310
- Software interrupts, enable/disable
 - FINTSTATE, 2-127
- Software library trap handling procedure, arm/disarm
 - XLIBTRAP, 2-323
- Son Process
 - Determining, 3-40
- Source of activation call for process, return
 - GETORIGIN, 2-214
- Special Considerations, 2-5
- SPL
 - Calling Intrinsics From, 1-2
- Split-Stack Operations, 2-6
- Spooled devicefile, control operations
 - FDEVICECONTROL, 2-90
- Stack, alter DB to Z area
 - ZSIZE, 2-325
- STACKDUMP
 - Execute full stack trace, 2-298
 - Intrinsic, 2-298
- Stack, expand/contract
 - DLSIZE, 2-51
- Stack, read logical record from file



- FREAD, 2-166
- Stack Sizes
 - Changing, 5-31
- Stack trace, execute full
 - STACKDUMP, 2-298
- Standard Traps, 5-40
- STARTSESS
 - Initiate session on specified terminal, 2-300
 - Intrinsic, 2-300
- Status for parent/child process, return
 - GETPROCINFO, 2-219
- Status information returned, file
 - FGETINFO, 2-111
- STDIN
 - Opening, 4-31
 - Using FREAD/FWRITE With, 4-30
- STDLIST
 - Opening, 4-32
 - Using FREAD/FWRITE With, 4-30
- String conversion, ASCII to EBCDIC
 - CTRANSLATE, 2-44
- String conversion, EBCDIC to ASCII
 - CTRANSLATE, 2-44
- String conversion, EBCDIK to JIS
 - CTRANSLATE, 2-44
- String conversion, JIS to EBCDIK
 - CTRANSLATE, 2-44
- String, convert ASCII to binary
 - BINARY, 2-23
- Subqueue Priority Classes, 3-6
- Subsystem break signal, accept another
 - RESETCONTROL, 2-291
- SUSPEND
 - Intrinsic, 2-305
 - Suspend a process, 2-305
- Suspend a process
 - SUSPEND, 2-305
- Suspend calling process for time specified
 - PAUSE, 2-266
- Suspended Process Substates, 3-28
- Suspending
 - The Calling Process, 5-26
- Suspending Processes, 3-34
- SWITCHDB
 - Intrinsic, 2-306
- Syntax
 - Description, 2-1
- System Console
 - Requesting a Reply From, 5-26
 - Writing Output to, 5-26
 - Writing Output To, 5-25
- System Console Messages, A-1
- System debugger, arm from process abort
 - SETDUMP, 2-296
- System-defined files, assign and create process

- CREATEPROCESS, 2-39
- System defined JCW value, return
 - GETJCW, 2-212
- System Failure Messages, A-1
- System Messages, A-10
- System Operator Messages, A-1
- System Time
 - Obtaining Information From, 5-53
- System time, return
 - CLOCK, 2-27
- System timer information, return
 - TIMER, 2-309
- System Trap, 5-46
- System trap handling procedure, arm/disarm
 - XSYSTRAP, 2-324

T

- Table search, return JCW value FINDJCW, 2-120
- Tape file, move record pointer
 - FSPACE, 2-186
- Tape, read logical record
 - FREADBACKWARD, 2-169
- Temporarily Privileged Programs, 3-2
- Terminal Carriage Control Codes, B-2
- Terminal, initiate specified session
 - STARTSESS, 2-300
- TERMINATE
 - Delete calling process, 2-308
 - Intrinsic, 2-308
- Terminate access to a file
 - FCLOSE, 2-78
- Terminating a Process, 5-27
- Testing
 - Mailbox Status, 3-36
- Test mailbox status
 - MAIL, 2-257
- Time
 - Obtaining Current, 5-54
- Time and Date Intrinsics, 5-53
- Time and date, return
 - DATELINE, 2-48
- Time Information
 - Formatting, 5-56
- Time of day, format
 - FMTCLOCK, 2-139
 - FMTDATE, 2-140
- TIMER
 - Intrinsic, 2-309
 - Return system timer information, 2-309
- Time, return actual system
 - CLOCK, 2-27
- Transaction, mark beginning of user logging
 - BEGINLOG, 2-21
- Transferring
 - Data From Extra Data Segment to Stack, 3-25

- Data From Stack to Extra Data Segment, 3-25
- Translating
 - Characters with CTRANSLATE Intrinsic, 5-22
- Transmitting Program Input/Output
 - From Job/Session Devices, 5-23
- Trap handling procedure, arm/disarm
 - XARITRAP, 2-318
 - XCONTRAP, 2-321
 - XLIBTRAP, 2-323
- Traps, 1-7
 - Arithmetic, 5-39
 - Commercial Instruction, 5-41
 - CONTROL-Y, 5-49
 - Disabling, 5-38
 - Enabling, 5-38
 - Extended Precision Floating Point, 5-40
 - Library, 5-45
 - Standard, 5-40
 - System, 5-46

U

- Unloading
 - Dynamic, 5-3
- UNLOADPROC
 - Dynamically unloads segmented library (SL)
 - procedure, 2-310
 - Intrinsic, 2-310
- Unlock file, dynamically
 - FUNLOCK, 2-188
- Unlock global RIN
 - UNLOCKGLORIN, 2-311
- UNLOCKGLORIN
 - Intrinsic, 2-311
 - Unlock global RIN, 2-311
- Unlocking
 - Global RINs, 3-45
 - Local RINs, 3-49
- Unlock local RIN
 - UNLOCKLOCRIN, 2-312
- UNLOCKLOCRIN
 - Intrinsic, 2-312
 - Unlock local RIN, 2-312
- Update logical record in disk file
 - FUPDATE, 2-189
- User Access Mode
 - Determining, 5-12
- User Attributes
 - Determining, 5-12
- User defined command image, parse
 - MYCOMMAND, 2-259
- User-Defined Job Control Words, 5-59
- User file label, read
 - FREADLABEL, 2-173
- User file label, write
 - FWRITELABEL, 2-201

- User Logging, 3-52
 - Procedures, 3-59
- User Logging Capability, 1-10
- User Logging facility, access
 - OPENLOG, 2-263
- user logging facility, close access
 - CLOSELOG, 2-28
- User Logging file, provide information
 - LOGINFO, 2-252
 - LOGSTATUS, 2-255
- User Logging memory buffer, flush contents
 - FLUSHLOG, 2-136
- user logging transaction, mark beginning
 - BEGINLOG, 2-21
- user logging transaction, mark end
 - ENDLOG, 2-60
- User Logging, write record
 - WRITELOG, 2-316
- User Messages, A-9
- User Processes, Organization Of, 3-28
- User, return information about
 - WHO, 2-313
- User-supplied information, return
 - GETINFO, 2-210
- Uses
 - Log File, 3-59
- Using
 - FERRMSG, 4-41
 - GENMESSAGE to Insert Parameters in Mes-
 - sages, 5-62
 - IOWAIT Intrinsic, 4-42
- Using Files, 4-6
- Using FREAD/FWRITE with \$STDIN/\$STDLIST,
 - 4-30
- USL file, change length
 - EXPANDUSLF, 2-63
- USL file, delete inactive entries
 - CLEANUSL, 2-26
- USL file, initialize to empty state
 - INITUSLF, 2-232
- Utility Functions, MPE, 5-1

V

- Validate and parse file designator
 - FPARSE, 2-162
- Value returned, search JCW table FINDJCW, 2-
 - 120
- Value, return system defined JCW
 - GETJCW, 2-212
- Volume Set Usage Capability, 1-10

W

WHO

- Intrinsic, 2-313

- Return information about user, 2-313

WRITELOG

- Intrinsic, 2-316

- Write record to User Logging file, 2-316

- Write logical/physical record from stack to file

- FWRITE, 2-191

- Write logical record from stack to disk file

- FWRITEDIR, 2-199

- Write record to User Logging file

- WRITELOG, 2-316

- Write user file label

- FWRITELABEL, 2-201

- Writing File System Error-Check Procedure, 4-40

- Writing Output

- To System Console, Requesting A Reply, 5-26

- Writing Output to Job/Session List Device, 5-25

- Writing Output to System Console, 5-25

X

XARITRAP

- Arm/disarm arithmetic trap handling procedure, 2-318

- Intrinsic, 2-318

XCONTRAP

- Arm/disarm subsystem break (Control-Y) trap handling procedure, 2-321

- Intrinsic, 2-321

XLIBTRAP

- Arm/disarm software library trap handling procedure, 2-323

- Intrinsic, 2-323

XSYSTRAP

- Arm/disarm system trap handling procedure, 2-324

- Intrinsic, 2-324

Z

ZSIZE

- Alter DB to Z area of stack, 2-325

- Intrinsic, 2-325

- Z to DB Area Size, Changing, 5-37

Part No. 32033-90007
Printed in U.S.A. 1989
E1089

