

MPE V Intrinsic Reference Manual



HP 3000 Computer Systems

MPE V INTRINSICS

Reference Manual



19447 PRUNERIDGE AVENUE, CUPERTINO, CA 95014

Part No. 32033-90007
E0286

Printed in U.S.A. 02/86

NOTICE

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

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

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

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

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition, and lists the dates of all changed pages. Unchanged pages are listed as "ORIGINAL". Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars and dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition JAN 1985
Second Edition FEB 1986

Effective Pages	Date
ALL	FEB 1986

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update.

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

First Edition	JAN 1985	E/F.00.00, G.00.00, G.01.00
Second Edition	FEB 1986	G.02.00

MPE V MANUAL PLAN

INTRODUCTORY LEVEL:

GENERAL
INFORMATION
Manual
5953-7553

GUIDE FOR THE
NEW USER
32033-90009

GUIDE FOR THE
NEW OPERATOR
32033-90021

STANDARD USER LEVEL:

MPE V COMMANDS
Reference
Manual
32033-90006

MPE V INTRINSICS
Reference
Manual
32033-90007

MPE V UTILITIES
Reference
Manual
32033-90008

SEGMENTER
Reference
Manual
30000-90011

DEBUG/STACK DUMP
Reference
Manual
30000-90012

FILE SYSTEM
Reference
Manual
30000-90236

ADMINISTRATIVE LEVEL:



MPE V SYSTEM OPERATION
& RESOURCE MANAGEMENT
Reference Manual
32033-90005

SUMMARY LEVEL:

MPE V
REFERENCE GUIDE
30000-90049

There are many more manuals applicable to the HP 3000. A complete list may be found in every issue of the MPE V Communicator. Please contact your System Manager.

CONVENTIONS USED IN THIS MANUAL

NOTATION	DESCRIPTION
COMMAND	Commands are shown in CAPITAL LETTERS. The names must contain no blanks and be delimited by a non-alphabetic character (usually a blank).
KEYWORDS	Literal keywords, which are entered optionally but exactly as specified, appear in CAPITAL LETTERS.
<i>parameter</i>	Required parameters, for which you must substitute a value, appear in <i>bold italics</i> .
<i>parameter</i>	Optional parameters, for which you may substitute a value, appear in <i>standard italics</i> .
[]	<p>An element inside brackets is optional. Several elements stacked inside a pair of brackets means the user may select any one or none of these elements.</p> <p>Example: [A] [B] user may select A or B or neither.</p> <p>When brackets are nested, parameters in inner brackets can only be specified if parameters in outer brackets or comma place-holders are specified.</p> <p>Example: [<i>parm1</i>[,<i>parm2</i>[,<i>parm3</i>]]] may be entered as:</p> <p style="text-align: center;"><i>parm1,parm2,parm3</i> or <i>parm1,,parm3</i> or <i>,,parm3</i> , etc.</p>
{ }	<p>When several elements are stacked within braces the user <i>must</i> select one of these elements.</p> <p>Example: { A } { B } user must select A or B.</p>
...	An ellipsis indicates that a previous bracketed element may be repeated, or that elements have been omitted.
<u>user input</u>	In examples of interactive dialog, user input is underlined. Example: NEW NAME? <u>ALPHA1</u>
superscript ^C	Control characters are indicated by a superscript ^C . Example: Y ^C . (Press Y and the CNTL key simultaneously.)
	 indicates a terminal key. The legend appears inside.
<<COMMENT>>	Programmer's comments in listings appear within << >>.
** Comment **	Editor's comments appear in this form.

CONTENTS

Section	Page
PREFACE	xv

Section I

INTRODUCTION TO MPE INTRINSICS

INTRINSIC CALLS	1-1
Calling Intrinsic From SPL	1-1
Procedure Declarations	1-2
Intrinsic Declarations	1-2
Implementing Intrinsic Calls	1-2
Calling Intrinsic From Languages Other Than SPL	1-4
INTRINSIC CALL ERRORS	1-5
MPE INTRINSICS AND THEIR FUNCTIONS	1-7
OPTIONAL CAPABILITIES	1-7

Section II

INTRINSIC DESCRIPTIONS

INTRINSIC NAME	2-1
SYNTAX	2-1
FUNCTIONAL RETURN	2-2
PARAMETERS	2-3
CONDITION CODES	2-3
SPECIAL CONSIDERATIONS	2-3
Required Capability	2-3
Split-Stack Operations	2-4
ADDITIONAL DISCUSSION	2-4
ABORTSESS	2-5
ACTIVATE	2-7
ADJUSTUSLF	2-9
ALTDSEG	2-11
ARITRAP	2-13
ASCII	2-14
BEGINLOG	2-16
BINARY	2-18
CALENDAR	2-19
CAUSEBREAK	2-20
CLEANUSL	2-21
CLOCK	2-23
CLOSELOG	2-24
COMMAND	2-26
CREATE	2-27
CREATEPROCESS	2-32
CTranslate	2-36
DASCII	2-38
DATELINE	2-40
DBINARY	2-41
DEBUG	2-42
DLsize	2-43
DMOVIN	2-45

CONTENTS (Continued)

INTRINSIC DESCRIPTIONS (Continued)	Page
DMOVOUT	2-47
ENDLOG	2-49
EXPANDUSLF	2-51
FATHER	2-53
FCARD	2-54
FCHECK	2-58
FCLOSE	2-65
FCONTROL	2-68
FDELETE	2-74
FDEVICECONTROL	2-75
FERRMSG	2-84
FFILEINFO	2-85
FGETINFO	2-88
FINDJCW	2-96
FINTEXT	2-98
FINTSTATE	2-99
FLABELINFO	2-101
FLOCK	2-103
FLUSHLOG	2-106
FMTCALENDAR	2-108
FMTCLOCK	2-109
FMTDATE	2-110
FOPEN	2-111
FPARSE	2-130
FPOINT	2-133
FREAD	2-135
FREADBACKWARD	2-137
FREADDIR	2-139
FREADLABEL	2-141
FREADSEEK	2-143
FREEDSEG	2-144
FREELOCRIN	2-145
FRELATE	2-146
FRENAME	2-148
FSETMODE	2-150
FSPACE	2-153
FUNLOCK	2-155
FUPDATE	2-156
FWRITE	2-158
FWRITEDIR	2-164
FWRITELABEL	2-166
GENMESSAGE	2-167
GETDSEG	2-171
GETINFO	2-173
GETJCW	2-175
GETLOCRIN	2-176
GETORIGIN	2-177
GETPRIORITY	2-178
GETPRIVMODE	2-180
GETPROCID	2-181
GETPROCINFO	2-182

CONTENTS (Continued)

INTRINSIC DESCRIPTIONS (Continued)	Page
GETUSERMODE	2-184
INITUSLF	2-185
IODONTWAIT	2-186
IOWAIT	2-188
JOBINFO	2-190
KILL	2-194
LOADPROC	2-195
LOCKGLORIN	2-196
LOCKLOCRIN	2-198
LOCRINOWNER	2-200
LOGINFO	2-201
LOGSTATUS	2-204
MAIL	2-206
MYCOMMAND	2-208
OPENLOG	2-211
PAUSE	2-213
PRINT	2-214
PRINTFILEINFO	2-216
PRINTOP	2-217
PRINTOPREPLY	2-218
PROCINFO	2-220
PROCTIME	2-223
PTAPE	2-224
PUTJCW	2-226
QUIT	2-228
QUITPROG	2-229
READ	2-230
READX	2-232
RECEIVEMAIL	2-234
RESETCONTROL	2-236
RESETDUMP	2-237
SEARCH	2-238
SENDMAIL	2-239
SETDUMP	2-241
SETJCW	2-242
STACKDUMP	2-243
STARTSESS	2-245
SUSPEND	2-248
SWITCHDB	2-250
TERMINATE	2-251
TIMER	2-252
UNLOADPROC	2-253
UNLOCKGLORIN	2-254
UNLOCKLOCRIN	2-255
WHO	2-256
WRITELOG	2-260
XARITRAP	2-262
XCONTRAP	2-264
XLIBTRAP	2-266
XSYSTRAP	2-267
ZSIZE	2-268

CONTENTS (Continued)

Section III	Page
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	3-5
DATA SEGMENT MANAGEMENT CAPABILITY	3-9
Creating an Extra Data Segment	3-10
Deleting an Extra Data Segment	3-21
Transferring Data From an Extra Data Segment to the Stack	3-21
Transferring Data from the Stack to an Extra Data Segment	3-21
Changing the Size of an Extra Data Segment	3-22
PROCESS HANDLING CAPABILITY	3-22
Processes	3-23
Organization of User Processes	3-23
Active and Suspended Process Substates	3-24
Creating and Activating Processes	3-24
Suspending Processes	3-29
Deleting Processes	3-29
Interprocess Communication	3-31
Testing Mailbox Status	3-32
Sending Mail	3-32
Receiving (Collecting) Mail	3-33
Avoiding Deadlocks	3-34
Rescheduling Processes	3-34
Determining Source of Activation	3-35
Determining Father Process	3-35
Determining Son Processes	3-36
Determining Process Priority and State	3-36
RESOURCE MANAGEMENT	3-37
Inter-Job Level (Global) RINs	3-38
Acquiring Global RINs	3-38
Releasing Global RINs	3-39
Locking and Unlocking Global RINs	3-39
Interprocess (Local) Level RINs	3-43
Acquiring Local RINs	3-43
Locking and Unlocking Local RINs	3-43
Identifying Local RIN Owners	3-44
Freeing Local RINs	3-45
USER LOGGING	3-45
How User Logging Works	3-46
User Logging Procedures	3-50
Suggested Log File Uses	3-51

CONTENTS (Continued)

Section IV	Page
ACCESSING AND ALTERING FILES	
FILE DEVICE RELATIONSHIPS	4-2
Non-Sharable Device Access	4-2
File Domains	4-2
Opening a File	4-3
Files on Non-Sharable Devices	4-4
HOW TO USE FILES	4-5
Internal Operations for File Accessing	4-5
Parsing and Validating File Designators	4-15
Opening a New Disc File	4-17
Opening an Old Disc File	4-20
Opening a File on a Device Other Than Disc	4-22
Using FREAD and FWRITE with \$STDIN and \$STDLIST	4-23
Opening \$STDIN	4-25
Opening \$STDLIST	4-25
CLOSING FILES.	4-28
Closing a New File as a Temporary File	4-28
Closing a New File as a Permanent File	4-31
WRITING A FILE SYSTEM ERROR-CHECK PROCEDURE	4-33
USING FERRMSG	4-33
USING THE IOWAIT INTRINSIC	4-35
DECLARING ACCESS-MODE OPTIONS	4-38

Section V	
OTHER APPLICATIONS OF MPE INTRINSICS	
DYNAMIC LOADING AND UNLOADING OF LIBRARY PROCEDURES	5-2
Dynamic Loading	5-3
Dynamic Unloading	5-3
SEARCHING ARRAYS	5-4
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	5-16
CONVERTING NUMBERS FROM AN ASCII NUMERIC STRING TO A BINARY CODED VALUE	5-20
TRANSLATING CHARACTERS WITH THE CTRANSLATE INTRINSIC.	5-22
TRANSMITTING PROGRAM I/O FROM JOB/SESSION I/O 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	5-27
ABORTING A PROCESS	5-28
ABORTING A PROGRAM	5-28
CHANGING STACK SIZES	5-30

CONTENTS (Continued)

OTHER APPLICATIONS OF MPE INTRINSICS (Continued)	Page
Changing the DL to DB Area Size	5-31
Changing the Z to DB Area Size	5-37
ENABLING AND DISABLING TRAPS	5-37
Arithmetic Traps	5-38
Standard Traps	5-39
Extended Precision Floating Point Traps	5-39
Commercial Instruction Traps	5-40
Library Trap	5-42
System Trap	5-44
CONTROL-Y Traps	5-46
TIME AND DATE INTRINSICS	5-49
Obtaining System Timer Information	5-50
Obtaining the Current Time	5-50
Obtaining the Calendar Date	5-52
Obtaining Process Run Time	5-52
Formatting Calendar Date and Time Information	5-52
JOB CONTROL WORDS	5-53
INTERPROCESS COMMUNICATION	5-54
USER-DEFINED JOB CONTROL WORDS	5-55
MPE MESSAGE FACILITY	5-56
Message Catalog	5-56
MAKECAT Program	5-57
Using GENMESSAGE to Insert Parameters in Messages	5-59
APPLICATION MESSAGE FACILITY	5-59

Appendix A

MPE DIAGNOSTIC MESSAGES

RUN-TIME MESSAGES	A-1
USER MESSAGES	A-10
OPERATOR MESSAGES	A-10
SYSTEM MESSAGES	A-10
FILE INFORMATION DISPLAY	A-12

Appendix B

DEVICE CHARACTERISTICS

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 the HP 7260A Optical Mark Reader	B-3
ASCII and Column Image Reading Formats	B-4

ILLUSTRATIONS

Title	Page
Calling the PRINTOP Intrinsic From SPL	1-4
Using Numeric Values as Parameters in an Intrinsic Call	1-4
Condition Code Checks	1-7
Foptions Bit Summary	2-95
Aoptions Bit Summary	2-95
Carriage Control Summary	2-163
Error Codes Returned for PROCINFO	2-221
Information Options for PROCINFO	2-222
Using the GETPRIVMODE and GETUSERMODE Intrinsics (Program DSINIT)	3-4
Master Queue Structure	3-7
Using the GETDSEG and DMOVOUT Intrinsics (Program DSINIT)	3-12
Creating and Activating Two Son Processes (Program DSBOSS)	3-13
Using the GETDSEG and DMOVIN Intrinsics (Program DSACCS)	3-14
Using the CREATE and ACTIVATE Intrinsics (Program PROG)	3-26
Process Deletion	3-30
Using the LOCKGLORIN and UNLOCKGLORIN Intrinsics	3-40
BOOKFILE	3-41
User Logging Facility	3-47
File Access Interface for New Disc Files	4-7
Directory File Name and Label Address Pointer	4-8
File Access Interface for Old Disc Files	4-9
Device Allocation Flowchart	4-13
Opening a New Disc File	4-19
Opening an Old Disc File	4-21
Opening a File on a Device Other Than Disc	4-24
Opening \$STDIN and \$STDLIST	4-27
Closing a New File as a Temporary File	4-30
Closing a New File as a Permanent File	4-32
Error-Check Procedure Example	4-34
Using the IOWAIT Intrinsic	4-37
Using the MYCOMMAND Intrinsic (Program UTILY)	5-6
Using the WHO Intrinsic	5-13
Using the ASCII Intrinsic	5-17
Using the DASCII Intrinsic	5-19
Using the BINARY Intrinsic	5-21
Using the PRINT and READ Intrinsics	5-24
Using the QUIT Intrinsic	5-29
Expanding and Contracting the DL to DB Area	5-32
Using the DLSIZE Intrinsic (Program DLAREA)	5-33
Changing the DL to DB Area Size (Program DLAREA)	5-36
Using the XARITRAP Intrinsic (Program ATRAP)	5-41
Using the XCONTRAP Intrinsic (Program CONTY)	5-49
Using the TIMER Intrinsic	5-51
FMTCALENDAR, FMTCLOCK, and FMTDATE Intrinsics Example	5-53
GENMESSAGE Intrinsic Example	5-61
FCARD Intrinsic Example	B-5

TABLES

Title	Page
Compatible Data Types	1-5
Summary of MPE Intrinsics	1-9
Item Values Returned by CREATEPROCESS	2-34
File System Error Codes	2-59
Item Values Returned by FFILEINFO	2-86
Item Values Returned by FLABELINFO	2-103
Carriage-Control Directives	2-161
Item Values Returned by JOBINFO	2-193
Item Values Returned by LOGININFO	2-203
Device-Dependent Restrictions	4-11
Classification of Devices	4-12
"PROGRAM TYPE" Error Messages	A-5
"INTRINSIC" Error Message Numbers	A-5
"RUN-TIME" Error Messages	A-7
"CREATE" Error Messages	A-7
"ACTIVATE" Error Messages	A-7
"SUSPEND" Error Messages	A-7
"MYCOMMAND" Error Messages	A-8
"LOCKGLORIN" Error Messages	A-8
"LOADER" Error Messages and Warnings	A-8
System Messages	A-10

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.02.00 release of MPE. To assist you in locating information, a brief description of each section in this manual follows:

Section I	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.
Section II	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. The FLABELINFO, FPARSE, LOGINFO, and GETINFO intrinsics are new effective with the G.02.00 release of MPE. They are documented in this section.
Section III	OPTIONAL CAPABILITIES MPE functions which must be performed by users with specifically assigned optional capabilities.
Section IV	ACCESSING AND ALTERING FILES Background information on the MPE file system, including descriptions of some typical file-related operations performed with MPE intrinsics.
Section V	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.

This manual replaces the First Edition of the MPE V Intrinsics Manual (32033-90007). 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).

INTRODUCTION TO MPE INTRINSICS

SECTION

I

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, COBOLII, 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 Section II 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 (O-V) appears in the intrinsic syntax as shown in Section II, 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 Section II 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 Section II 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 Section II 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 (60) 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 Intrinsics From Languages Other Than SPL

Direct calls to intrinsics are allowed in some languages. These languages include BASIC, COBOLII, 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	COBOLII	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	Char
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:

- 00 Is CCG, or Condition Code Greater Than (>).
- 01 Is CCL, or Condition Code Less Than (<).
- 10 Is CCE, or Condition Code Equal (=).
- 11 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 Section II for a description of these meanings). For a more detailed discussion of condition codes, refer to the Machine Instruction Set Reference Manual (30000-90022).

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.)

Condition Code State	SPL Branch Word	General Meaning
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 V, "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. Additional special actions taken by a particular intrinsic are described in the "Special Considerations" discussion of that intrinsic in Section II.

Figure 1-3 illustrates the use of condition code checks in a program. If the condition code is CCE, 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

MPE INTRINSICS AND THEIR FUNCTIONS

MPE intrinsics allow you to access and alter files, request various utility functions, and access and manage system resources. When intrinsics 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 intrinsics, refer to Table 1-2, which lists each intrinsic, its purpose, and the capability needed to use it.

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.

Introduction To MPE Intrinsic

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 disc. If the data base is lost, the logging tape or disc 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).

The Volume Set Usage (UV) or Create Volumes (CV) capabilities allows you to maintain files on private disc 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 disc file with the FOPEN intrinsic. Similarly, when you close and save a disc 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 Section III.

Table 1-2. Summary of MPE Intrinsic

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) and 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) and 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)

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
CTranslate	Converts a string of characters between EBCDIC and ASCII or between EBCDIC and JIS (KANA8).	Standard
DASCI	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)
ENDLOG	Marks the end of a user logging transaction.	User Logging (LG) and 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
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

Table 1-2. Summary of MPE Intrinsics (Continued)

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FINDJCW	Searches Job Control Word Table for named JCW.	Standard
FINTEXT	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 disc 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) and 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 disc file.	Standard
FREAD	Reads logical record from file to user's stack.	Standard
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 disc file to a buffer in anticipation of a FREADDIR intrinsic call.	Standard
FREEDSEG	Releases an extra data segment.	Data Segment Management (DS)
FREELOCRIN	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 disc file.	Standard

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
FSETMODE	Activates or deactivates file access modes.	Standard
FSPACE	Moves a physical record pointer forward or backward on a tape or disc file.	Standard
FUNLOCK	Dynamically unlocks a file.	Standard
FUPDATE	Updates (writes) a logical record in a disc 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 disc file.	Standard
FWRITELABEL	Writes a user file label.	Standard
GENMESSAGE	Accesses the MPE message system.	Standard
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
GETLOCRI	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)
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

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
KILL	Deletes a son process.	Process Handling (PH)
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) and System Supervisor (OP)
LOGSTATUS	Provides information about an opened user logging file.	User Logging (LG) and 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) and 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 disc file.	Standard
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

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

INTRINSIC NAME	PURPOSE	CAPABILITY REQUIRED
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
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) and 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

INTRINSIC DESCRIPTIONS

SECTION

II

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 its assigned number and a brief summary of its function. The intrinsic number is useful in determining error diagnosis and for implementing trap procedures. Several intrinsics may have the same number (e.g. BEGINLOG, ENDLOG, and WRITELOG) since they are really the same procedure with alternate entry points. Other intrinsics have no number at all, because they will not abort and therefore do not need an intrinsic number for error diagnosis.

SYNTAX

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

0-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 over ***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 0-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 intrinsic is 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 section 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
<i>length</i> := READ(<i>message</i> , <i>expectedl</i>);		

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 *expectedl* 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:

```
LEN:=READ(MESS,20);
```

```
** User program may access LEN **
```

```
READ(MESS,20)
```

```
** Return variable unavailable **
```

PARAMETERS

All parameters are described. In the intrinsic call description, required parameters are shown in ***bold italics***, and optional parameters are shown in large (12-point) *standard italics*. Within the text of this manual, this distinction is not shown for required and optional parameters, and all parameters are shown in small (10-point) *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 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.

ADDITIONAL DISCUSSION

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

Aborts the specified session from the system. (Available only on version G.01.00 and later.)

SYNTAX

```
IV      DV      IA
ABORTSESS(jsid,jsnum,errorstat);
```

ABORTSESS provides programmatic access to the functions of the :ABORTJOB command. The caller of ABORTSESS must have :ABORTJOB rights to the specified session. :ABORTJOB rights can be distributed with the :ALLOW command, or the :JOBSECURITY can be set low.

PARAMETERS

- jsid*** *integer by value (required)*
Indicates the type of Command Interpreter (CI) process. If an earlier STARTSESS call was successful, the value returned is 1; if unsuccessful, the value is 0. When the value returned to *jsid* is 1, *jsid* and *jsnum* can be used as input to JOBINFO to check on the attributes of the session.
- jsnum*** *double by value (required)*
A 32-bit value which, when used with *jsid*, uniquely identifies the session.
- errorstat*** *integer array (required)*
A two-element array in which the status of the call is returned. The second element is reserved for future use, and will always contain a zero. The first element will contain a zero if no errors occurred. If an error occurs one of the following error values is returned in the first element:

Error No.	Meaning
1	Job security too high, or job is not yours.
2	Job does not exist.
3	Job is being introduced and cannot be aborted when in the INTRO state.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

ABORTSESS requires the caller to be allowed :ABORTJOB via the :ALLOW command, have Account Manager (AM) or System Manager (SM) capabilities, or that the :JOBSECURITY be set low.

ADDITIONAL DISCUSSION

MPE V Commands Manual (32033-90006).

Activates a process.

SYNTAX

O-V IV LV

ACTIVATE(*pin*,*susp*);

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

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

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

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

PARAMETERS

pin

integer by value (required)

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

susp

logical by value (optional)

A word that specifies one of the following:

- The calling process is to be suspended while the called process is activated and commences execution.
- The called process is to be activated by the operating system but will not commence execution immediately. Instead, control is returned to the calling process which will continue execution.

When *susp* is omitted or is zero, the calling process remains active. When *susp* is specified (and not zero), the calling process is suspended. The bits (14:2) of *susp* specify the anticipated source of the call that later will reactivate the calling process.

Bit (15:1) Father activation bit.

=0 The process does not expect to be activated by its father.

=1 The process expects to be activated by its father.

Bit (14:1) Son activation bit.

=0 The process does not expect to be activated by one of its sons.

=1 The process expects to be activated by one of its sons.

If both bits=1, the suspended process can be activated by either its father or one of its sons.

Bits (0:14) Reserved for MPE and should be set to zero.

Default: Calling process remains active.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Creating an Extra Data Segment" and "Creating and Activating Processes" in Section III.

Adjusts directory space in a USL file.

SYNTAX

I	IV	IV
<i>errnum</i> :=	ADJUSTUSLF	(<i>uslfnum</i> , <i>records</i>);

The ADJUSTUSLF intrinsic moves the start of the information block forward or backward on a User Subprogram Library (USL) file, thereby increasing or decreasing, respectively, the space available for the file directory block. This does not change the overall length of the file. This intrinsic is intended for programmers writing compilers. Refer to the MPE Segmenter Reference Manual (30000-90011) for a discussion of USLs, the ADJUSTUSLF intrinsic, information blocks, and directory blocks.

FUNCTIONAL RETURN

errnum

integer

Returns an error number if an error occurs. If no error occurs, no value is returned. The error number returned corresponds to the following errors:

Error No.	Meaning
0	The file specified by <i>uslfnum</i> was empty, an unexpected end-of-file was encountered when reading the <i>uslfnum</i> , or an unexpected end-of-file was encountered when writing on the <i>uslfnum</i> .
1	Unexpected input/output error occurred.
4	Your request attempted to exceed the maximum file directory size (32,768 words).
5	Insufficient directory space.
6	Insufficient space was available in the USL file information block.

PARAMETERS

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

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied. An error number was returned to <i>errornum</i> indicating the reason for this failure.

ADDITIONAL DISCUSSION

MPE Segmenter Reference Manual (30000-90011).

Alters the size of an extra data segment.

SYNTAX

```

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

```

The ALTDSEG intrinsic alters the current size of an extra data segment. ALTDSEG can be used to reduce the storage required by the segment when it is moved into main memory, then used again to expand storage as required, thus allowing more efficient use of memory.

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

Present Segment Size (Words)	Change Value (<i>inc</i>) (Words)	New Segment Size (Words)
128	-3	128
128	-4	124
128	+1	132
128	+3	132
128	+4	132

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

In no case may ALTDSEG increase the size of a data segment to exceed the virtual space originally allocated through GETDSEG. On version G.00.00 and later when GETDSEG is called in non-Privileged Mode, the ALTDSEG intrinsic must also be called in non-Privileged Mode. When GETDSEG is called in Privileged Mode, the ALTDSEG intrinsic must be called in Privileged Mode.

PARAMETERS

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

inc *integer by value (required)*
The value, in words, by which the data segment is to be changed. A positive integer value requests an increase, and a negative integer value requests a decrease.

size

integer (required)

A word to which the new size of the data segment is returned after incrementing or decrementing occurs.

CONDITION CODES

CCE

Request granted.

CCG

Request not fully granted. An illegal decrement requesting a new total segment size of zero or less, or an illegal increment requesting a new size greater than the virtual space originally assigned by GETDSEG, was attempted. In the first case, the current size remains in effect. In the second case, the size of the virtual space is granted and this size is returned through the *size* parameter.

CCL

Request denied because an illegal *index* parameter was specified.

SPECIAL CONSIDERATIONS

Data Segment Management (DS) capability required.

ADDITIONAL DISCUSSION

"Changing the Size of an Extra Data Segment" in Section III.

Enables or disables internal interrupt signals from all hardware arithmetic traps.

SYNTAX

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

When a user process begins execution, all internal arithmetic user traps are enabled. That is, if an arithmetic error occurs in the user process, it is aborted in the trap mechanism. The possible interrupts listed below are collectively called the arithmetic user traps:

- Integer Overflow.
- Floating Point Overflow.
- Floating Point Underflow.
- Integer Divide By Zero.
- Floating Point Divide By Zero.
- Double Precision Overflow.
- Double Precision Underflow.
- Double Precision Divide By Zero.
- Decimal Overflow.
- Invalid ASCII Digit.
- Invalid Decimal Digit.
- Invalid Source Word Count.
- Invalid Decimal Operand Length.
- Decimal Divide By Zero.

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

The ARITRAP intrinsic always clears the overflow indicator (bit (4:1)) located in the caller's status word.

PARAMETERS

state *logical by value (required)*
A word in which bit (15:1) specifies whether arithmetic traps are enabled or disabled. The settings for bit (15:1) are as follows:

=0 Arithmetic traps are disabled.

=1 Arithmetic traps are enabled.

Bits (0:15) are reserved for MPE and should be set to zero.

CONDITION CODES

CCE	Request granted. The arithmetic traps were originally disabled.
CCG	Request granted. The arithmetic traps were originally enabled.
CCL	Not returned by this intrinsic.

ADDITIONAL DISCUSSION

"Enabling and Disabling Traps" in Section V.

ASCII

INTRINSIC NUMBER 63

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

SYNTAX

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

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

FUNCTIONAL RETURN

numchar *integer*
The number of characters in the resulting string.

PARAMETERS

word *logical by value (required)*
The number to be converted to an ASCII string.

base *integer by value (required)*
One of the following integers indicating octal or decimal conversion:

- 8 Octal conversion (pads with zeros).
- 10 Decimal conversion (left-justified).
- 10 Decimal conversion (right-justified).

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

string *byte array (required)*
A byte array into which the converted value of *word* is placed. This array must be long enough to contain the result. No result, however, will exceed six characters. For octal conversion (*base* = 8), six characters, including leading zeros, are always returned in *string*. In octal conversions, the length returned by ASCII is the number of significant (right-justified) characters in *string* (excluding leading zeros). If *word* = 0, the length (*numchar*) returned by ASCII is 1.

For decimal conversions, *word* is considered as a 16-bit, two's complement integer ranging from -32768 to +32767. If the value of *word* = 0, only one zero character is returned in *string*. The length (*numchar*) returned by ASCII is the total number of characters in *string* (including the sign). If *word* = 0, the length returned by ASCII is 1.

For decimal left-justified conversions (*base* = 10), leading zeros are removed and the numeric ASCII result is left-justified in *string*. Thus, the most significant digit (or the "-" sign) is in *string*(0), the next most significant digit is in *string*(1), and so on.

For decimal right-justified conversions (*base* = -10), the result is right-justified in *string*. Thus, the least significant digit is in *string*(0), the next least significant digit is in *string*(-1), and so on.

For right-justified conversions, the byte array into which the converted value is to be placed must specify the right-most byte into which data is to go. For example, if *string* is a 10-byte array declared as:

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

Then it must be specified in the ASCII intrinsic call as follows (for right justification):

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

The result will be right-justified in *string*, with the right-most digit of the result contained in the last (right-most) byte of *string*.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Converting Numbers from Binary Code to ASCII Strings" in Section V.

BEGINLOG

NO INTRINSIC NUMBER ASSIGNED

Marks the beginning of a user logging transaction.

SYNTAX

D LA I I I

BEGINLOG(*index*,*data*,*len*,*mode*,*status*);

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

PARAMETERS

- | | |
|----------------------|--|
| <i>index</i> | <i>double (required)</i>
The parameter returned from OPENLOG that identifies the user's access to the logging system. |
| <i>data</i> | <i>logical array (required)</i>
An array in which the actual information to be logged is passed. A log record contains 128 words of which 119 words are available to the user. Thus, the most efficient use of logging file space is to structure arrays with lengths in multiples of 119 words. |
| <i>len</i> | <i>integer (required)</i>
The length of the data in <i>data</i> . A positive integer indicates words, and a negative integer indicates bytes. If the length is greater than 119 words (or 238 bytes), the information in data will be divided into two or more physical log records. |
| <i>mode</i> | <i>integer (required)</i>
An integer which specifies whether you want your process impeded by the logging process if the logging buffer is full. If it is not possible to log the transaction and the mode is set to NOWAIT, the BEGINLOG intrinsic will return an indication via <i>status</i> that the request was not completed. The following integers are valid for this parameter:

0 Specifies WAIT.

1 Specifies NOWAIT. |
| <i>status</i> | <i>integer (required)</i>
One of the following integers that the logging system uses to return information on the status of the intrinsic call to the user: |

Message No.	Meaning
0	No error occurred for this call.
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. 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 is out of disc space, logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user logging file.
16	Invalid logging identifier.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capabilities required.

ADDITIONAL DISCUSSION

"User Logging" in Section III.

BINARY

INTRINSIC NUMBER 62

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

SYNTAX

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

FUNCTIONAL RETURN

bineqv

integer

The binary equivalent of the numeric string.

PARAMETERS

string

byte array (required)

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

length

integer by value (required)

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

CONDITION CODES

CCE

Successful conversion. A one-word binary value is returned to the user's process.

CCG

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

CCL

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

ADDITIONAL DISCUSSION

"Converting Numbers from an ASCII Numeric String to a Binary Coded Value" in Section V.

Returns the calendar date.

SYNTAX

^L
date := CALENDAR;

FUNCTIONAL RETURN

date

logical

The calendar date is returned in the format:

Bits (7:9) - The day of the year.

Bits (0:7) - The year of the century.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Obtaining the Calendar Date" in Section V.

CAUSEBREAK

INTRINSIC NUMBER 56

Places a session in **BREAK** mode.

SYNTAX

```
CAUSEBREAK;
```

Using the **CAUSEBREAK** intrinsic will interrupt an interactive session. While not applicable in jobs, the **CAUSEBREAK** intrinsic is the programmatic equivalent to pressing **(BREAK)** in a session. Execution of the process can be resumed where the interruption occurred by entering the command:

:RESUME

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Requesting a Process Break" in Section V.

Deletes inactive entries from a USL file.

SYNTAX

```
I  
filenum := IVCLEANUSL(BAuslfnum, filename);
```

CLEANUSL deletes all inactive entries from currently managed USL files and returns the file number of the new USL file. The condition code, therefore, must be tested immediately on return from the intrinsic. Unpredictable results occur if an error number is used as a file number.

FUNCTIONAL RETURN

filenum

integer

The new file number. If an error occurs, *filenum* is one of the following error numbers:

Error No.	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)

A word supplying the file number of the file.

filename

byte array (required)

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

CONDITION CODES

CCE	Request granted. The new file number is returned.
CCG	Not returned by this intrinsic.
CCL	Request denied. An error number is returned to <i>filenum</i> .

ADDITIONAL DISCUSSION

MPE Segmenter Reference Manual (30000-90011).

Returns the actual time according to the system timer.

SYNTAX

^D
time := CLOCK;

FUNCTIONAL RETURN

time

double

A double-word containing the actual time, as monitored by the system timer. The first word contains the hour of the day and the minute of the hour; the second word contains seconds and tenths of seconds as follows:

Word 1:

Bits (8:8) - The minute of the hour.

Bits (0:8) - The hour of the day.

Word 2:

Bits (8:8) - The tenths of seconds.

Bits (0:8) - The seconds.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Time and Date Intrinsics" in Section V.

Closes access to the user logging facility.

SYNTAX

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

Effective with the G.02.00 version of MPE, the number of users and log entries are independent of the number of times the OPENLOG/CLOSELOG intrinsics are called within an application. A logging buffer entry is obtained and the user count is incremented only if this is the first OPENLOG call for this user. A counter is used to keep track of the number of times a user has called OPENLOG and CLOSELOG. The counter is incremented for every OPENLOG and decremented for every CLOSELOG. This is done to ensure the entry in LOGBUFF is released only if this is the last CLOSELOG call for this user (i.e. counter=0).

PARAMETERS

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

mode *integer (required)*
An integer used to indicate whether your process should be suspended if your request for service cannot be completed immediately. The following integers are valid for this parameter:

0 Specifies WAIT.

1 Specifies NOWAIT.

status *integer (required)*
An integer which indicates one of the following status conditions:

Message No.	Meaning
0	No error occurred for this call.
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. 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 is out of disc space, logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user logging file.
16	Invalid logging identifier.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capabilities required.

ADDITIONAL DISCUSSION

"User Logging" in Section III.

COMMAND

INTRINSIC NUMBER 68

Executes an MPE command programmatically.

SYNTAX

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

User-defined commands (UDCs) cannot be used with the `COMMAND` intrinsic.

PARAMETERS

comimage

byte array (required)

Contains an ASCII string of not more than 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. No prompt character, however, should be included in this string. The *comimage* array will be altered by the `COMMAND` intrinsic (lowercase characters in it will be shifted to uppercase), and will be returned to the caller in uppercase.

error

integer (required)

A word to which any error code set by the command is returned. This is the same error code that would appear on a job/session list device if the command was part of an input stream, i.e. a Command Interpreter rather than file system error code. If no error occurs, *error* returns zero. If an MPE warning was detected, a negative number is returned (for example, "-383" is returned if `CIWARN 383` is detected).

param

integer (required)

A word to which the number (index) of the erroneous parameter is returned. If no parameters are in error, *param* returns zero. If there are errors, *param* may be zero or some positive integer. In the case where an error refers to a file system problem, *param* is the file system error code.

CONDITION CODES

CCE

Request granted although a Command Interpreter warning may have been detected.

CCG

An executor-dependent error, such as an erroneous parameter, prevented execution of the command. The numeric error code is contained in *error*. The command with the error terminated before executing completely.

CCL

Request denied. The command was an undefined command.

ADDITIONAL DISCUSSION

"Executing Commands Programmatically" in Section V.

Creates a process.

SYNTAX

```

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

If a running process, has Process Handling capability, it can request the creation of a son process by calling 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 the new Process Identification Number (PIN) to the requesting process. The CREATE intrinsic will achieve only partial completion if the circumstances described below exist.

The process is not created and a PIN of zero is returned if one of the following conditions exist:

- The *stacksize* is less than 512 (decimal) and is not -1.
- The *dlsiz*e is less than 0 and is not -1.
- The *maxdata* is less than or equal to 0, and is not -1.
- The stack space required exceeds *maxdata*. The *dlsiz*e may have been modified by the intrinsic to satisfy Condition 2 under CCG. The DB global size value is the sum of the primary DB plus the secondary DB values (found on the compiler listing) or the total DB given at program preparation time by the program map (PMAP).
- An illegal value (a nonexistent subqueue) was specified for the *priorityclass* parameter.

The process is not created and the PIN is unchanged if one of the following conditions exist:

- The program file of the creating process does not have Process Handling (PH) capability.
- A required parameter (*progrname* or *pin*) is omitted.
- A reference parameter was 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 system configuration. The *dlsiz*e may have been modified to satisfy Condition 2 under CCG.

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

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

PARAMETERS

progrname

byte array (required)

Contains a string, terminated by a blank, specifying the name, and optionally, the account and group of the file containing the program to be run. If the program has a lockword, the lockword must be specified.

entryname

byte array (optional)

Contains a string, terminated by a blank, specifying the declared entry point (label) in the program where execution is to begin when the process is activated. The primary entry point in the program can be specified by setting the array equal to a blank character alone.

Default: The primary entry point is used.

pin

integer (required)

A word in which the PIN of the new process is returned to the requesting process. This PIN is used in other intrinsics to reference the new process. The PIN can range from 1 to 1024. If an error is detected, a PIN of zero is returned to the requesting process.

param

integer by value (optional)

A word used to transfer control information to the new process. Any instruction in the outer block of code in the new process can access this information in location Q-4 of the stack.

Default: Word is filled with zeros.

flags

logical by value (optional)

A word whose bits specify the loading options:

Bit (15:1) - ACTIVE bit.

=0 The calling process is not activated when the new process terminates.

=1 MPE reactivates the calling process (father) when the new process terminates.

Default: (15:1)=0

Bit (14:1) - LOADMAP bit.

=0 No map is produced.

=1 A listing of the allocated (loaded) program is produced on the job/session list device. This map shows the Code Segment Table (CST) entries used by the new process.

Default: (14:1)=0

Bit (13:1) - DEBUG bit.

This bit is ignored if the user is nonprivileged and the new process requires Privileged Mode. It is also ignored if the user does not have read/write access to the program file of the new process.

=0 The breakpoint is not set.

=1 DEBUG is called at the first executable instruction of the new process.

Default: (13:1)=0

Bit (12:1) - NOPRIV bit.

=0 The program is loaded in the mode specified when the program file was prepared.

=1 The program is loaded in non-Privileged Mode.

Default: (12:1)=0

Bits (10:2) - LIBSEARCH bits.

These bits denote which libraries are to be searched for the program.

=00 Search the System Library.

=01 Search the Account Public Library, followed by System Library.

=10 Search the Group Library first, then the Account Public Library, and finally the System Library.

Default: (10:2)=00

Bit (9:1) - NOCB bit.

If you are using a large stack set (9:1)=1.

=0 Control blocks may be established in the Process Control Block Extension (PCBX) area.

=1 The file system control blocks are established in an extra data segment.

Default: (9:1)=0

Bits (7:2) – These bits are reserved for MPE and should be set to zero.

Bits (5:2) – STACKDUMP bits.

These bits control the enabling/disabling of the mechanism by which the stack is dumped in the event of an abort.

=00 Stackdump mechanism enabled only at father level.

=01 Stackdump mechanism is enabled unconditionally.

=10 Same as (5:2)=00.

=11 Stackdump mechanism disabled unconditionally for new process.

Default: (5:2)=00

Bit (4:1) – Reserved for MPE and should be set to zero.

Bit (3:1) – DL to QI bit. (This bit is used only when bits (5:2)=01.)

=0 This portion of the stack will not be dumped.

=1 Causes the portion of the stack from DL to QI to be dumped.

Default: (3:1)=0

Bit (0:3) – Reserved for MPE and used only when (5:2)=01.

stacksize

integer by value (optional)

An integer denoting the number of words assigned to the local stack area bounded by the initial Q and Z registers. A value of -1 indicates that the MPE Segmenter is to assign default values. Specifying -1 is equivalent to omitting the parameter.

Default: The same as that specified in the program file.

dlsize

integer by value (optional)

An integer denoting the number of words in the user-managed stack area bounded by the DL and DB registers. A value of -1 indicates that the MPE Segmenter is to assign default values. This is equivalent to omitting the parameter.

Default: The same as that specified in the program file.

maxdata

integer by value (optional)

The maximum size allowed for the process stack in words. When specified, this value overrides the one established at program preparation time. A value of -1 indicates that the MPE Segmenter is to assign default values.

Default: If not specified in either the intrinsic call or the program file, MPE assumes that the stack will remain the same size.

priorityclass

logical by value (optional)

A string of two ASCII characters describing the priority class in which the new process is scheduled. Standard users (nonprivileged) can reschedule processes into any of the five subqueues except the AS queue. However this function is limited by the maximum priority assigned to the account by the System Manager. Users who have Privileged Mode capability can schedule processes into the AS subqueue, as well as any of the others. A process in the linear queue AS or BS will not give up CPU voluntarily. Therefore, it could loop infinitely and prevent other processes from accessing the CPU.

Default: The same as the priority of the calling process.

rank

integer by value (optional)

This parameter is used only for backward-compatibility with pre-MPE IV operating systems. This parameter is ignored on versions G.00.00 and later.

CONDITION CODES

CCE

Request granted. The new process has been created.

CCG

Request granted. The *maxdata* and/or *dsize* parameters given were illegal, but other values were assigned as follows:

1. If the *maxdata* specified exceeds the maximum Z-DL allowed by the configuration, the configured maximum value is assigned.
2. If the area from Base to DB is not zero, then the DL area pads this area to round it up to 128.

CCL

Request denied because the *progrname* or *entryname* specified does not exist.

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Creating an Extra Data Segment" and "Creating and Activating Processes" in Section III.

CREATEPROCESS

INTRINSIC NUMBER 101

Creates a process and can assign \$STDIN and \$STDLIST to any file.

SYNTAX

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

CREATEPROCESS entails a superset of the CREATE intrinsic and is designed to be more flexible and extendable than CREATE. The CREATEPROCESS intrinsic not only creates a process, it also allows you to assign the system-defined files, \$STDIN and \$STDLIST, to any file when the process is created. You are not limited to system-defined defaults for \$STDIN and \$STDLIST. If the intrinsic is called with the *error* parameter omitted, an invalid address for parameter *error* is returned. In split-stack mode, the calling process will be aborted.

PARAMETERS

error

integer (required)

An integer indicating type of success or failure. A minus sign (-) preceding the returned value indicates that the associated message is a warning. An error terminates the intrinsic call while a warning means the call will continue. The Job Control Word (JCW) should be checked before continuing. The following is a list of the possible values returned:

Message No.	Meaning
0	Process created as requested.
1	Caller lacks Process Handling (PH) capability.
2	Required parameter (other than <i>error</i>) omitted.
3	Parameter address (other than <i>error</i>) out of bounds.
4	Out of system resources (e.g. PCBs, DSTs).
5	Process not created because an invalid <i>itemnums</i> was specified.
6	Unable to create process because <i>progrname</i> does not exist.
7	Unable to create process because entry name does not exist or is invalid.
8	Process not created because <i>progrname</i> is invalid.
-9	Process created with default stacksize from the program file (specified stacksize was < 512).

- 10 Process was created with default *dlsiz*e from the program file (specified *dlsiz*e was < 0).
- 11 Process created with default *maxdata* from the program file (specified *maxdata* was < 0).
- 12 Process created with *dlsiz*e rounded up to next 128-word multiple.
- 13 Process created with *maxdata* decreased to configuration maximum.
- 14 Process created with *maxdata* increased to *dlsiz*e+*glob-size*+*stacksize* (*globsize* is defined to be primary DB space+secondary DB space).
- 15 Process not created because *dlsiz*e+*globsize*+*stacksize* was > configured maximum stacksize.
- 16 Process not created because "hard" load error occurred (for example, I/O error reading the program file).
- 17 Process not created because an illegal value was specified for priority class.
- 18 Unable to create process because specified \$STDIN could not be opened.
- 19 Unable to create process because \$STDLIST could not be opened.
- 20 Process not created because string to be passed to new process was invalid (pointer without length, length without pointer, or length exceeds stack size of calling process).

pin

integer (required)

An integer in which the PIN of the newly created process is returned. If there is an error in creating the new process, i.e. parameter *error* > 0, a zero is returned.

programe

byte array (required)

A byte array containing a string, terminated by any nonalphanumeric character other than a period (.) or a slash (/), which specifies the name of the program file to be run by the new process.

itemnums

integer array (optional)

An array containing the item numbers (in any order) of the options you want to use in creating a new process. This array must contain a zero as its last element to indicate the end of the option list. (Refer to Table 2-1.)

items

logical array (optional)

An array containing the items (in the same order as the item numbers in *itemnums*), to be used in creating the new process. (Refer to Table 2-1.)

Table 2-1. Item Values Returned by CREATEPROCESS

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

Itemnumber	Items
0	Indicates the end of the option list.
1	A pointer to a byte array containing the name of the entry point in the program where the new process is to begin execution. The name is specified as a string of characters terminated by a blank.
2	An integer containing a parameter to be passed to the new process (accessed through location Q-4 of the outer block).
3	A logical value containing the load option flags to be used in loading the program file for the new process. This parameter has the same definition as the <i>flags</i> parameter of the CREATE intrinsic.
4	An integer specifying the initial stack size (Q-Z).
5	An integer specifying the initial <i>dsize</i> (DL-DB) for the new process.
6	An integer specifying the maximum stack size for the new process (i.e. <i>maxdata</i>).
7	A string of two ASCII characters specifying the priority class in which the new process is to be scheduled (AS, BS, CS, DS, or ES). Selecting AS or BS can cause performance degradation since the process will wholly own the CPU.
8	A pointer to a byte array containing the definition of a file to be used as \$STDIN for the new process. (See Notes 1 & 2.)
9	A pointer to a byte array containing the definition of a file to be used as \$STDLIST for the new process. (See Notes 1 & 3.)
10	A logical value indicating that the process has suspended and its anticipated source of reactivation. Specification of this parameter causes the newly created process to be activated immediately. The meanings of the individual bit fields of this parameter are as follows: <div style="margin-left: 40px;"> <p>Bit (15:1) Father activation bit. =1 The process expects to be activated by its father. =0 The process does not expect to be activated by its father.</p> <p>Bit (14:1) Son activation bit. =1 The process expects to be activated by one of its sons. =0 The process does not expect to be activated by one of its sons.</p> </div>
11	A pointer to a byte array containing a string of information to be passed to the new process. The length of the string is specified in Item Number 12. (See Note 4.)
12	An integer specifying the length in bytes of the string specified with Item Number 11. (See Note 4.)

Table 2-1. Item Values Returned by CREATEPROCESS (Continued)

NOTES

1. If Item Numbers 8 or 9 are not specified, the default \$STDIN and \$STDLIST will be used in creating the new process. These defaults are the current \$STDIN and \$STDLIST files for the creating (father) process.
2. Item Number 8 indicates that the corresponding item in the item array is the address of a byte array which contains the definition of the file to be used as \$STDIN for the new process. This byte array must contain an ASCII string (terminated by a carriage return) which is the right-hand side of a file equation specifying the file to be used as \$STDIN (i.e. everything after the ":FILE *formaldesignator*=" portion of the file equation).
3. Item Number 9 indicates that the corresponding item in the item array is the address of a byte array which contains the definition of the file to be used as \$STDLIST for the new process. This array is defined as above for \$STDIN.
4. Item Numbers 11 and 12 indicate that a string is to be passed to the new process. This string will be placed just after the global area of the new process stack. A DB relative value byte pointer to the string in the new process stack will be placed at the Q-5 of the stack (where Q is the initial value of the Q-register at activation time) and the length of the string in bytes will be placed at Q-6. If no string is specified to be passed to the new process, Q-5 and Q-6 will both contain 0.

CONDITION CODES

CCE	No error.
CCL	Unsuccessful.
CCG	Successful call, however a warning might have been returned to <i>error</i> .

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Creating and Activating Processes" in Section III.

CTranslate

INTRINSIC NUMBER 61

Converts a string of characters between EBCDIC and ASCII, or between EBCDIK and JIS (KANJI).

SYNTAX

0-VIVBABAIVBA

```
CTranslate(code,instring,outstring,stringlength,table);
```

The CTranslate intrinsic is used for character code translating. Translation can occur between the standard computer character codes or with an optional user-defined code permitting you to obtain character code conversions within programs of your own design.

EBCDIK is a Hewlett-Packard-specific (Japanese) version of EBCDIC.

KANJI is an 8-bit JIS (Japanese International Standard, JISCII, a Japanese version of USASCII) code.

PARAMETERS

code

integer by value (required)

An integer identifying a specific translation to be used as follows:

- 0 The user-supplied table specified in the *table* parameter.
- 1 EBCDIC to ASCII.
- 2 ASCII to EBCDIC.
- 3 Reserved for future use.
- 4 Reserved for future use.
- 5 EBCDIK to JIS (KANJI).
- 6 JIS (KANJI) to EBCDIK.

instring

byte array (required)

The string of characters to be translated.

outstring

byte array (optional)

A byte array to which the translated character string is returned. If an *outstring* is not specified, all translation will occur within *instring*. The *instring* and *outstring* parameters may specify the same array.

stringlength

integer by value (required)

A positive integer specifying the length (in bytes) of *instring*.

table

byte array (required when code=0, otherwise optional)

A byte array to be used as the translation table. The contents of *table*, and the order of these contents, define the translation process. The length of *table* may be as large as 256 bytes, but it needs to be only as large as the largest numeric value of any source byte in *instr*. The *table* array is constructed such that each byte in *table* corresponds to a byte value in the source string to be translated. For example, the first byte in *table* gives the code to be substituted for source bytes whose value is 0.

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Translating Characters With the CTRANSLATE Intrinsic" in Section V.

DASCII

INTRINSIC NUMBER 75

Converts a double-word (32-bit) binary number to an ASCII string.

SYNTAX

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

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

FUNCTIONAL RETURN

numchar *integer*
The number of characters in the resulting string.

PARAMETERS

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

base *integer by value (required)*
One of the following integers indicating octal or decimal conversion:

8 Octal conversion.

10 Decimal conversion (left-justified).

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

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

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

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

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Converting Numbers from Binary Code to ASCII Strings" in Section V.

DATELINE

NO INTRINSIC NUMBER ASSIGNED

Returns the current date and time.

SYNTAX

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

PARAMETERS

datebuf

byte array (required)

A 27-character byte array which contains the date and time information in the format:

FRI, MAY 27, 1983, 12:06 PM

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

None.

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

SYNTAX

D	BA	IV
<i>dval</i> := DBINARY(<i>string</i> , <i>length</i>);		

FUNCTIONAL RETURN

<i>dval</i>	<i>double (optional)</i> The converted double-word binary value.
-------------	---

PARAMETERS

<i>string</i>	<i>byte array (required)</i> Contains the octal or signed decimal number (in ASCII characters) to be converted. If the character string in this array begins with a percent sign (%), it is treated as an octal representation, if the string begins with a plus sign, a minus sign, or a number, it is treated as a decimal representation. Leading blanks are not allowed, and are treated as illegal characters.
<i>length</i>	<i>integer by value (required)</i> An integer representing the length (number of bytes) in the string containing the ASCII-coded value. If the value of <i>length</i> is 0, the intrinsic returns 0 to the calling process. If the value of <i>length</i> is less than 0, the intrinsic causes the user process to abort.

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Converting Numbers from an ASCII Numeric String to a Binary Coded Value" in Section V.

DEBUG

INTRINSIC NUMBER 99

Invokes the DEBUG facility.

SYNTAX

DEBUG;

PARAMETERS

None.

CONDITION CODES

The condition code remains the same.

ADDITIONAL DISCUSSION

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

Expands or contracts the area between DL and DB.

SYNTAX

I IV

dldbsize := DLSIZE(*size*);

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

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

FUNCTIONAL RETURN

dldbsize *integer (optional)*
The size actually granted. This value is a negative quantity except on error condition CCL when it is possible to have a positive value returned.

PARAMETERS

size *integer by value (required)*
A negative integer representing the new *size* of the DL-to-DB area. 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* granted will be an absolute value which is rounded up so that the distance between the beginning of the segment and DB is a multiple of 128 words.

CONDITION CODES

CCE	Request granted. The value returned is at least as large as the value requested.
CCG	Requested size exceeded maximum limit allowed. The maximum limit allowable is granted and its size is returned.
CCL	An illegal <i>size</i> parameter was specified. The <i>size</i> parameter was a positive integer. The original area <i>size</i> , assigned when the stack segment was created, is granted.

ADDITIONAL DISCUSSION

"Changing the DL to DB Area Size" in Section V.

Copies data from an extra data segment into the stack.

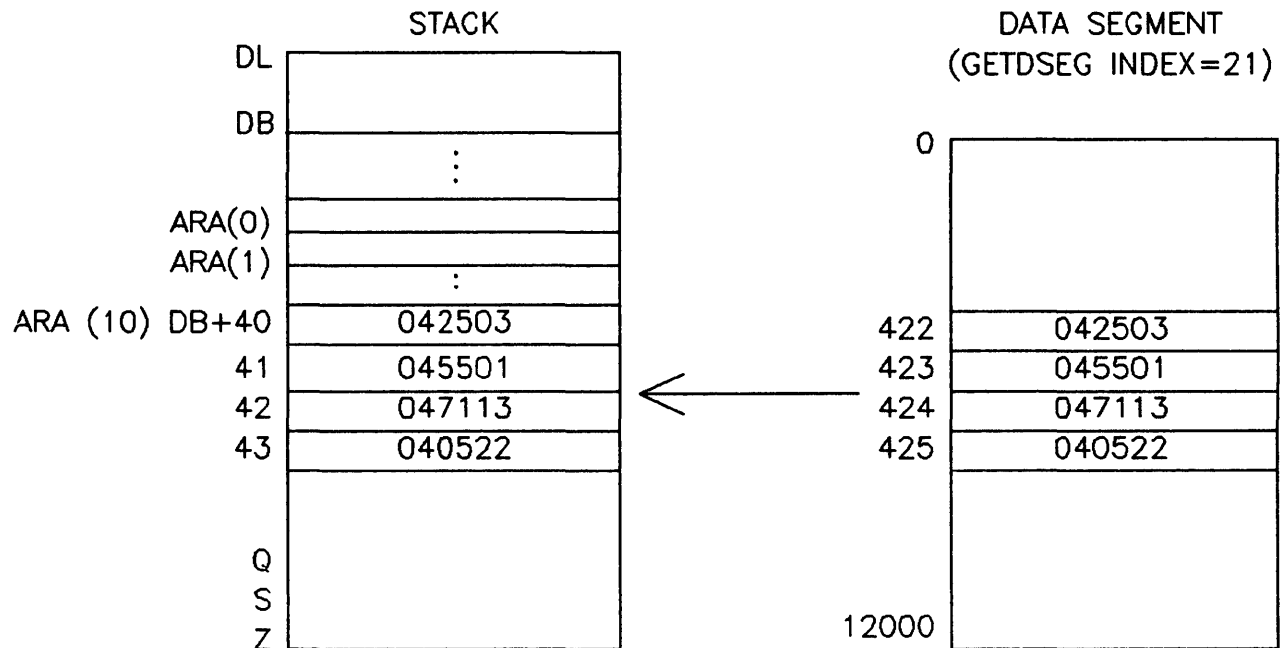
SYNTAX

^{LV} ^{IV} ^{IV} ^{LA}
DMOVIN(*index*,*disp*,*number*,*location*);

A process can copy data from an extra data segment into the stack by issuing the DMOVIN intrinsic call. A bounds check is initiated 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. For example, in the diagram shown below, to move four words from locations 422 through 425 of the data segment whose *index* is 21, to DB+40 through DB+43 of the stack, the intrinsic call would be:

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

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



PARAMETERS

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

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Data Segment Management (DS) capability required.

ADDITIONAL DISCUSSION

The GETDSEG intrinsic in this section, and "Creating an Extra Data Segment" in Section III.

Copies data from the stack to an extra data segment.

SYNTAX

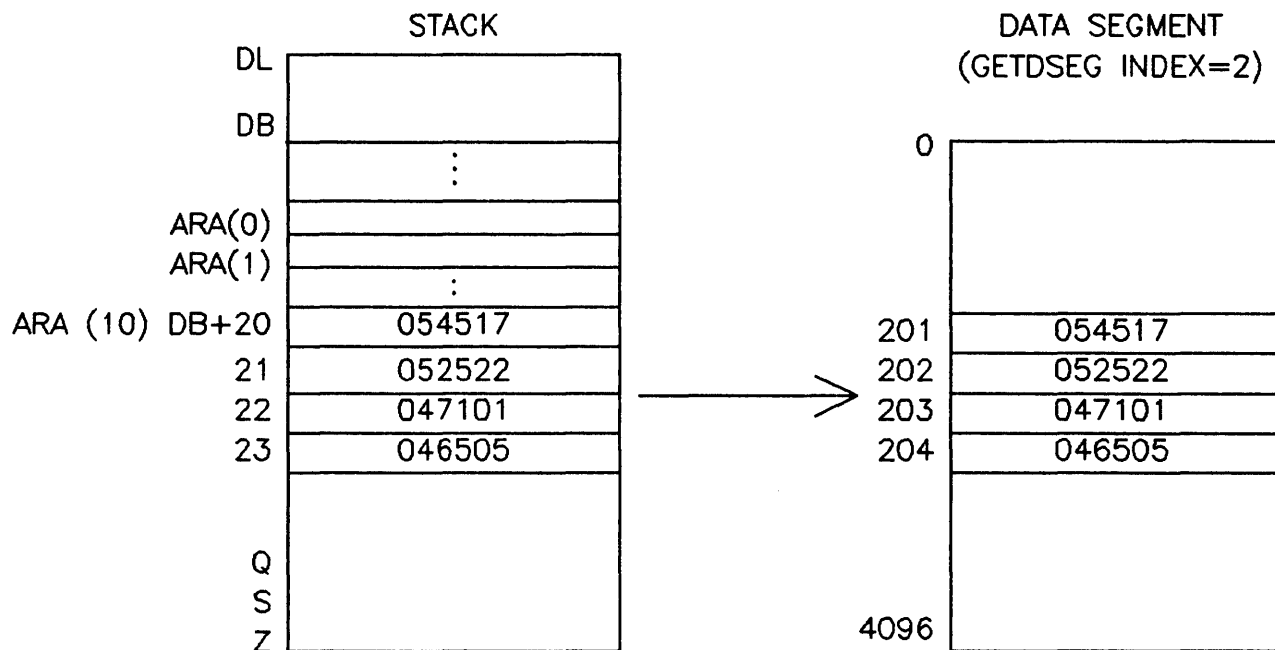
LV IV IV LA

DMOVOUT(*index*,*disp*,*number*,*location*);

The DMOVOUT intrinsic copies data from the stack to an extra data segment. When DMOVOUT is called, a bounds check is initiated to ensure that the data is taken from an area within the stack boundaries and moved to an area within the extra data segment boundaries. For example, in the diagram shown below, to move four words from ARA(10) within the stack 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 (*disp*) within the data segment is 201; the *number* of words to be moved to the data segment is 4; and the *location* of the data within the stack is the address of ARA(10). If ARA(10) is at DB+20, the end result is that the four words within the stack will be moved to words 201 through 204 of the data segment, as shown below:



PARAMETERS

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

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Data Segment Management (DS) capability required.

ADDITIONAL DISCUSSION

The GETDSEG intrinsic in this section, and "Creating an Extra Data Segment" in Section III.

Marks the end of a user logging transaction.

SYNTAX

D LA I I I

ENDLOG(*index*,*data*,*len*,*mode*,*status*);

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

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

PARAMETERS

<i>index</i>	<i>double (required)</i> The parameter returned from OPENLOG that identifies the user's access to the logging file.
<i>data</i>	<i>logical array (required)</i> An array in which the actual information to be logged is passed. A log record contains 128 words of which 119 words are available to the user. Thus, the most efficient use of logging file space is to structure arrays with lengths in multiples of 119 words.
<i>len</i>	<i>integer (required)</i> The length of the data in <i>data</i> . A positive count indicates words and a negative count indicates bytes. If the length is greater than 119 words, the information in data will be divided into two or more physical log records.
<i>mode</i>	<i>integer (required)</i> An integer which specifies whether you want your process impeded by the logging process if the logging buffer is full. If it is not possible to log the transaction and the mode is set to NOWAIT, the ENDLOG intrinsic will return an indication in the status word that the request was not completed: 0 Specifies WAIT. 1 Specifies NOWAIT.
<i>status</i>	<i>integer (required)</i> One of the following integers that the logging system uses to return information on the status of the intrinsic call:

Message No.	Meaning
0	No error occurred for this call.
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. 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 is out of disc space, logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user logging file.
16	Invalid logging identifier.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capabilities required.

ADDITIONAL DISCUSSION

"User Logging" in Section III.

Changes length of a USL file.

SYNTAX

I	IV	IV
<i>filenum</i>	=	EXPANDUSLF(<i>uslfnum</i> , <i>records</i>);

You can increase or decrease the length of a USL file by calling the EXPANDUSLF intrinsic. When this intrinsic is executed, a new USL file is created with a *records* 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.

FUNCTIONAL RETURN

filenum

integer (optional)

The new file number. If an error occurs, the error number is returned instead of the new file number; the condition code must be tested immediately on return from this intrinsic. If an error number were to be used as a file number, unpredictable results would occur. The following lists the error numbers and their associated meanings:

Error No.	Meaning
0	The 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 input/output error occurred. This can occur on the old USL file, or the new <i>uslfnum</i> to which the intrinsic is copying the information.
7	The intrinsic was unable to open the new USL file.
8	The intrinsic was unable to close (purge) the old USL file.
9	The intrinsic was unable to close (save) the new USL file.
10	The intrinsic was unable to close \$NEWPASS.
11	The intrinsic was unable to open \$OLDPASS.

PARAMETERS

uslfrum

integer by value (required)

A word supplying the file number of the file.

records

integer by value (required)

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

CCL

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

ADDITIONAL DISCUSSION

MPE Segmenter Reference Manual (30000-90011).

Requests the Process Identification Number (PIN) of its father process.

SYNTAX

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

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

FUNCTIONAL RETURN

pin

integer

An integer containing the Process Identification Number (PIN) for the father of the process.

CONDITION CODES

CCE

Request granted. The father is a user process.

CCG

Request granted. The father is a job or session main process.

CCL

Request granted. The father is a system process.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Determining Father Process" in Section III.

FCARD

NO INTRINSIC NUMBER ASSIGNED

Drives the HP 7260A Optical Mark Reader (OMR).

SYNTAX

I I IA I I

FCARD(*recode*,*filenum*,*bufadr*,*count*,*status*);

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.

PARAMETERS

recode

integer (required)

A positive integer represented as an input or output parameter.

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

- 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* is described below:

- 0 Indicates that the request, i.e. the call to FCARD, has been successfully performed. For the following conditions, when output *recode=0*, the specified parameters are significant to the program:
 - a. If the request was to open a file (*recode=0*), then *filenum* is significant.
 - b. If the request was either to read (*recode=1*) or to retransmit (*recode=3*), then *bufadr* (the first byte may contain status information identical to that contained in the parameter *status*), *count*, *filenum*, and *status* are significant.
 - c. If the request was to select the previously read card (*recode=2*), then *status* is significant.
 - d. If the request was to perform a temporary suspension of the program (*recode=4*), then *status* is significant.
 - e. For all other requests (*recode=10, 11, 12, 17, 18, and 20*), none of the other parameters are significant.
- 1 Indicates that *recode* specified in the request was not one of the following legal values: 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*).

filenum

integer (required)

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 *recode=0*. It must be provided to FCARD on all requests.

bufadr

integer array (required)

The array to which the record is to be transferred. The declared length of *bufadr* should be set to 120 words.

count

integer (required)

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

status

integer (required)

An integer indicating whether the HP 7260A has successfully performed or responded to the read, select, retransmit, or temporary suspend request. If *status* is equal to zero, then the request has been successfully performed. If *status* is not equal to zero, then it contains an octal value specifying the HP 7260A condition. The octal values indicate the following conditions:

- | | |
|----------|--|
| OCTAL 07 | Input hopper empty or hopper full. Can either be returned upon a read request (<i>recode=1</i>) or upon a retransmit request, if there is no data to retransmit (<i>recode=3</i>). |
| OCTAL 11 | Pick fail. Can either be returned upon a read request (<i>recode=1</i>) or upon a retransmit request, if there is no data to transmit (<i>recode=3</i>). |
| OCTAL 13 | Select hopper full. Indicates that the OMR select hopper was full when the select request (<i>recode=2</i>) was issued. |

OCTAL 14	Successful select. Indicates that the HP 7260A has successfully selected the card upon the select request (<i>recode</i> =2).
OCTAL 22	READY <i>status</i> . Indicates that the HP 7260A READY push button has been pressed (<i>recode</i> =4). Would also indicate that the HP 7260A is ready but there is no data to be retransmitted input (<i>recode</i> =3).
OCTAL 37	Not ready. Can either be returned upon a read request (<i>recode</i> =1) or upon a retransmit request (<i>recode</i> =3). This <i>status</i> 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.

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

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

CONDITION CODE

The condition code remains unchanged.

ADDITIONAL DISCUSSION

Appendix B, "DEVICE CHARACTERISTICS".

FCHECK

INTRINSIC NUMBER 10

Requests details about file input/output errors.

SYNTAX

O-VIVIIDI

```
FCHECK(filenum,errorcode,tlog,blknum,numrecs);
```

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

FCHECK accepts zero as a legal *filenum* parameter value. When zero is specified, the information returned in *errorcode* reflects the status of the last call to FOPEN. When FOPEN fails, there is no file number which can be referenced in *filenum*. Therefore when FOPEN fails, a *filenum* of zero can be used in the FCHECK intrinsic call to obtain the *errorcode* only.

PARAMETERS

filenum

integer by value (optional)

A word supplying the file number of the file for which error information is to be returned. If omitted or 0, FCHECK assumes you want the last FOPEN error.

errorcode

integer (optional)

A word to which a 16-bit code specifying the type of error that occurred is returned. If the previous operation was successful or an EOF was encountered, all bits are set to zero. (Refer to Table 2-2 for a listing of File System Error Codes.)

Default: The error code is not returned.

tlog

integer (optional)

A word to which the transmission log value recorded on the last data transfer is returned. This word specifies the number of words actually read or written if an input/output error occurred.

Default: The transmission log value is not returned.

blknum

double (optional)

The physical record count if the file is not a spoolfile; or the logical record count if the file is a spoolfile. The physical count is the number of physical records transferred to or from the file since either FOPEN, for fixed and undefined length record files; or the last rewind, rewind/unload, space forward or backward to tape mark, for variable length record files.

numrecs

integer (optional)

A word to which the number of logical records in the bad block (blocking factor) is returned.

Default: The number of logical records is not returned.

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	Disc space allocation disabled on all discs 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 disc address.
37	Invalid memory address.
38	Tape parity error.
39	Recovered tape error.
40	Operation inconsistent with access type.
41	Operation inconsistent with record type.
42	Operation inconsistent with device type.
43	Write exceeds record size.
44	Update at record zero.
45	Privileged file violation.
46	Out of disc space.
47	Input/output error on a file label.

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

Code (Decimal)	Meaning
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 disc space.
62	Out of account disc 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.
91	Exclusive violation: file accessed exclusively.
92	Lockword violation.
93	Security violation.
94	User is not creator.

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

Code (Decimal)	Meaning
95	Read not completed due to BREAK.
96	Disc 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").
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 disc.
138	Track does not exist on foreign disc.
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.

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

Code (Decimal)	Meaning
152	Unrecognized key word in FOPEN <i>device</i> parameter.
153	Expecting ";" or "carriage return" in FOPEN <i>device</i> parameter.
154	Environment file open error.
155	File not environment file. Check file code or record size.
156	Header record incorrect.
157	Uncompiled environment file.
158	Error reading environment file.
159	Error closing environment file.
160	Error doing FDEVICECONTROL from environment file.
161	Too many parameters in device string overflow.
162	Expecting "=" after keyword in <i>device</i> parameter.
163	"ENV" backreference in file equation incorrect.
164	The <i>device</i> parameter too large or missing carriage return.
165	Invalid density specification.
166	FFILEINFO failed in accessing remote spoolfile.
167	Spoolfile label error, can't insert environment file name.
168	Item not supported on remote system.
170	The record is marked deleted. FPOINT positioned pointer to a record that was marked for deletion.
171	Duplicate key value (KSAM error).
172	No such key (KSAM error).
173	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 handle the file built by previous version.
181	Invalid key starting position (KSAM error).
182	File is empty (KSAM error).
183	Record does not contain all keys (KSAM error).
184	Invalid record number (KSAM FFINDN intrinsic error).
185	Sequence error in primary key (KSAM error).
186	Invalid key length (KSAM error).
187	Invalid key specification (KSAM error).
188	Invalid device specification (KSAM error).
189	Invalid record format (KSAM error).
190	Invalid key blocking factor value (KSAM error).
191	Record does not contain search key for deletion. Specified key value points to record which does not contain that value (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).

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

Code (Decimal)	Meaning
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 IDWAIT is not a DS line number.
214	The requested DS line has not been opened with a user :DSL INE 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.
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.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

None.

Closes a file.

SYNTAX

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

The FCLOSE intrinsic terminates access to a file. This intrinsic applies to files on all devices. FCLOSE deletes buffers and control blocks through which the user process accessed the file. It also deallocates the device on which the file resides and it may change the *disposition* of the file. If you do not issue FCLOSE calls for all files opened by your process, such calls are issued automatically by MPE when the process terminates. All magnetic tape files are left offline after such FCLOSE calls, to indicate to the System Operator that they may be removed.

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

If an 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 to occur before the rewind, and only if the tape was written on. (This feature applies to version G.01.00 or later only.)

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

PARAMETERS

filenum

integer by value (required)

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

disposition

integer by value (required)

Indicates the disposition of the file, significant only for files on disc and magnetic tape (*disposition* is ignored by the Foreign Disc Facility). This *disposition* can be overridden by a corresponding parameter in a :FILE command entered prior to program execution. The *disposition* options are defined by the bit fields (13:3) and (12:1) as follows:

(13:3) – Domain Disposition.

=000 No change. The *disposition* remains as it was before the file was opened. Thus, if the file is new, it is deleted by FCLOSE; otherwise, the file is assigned to the domain it belonged to previously. An unlabeled tape file is rewound. If the file resides on a labeled tape, the tape is rewound and unloaded.

- =001 Permanent file. If the file is a disc file, it is saved in the system file domain. A new or old temporary file on disc will have an entry created for it in the system file directory. Should a file of the same name already exist in the directory, an error code is returned and the file remains open. If the file is an old permanent file on disc, this domain disposition has no effect. Also, if the file is stored on magnetic tape, the tape is rewound and unloaded.

- =010 Temporary job file (rewound). The file is retained in the user's temporary (job/session) file domain and can be requested by any process within the job/session. If the file is a disc file, the uniqueness of the file name is checked. Should a file of the same name already exist in the temporary file domain, an error code is returned and the file remains open. When a file resides on unlabeled magnetic tape, the tape is rewound. However, if the file resides on labeled magnetic tape, the tape is backspaced to the beginning of the presently opened file.

- =011 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 Released file. The file is deleted from the system.

(12:1) - Disc Space Disposition (for fixed, undefined, and variable format files).

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

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

Bit (0:12) - Reserved for MPE and should be set to zero.

When a file is opened by the FOPEN intrinsic, a file count (maintained by MPE for each file) is incremented by one. When the 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. Then the effective (saved) *disposition* is the smallest nonzero *disposition* parameter specified among 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 FCLOSE *disposition* is %14, and the third (and last) FCLOSE *disposition* is %12. The final *disposition* on the file XYZ will be *disposition* 1 (permanent file and no return of disc space).

seccode

integer by value (required)

Denotes the type of security initially applied to the file. This is significant only for new permanent files (*seccode* is ignored by the Foreign Disc Facility). The options are:

- 0 Unrestricted access. The file can be accessed by any user, unless prohibited by current MPE provisions.
- 1 Private file creator security. The file can only be accessed by its creator.

CONDITION CODES

CCE The file was closed successfully.

CCG Not returned by this intrinsic.

CCL The file was not closed because an incorrect *filenum* was specified or because another file with the same name and *disposition* exists in the system. Any outstanding write I/Os which failed will also cause the FCLOSE to fail (such I/Os as buffered writes which are done in background). Additionally, an illegal *disposition* (5, 6, or 7) may have been specified. This can be detected by FCHECK returning an *error* of 49.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Closing Files", "File Domains", and "Internal Operations for File Accessing" in Section IV.

For further information on magnetic tape files and associated functions, refer to the MPE File System Reference Manual (30000-90236).

FCONTROL

INTRINSIC NUMBER 13

Performs control operations on a file or device.

SYNTAX

IV IV L

FCONTROL(*filenum*,*controlcode*,*param*);

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

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

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

PARAMETERS

filenum

integer by value (required)

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

controlcode

integer by value (required).

An integer specifying one of the following operations to be performed:

- 0 General device control. The *param* parameter is transmitted to the appropriate device driver. The value returned is transmitted to the user through *param*.
- 1 Line control. A request to send the value specified in *param* to the terminal or line printer driver as a carriage control directive. For non-spoiled devices only, use line controls provided by FWRITE when directing to a disc or a spoiled file.

- 2 Complete input/output. This ensures that requested input/output has been physically completed. Valid only for buffered files. Posts the block being written whether full or not. This control code is ignored for message files.
- 3 Read hardware status word. This operation returns the status word from the device on which the file resides to *param*. The returned value is the status of the device from the previous input/output operation, including FOPEN of the file.
- 4 Set time-out interval. This code indicates that a time-out interval is to be applied to input from 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 itself is specified, in seconds, in a word on the user's stack, indicated by *param*. If this interval is zero, any previously established interval is cancelled, and no time-out occurs. A *controlcode* 4 only affects the next read and is ignored if the addressed file is not being read from the terminal. This code may also be applied to an Interprocess Communication (IPC) file. In this case, *param* specifies the length of time that a process will wait when reading from an empty file or writing to a full one. For IPC files, the time-out will remain enabled until it is explicitly cancelled. Refer to the MPE File System Reference Manual (30000-90236).
- 5 Reposition the file at its beginning. The next record read or written is the first record in the file. This code is not valid for files accessed as "append-only". Note that on a labeled magnetic tape file, the tape is positioned at the beginning of the opened file, and not necessarily at the beginning of the volume.
- 6 Write end-of-file. This operation is used to denote the end-of-file (EOF) on disc or magnetic tape, and is effective only for those devices. If applied to a disc file, the operation writes a logical end-of-data indicator at the point where the file was last accessed. The disc file label also is updated and written to disc. For a file residing on unlabeled magnetic tape, a tape mark is written at the current position of the tape. This *controlcode* is not allowed for labeled magnetic tape files. When applied in Interprocess Communication (IPC), this *controlcode*, is used to verify the state of the file by writing the file label and buffer area to disc; this ensures that the message file can survive system crashes. No EOF is written.
- 7 Space forward to tape mark. This moves a magnetic tape forward until a tape mark is encountered. If used on labeled magnetic tapes, the tape is positioned at the beginning of user trailer labels, if any.
- 8 Space backward to tape mark. On unlabeled tapes, this moves a magnetic tape backward until a tape mark is encountered. If used on labeled tapes, the tape is positioned at the beginning of user header labels, if any.
- 9 Rewind and unload tape. This repositions a magnetic tape file at its beginning and places the tape offline. Not allowed for labeled tapes.

NOTE

Control codes 0 and 3 will be rejected for spooled devicefiles. Control codes 5 through 9 (magnetic tape control) will be rejected for spooled :DATA tapes. Control codes 6 and 9 will be rejected for labeled magnetic tape files. All descriptions pertaining to magnetic tape are also valid for serial disc. Refer to the discussion of magnetic tape files in the MPE File System Reference Manual (30000-90236) for special considerations not covered here.

The following values for *controlcode* 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. Also listed are the default settings for each:

- 10 Change terminal input speed. (Not reset in BREAK mode; not reset after FCLOSE.)
- 11 Change terminal output speed. (Not reset in BREAK mode; not reset after FCLOSE.)
- 12 Turn echo facility on. (Not reset in BREAK mode; not reset after FCLOSE. Default.)
- 13 Turn echo facility off. (Not reset in BREAK mode; not reset after FCLOSE.)
- 14 Disable the system BREAK function. (Reset after FCLOSE.)
- 15 Enable the system BREAK function. (Reset after FCLOSE. Default.)
- 16 Disable the subsystem BREAK function. (Reset in BREAK mode; reset after FCLOSE. Default.)
- 17 Enable the subsystem BREAK function. (Reset in BREAK mode; reset after FCLOSE.)
- 18 Disable tape option. (Default.)
- 19 Enable tape option.
- 20 Disable the terminal input timer. (Reset in BREAK mode; reset after FCLOSE. Default.)
- 21 Enable the terminal input timer. (Reset in BREAK mode; reset after FCLOSE.)
- 22 Read the terminal input timer.
- 23 Disable parity checking. (Default.)
- 24 Enable parity checking.
- 25 Define line termination characters for terminal input.

- 26 Disable binary transfers. (Default.)
- 27 Enable binary transfers.
- 28 Disable user block mode transfers. (Not reset in **BREAK** mode; not reset after **FCLOSE**. Default.)
- 29 Enable user block mode transfers. (Not reset in **BREAK** mode; not reset after **FCLOSE**.)
- 34 Enable line deletion echo suppression. (Not reset in **BREAK** mode; reset after **FCLOSE**. Default.)
- 35 Disable line deletion echo suppression. (Not reset in **BREAK** mode; reset after **FCLOSE**.)
- 36 Set parity.
- 37 Allocate a terminal.
- 38 Set terminal type.
- 39 Obtain terminal type information.
- 40 Obtain terminal output speed.
- 41 Set unedited terminal mode.
- 43 Aborts pending **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 Enable/disable extended wait. For Interprocess Communication, a value of 1 for *param* enables extended wait. This 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 will remain in effect until an **FCONTROL** call with a *controlcode* of 45 and a *param* value of 0 is issued. Disabled extended wait, *param* value of 0, specifies that if a second **FREAD** call is issued and it encounters an empty file that has no reader, it will return an end-of-file condition.

Default: 0.
- 46 Enable/disable reading writer's ID. For Interprocess Communication, a value of 1 for *param* enables reading the writer's ID. Each record read will have a two-word header. The first word will indicate the type of record with the following codes:
 - 0 Data record.
 - 1 Open record.
 - 2 Close record.

The second word will contain the writer's ID number. If the record is a data record, the data will follow the header; open and close records contain no more information. If the value of *param* 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 Nondestructive read. If the value of *param* is 1 the next FREAD by this reader will not delete the record. Subsequent FREAD calls will be unaffected. When 0 is specified for *param* the next FREAD by this reader will delete the record.

Default: 0.

- 48 Enable/disable software interrupts. The external label (*plabel*) is contained in *param* of your interrupt procedure. In SPL it is passed as a parameter by placing an "@" before the procedure name. If the value of *param* is 0, the interrupt mechanism is disabled for this file.

param

logical (required)

If *controlcode* is 1, *param* denotes a word containing the value to be transmitted to the terminal or line printer driver as a carriage control or mode-control directive. The carriage control directive is selected from the listing in Table 2-5, located in the description of the FWRITE intrinsic.

The mode-control directive determines whether any carriage control directive transmitted through the FWRITE intrinsic takes effect before printing (pre-space movement) or after printing (post-space movement). The mode-control directive is selected from octal codes %400 or %401 listed in Table 2-5, located in the description of the FWRITE intrinsic.

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

0 Post-spacing.

1 Pre-spacing.

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

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

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

FDELETE

NO INTRINSIC NUMBER ASSIGNED

Deactivates a Relative I/O (RIO) record.

SYNTAX

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

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

PARAMETERS

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

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Not applicable to message files.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

FDEVICECONTROL

NO INTRINSIC NUMBER ASSIGNED

Provides control operations to a printer, Workstation Configurator, (version G.01.00 or later) or a spooled devicefile.

SYNTAX

```
IV      LA      IV      IV
FDEVICECONTROL(filenum,target,tcount,controlcode,
               LV      LV      I
               param1,param2,errnum);
```

FDEVICECONTROL may be used to download character sets, forms, and internal or control tables used in printing. It may also be used to control the page size, pen positioning, use of character sets and form, and the number of copies of each page to be printed, and other characteristics of the printing environment. The IFS/3000 intrinsics (which perform the same functions for the HP 268x as FDEVICECONTROL), together with the layout of the Character Set Load Record and Form Set Load Record and the Logical Page Table, are discussed in the IFS/3000 Manual (36580-90001).

PARAMETERS

filenum *integer by value (required)*
A word supplying the file number of the devicefile. This value is obtained from FOPEN.

target *logical array (required)*
This array contains data to be passed to the device. In general, it contains character sets, forms, or Vertical Format Control (VFC) information.

tcount *integer by value (required)*
An integer indicating the length of *target*, the units of which are determined by the sign. A positive value indicates the specified length is words; a negative value indicates bytes.

controlcode *integer by value (required)*
The code number of the operation to be performed. These are described below.

128 - Character Set Selection.

param1 (8:8) Primary character set identification.

param2 (8:8) Secondary character set identification.

The HP 268x page printer can contain up to 32 character sets, thus allowing the use of a variety of fonts, styles, print rotations, and languages. Use *controlcode* 134 to download character sets to the printer. Use *controlcode* 128 to select any two downloaded character sets to be the current primary and secondary character sets.

To change to the secondary character set a character at a time, set the eighth bit of the byte coding for the desired ASCII character. The HP 268x will strip out this bit and print, in the secondary character set, the character represented by the remaining 7-bit value. To change to the secondary character set for a number of characters and over several lines, insert a shift-in character (N^C) in the data. Insert a shift-out character (O^C) where the primary character set is to be re-activated.

129 - Logical Page Activation/Deactivation Request.

<i>param1</i> (0:1)	=0 Ignores the left byte of <i>param2</i> .
	=1 Deactivates the Logical Page Table entry identified in the left byte of <i>param2</i> .
<i>param1</i> (1:1)	=0 Ignores the right byte of <i>param2</i> .
	=1 Activates the Logical Page Table entry identified in the right byte of <i>param2</i> .
<i>param2</i> (0:8)	Logical Page Table entry (from 0 to 31) to be deactivated. Ignored if <i>param1</i> (0:1)=0.
<i>param2</i> (8:8)	Logical Page Table entry (from 0 to 31) to be activated. Ignored if <i>param1</i> (1:1)=0.

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

130 - Relative Pen Displacement.

<i>param1</i>	A 16-bit signed integer containing the desired X axis displacement, in dots, of the pen from its current position.
<i>param2</i>	A 16-bit signed integer containing the desired Y axis displacement, in dots, of the pen from its current position.

No pen movement will result from requests to move the pen off the logical page. As the coordinate system is based upon the current logical page itself and not upon the page's orientation with respect to the printer, you need not consider how the page has been rotated when assigning displacement values to *param1* and *param2*. Since the dot density for the HP 2680 (180 dots per inch) differs from that for the HP 2688 (300 dots per inch), the effects of *param1* and *param2* will be different.

131 - Absolute Pen Move.

<i>param1</i>	An integer containing the X coordinate, in dots, of the point to which you wish to move the pen.
<i>param2</i>	An integer containing the Y coordinate, in dots, of the point to which you wish to move the pen.

The values in *param1* and *param2* are measured from the upper left corner of the logical page. As with *controlcode* 130, you need not take page rotation into account when assigning coordinates, and the printer will not move the pen if the location you specify is off the logical page. Since the dot density for the HP 2680 (180 dots per inch) differs from that for the HP 2688 (300 dots per inch), the effect of *param1* and *param2* will be different.

132 - Define Job Characteristics.

<i>param1</i> (0:1)	=0 This bit is ignored. =1 The printer will not print job separation marks until the next job is open.
<i>param1</i> (1:1)	=0 This bit is ignored. =1 <i>Param2</i> contains the maximum allowable number of copies of each page.
<i>param2</i>	Significant only if <i>param1</i> (1:1)=1. This specifies the maximum number of copies the printer will make of any one page for the current job. The default maximum is 32,767.

133 - Define Physical Page.

The following bits are ignored if set to zero:

<i>param1</i> (0:1)	=1 Turn on multicopy form overlay feature.
<i>param1</i> (1:1)	=1 Turn off multicopy form overlay feature.
<i>param1</i> (2:1)	=1 Reserved.
<i>param1</i> (3:1)	=1 Redefine the physical page length.
<i>param1</i> (4:1)	=1 Redefine the number of copies per page desired.
<i>param1</i> (5:1)	=1 Reserved.
<i>param1</i> (6:1)	=1 Reserved.
<i>param1</i> (7:1)	=1 Reserved.

<i>param1</i> (8:8)	New physical page length in units of 0.25 inches. The length may not be less than 3.0 inches (a value of 12) or greater than 17.0 inches (a value of 68).
<i>param2</i>	Contains the number of copies of each page you want to print. If this number exceeds the maximum defined in <i>param2</i> of <i>controlcode</i> 132, only the maximum number of copies is printed.

Although FDEVICECONTROL will accept 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, only use even values in *param1* (8:8). In other words, bit (15:1)=0.

134 - Download/Delete Character Set.

<i>param1</i> (0:1)	=0 Download the character set identified in the right-hand byte of <i>param2</i> into the HP 268x. =1 Purge the character set identified in the right-hand byte of <i>param2</i> from the HP 268x.
<i>param2</i> (0:1)	=0 Indicates the first record of a load. =1 Indicates a continuation of the previous record.
<i>param2</i> (8:8)	Character set identifier - an integer from 0 to 31.

If you attempt to download a character set having the same identifier as one existing in the printer, then the HP 268x will purge the existing character set and repack 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 whenever you load, overlay, or delete a character set, form, or Vertical Format Control set (VFC).

135 - Download/Delete Form.

<i>param1</i> (0:1)	=0 Load the form set identified in the right-hand byte of <i>param2</i> . =1 Purge the identified form set from the HP 268x printer's memory.
<i>param2</i> (0:1)	=0 Indicates the first record of a load. =1 Indicates a continuation of the previous record.
<i>param2</i> (8:8)	Form set identifier - an integer from 0 to 31.

If you attempt to download a form set having the same identifier as one existing in the printer, then the HP 268x will purge the existing form set and repack 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 - Download Logical Page Table.

<i>param1</i>	Is not used.
<i>param2</i> (0:1)	=0 Indicates the first record of a load.
	=1 Indicates a continuation of the previous record.

A logical page is a page of data that may or may not take up an entire sheet of paper. It is possible to print up to 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 may be varied.

137 - Download Multicopy Form Overlay Table.

<i>param1</i>	Is not used.
<i>param2</i>	Is not used.

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

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

138 - Download/Delete Vertical Format Control (VFC).

param1 (0:1) =0 Load a VFC.
 =1 Delete a VFC.

param2 (0:1) =0 This is the first record of a load.
 =1 This record is logically a continuation of the
 previous record.

param2 (8:8) VFC set identifier - an integer from 0 to 31.

The Vertical Format Control 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, you may use the same or different Vertical Format Control tables for logical pages of different rotations.

139 - Download/Delete a Picture.

param1 (0:1) =0 Load a picture.
 =1 Delete a picture.

param2 (0:1) =0 This is the first record of a load.
 =1 This record is a logical continuation of the
 previous record.

param2 (8:8) Picture identifier - an 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 will be freed as soon as the page has been transferred to paper.

140 - Page Control.

<i>param1</i> (15:1)	=0 Do not eject a physical page.
	=1 Do a physical page eject before going to the specified logical page. This bit has no effect if this is the first record since an environment load, FOPEN or FCLOSE.
<i>param1</i> (13:2)	Auto eject mode.
	=00 Use auto eject flag of last data record (default at start of job is 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>param2</i> (8:8)	Logical page number - an integer from 0 to 31.

The logical page identified in *param2* becomes the current logical page even if other logical pages have entries which precede it in the Logical Page Table. FDEVICECONTROL activates the specified page if it is inactive, and the HP 268x performs a physical page eject if *param1* (15:1)=1.

141 - Clear Environment. When set to zero, the following bits are ignored.

<i>param1</i> (0:1)	=1 Clear all character sets.
<i>param1</i> (1:1)	=1 Clear all forms.
<i>param1</i> (2:1)	=1 Clear all Vertical Format Controls (VFCs).
<i>param1</i> (3:1)	=1 Clear all pictures.
<i>param2</i>	Is not used.

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

142 - Reserved.

143 - Load the Default Environment.

<i>param1</i>	Is not used.
<i>param2</i>	Is 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 - Print Picture.

<i>param1</i> (0:1)	=0 Temporary picture. =1 Addressable picture.
<i>param1</i> (1:1)	=0 X and Y are relative to the current pen position. =1 X and Y are absolute pen position to the logical page.
<i>param1</i> (2:14)	X coordinate for picture placement (radixed integer).
<i>param2</i> (0:1)	=0 First of temporary picture load. =1 Continuation record for load.
<i>param2</i> (1:1)	Is not used.
<i>param2</i> (2:14)	Y coordinate for picture placement (radixed integer).

The X and Y values are radixed integers expressed in the coordinate system of the logical page. The *param1* values allow the specification of picture location relative to the current position of the logical pen or at an absolute location on the logical page. In either case, the position refers to the user-selected location on the picture specified in the Picture Descriptor Block (PDB) as downloaded by the host. No bounds checking is done for pictures.

145 - End of Job.

<i>param1</i>	Is not used.
<i>param2</i>	Is not used.

146 - Device Extended Capability Mode.

<i>param1</i>	=0 Clear. =1 Set.
<i>param2</i>	Is not used.

192 - Access Workstation Configurator terminal configuration file.

param1 Item number of function. Refer to the Workstation Configurator Reference Manual (30239-90001) for values.

param2 Access code
1= Return current value in *target*.

2= Set item to value in *target*.

3= Set item to value in *target*, and return current value in *target*.

193 -Record processing information for NRJE spoolfiles.

param1 Integer indicating the character code (e.g. ASCII) of spoolfile data. Refer to SNA NRJE Network Remote Job Entry User/Programmer Reference Manual (30245-90001).

param2 (14:1) 0: Data is not compacted.

1: Data is compacted.

param2 (15:1) 0: Data is not compressed.

1: Data is compressed.

param1, param2 *logical by value (required)*
For each value of *controlcode*, there may be several possible values for *param1* and *param2*, which define the operation in more detail. These are described in the list of operation code numbers under *controlcode*.

errnum *integer (required)*
If no error occurs *errnum* is set to zero. If an error occurs *errnum* contains the File System error code. If FDEVICECONTROL detects a bounds violation for *errnum* (i.e. an address outside the user's stack area), *errnum* is unchanged.

CONDITION CODES

CCE Request granted.

CCG Not returned by this intrinsic.

CCL Request denied because an error occurred.

ADDITIONAL DISCUSSION

IFS/3000 Reference Guide (36580-90001).

Point-to-Point Workstation I/O Reference Manual (30000-90250).

FERRMSG

INTRINSIC NUMBER 305

Returns message corresponding to FCHECK error number.

SYNTAX

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

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

PARAMETERS

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

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Using FERRMSG" in Section IV.

Provides access to file information.

SYNTAX

O-V	IV	IV	BA
FFILEINFO(<i>filenum</i>	[, <i>itemnum1</i> , <i>itemvalue1</i>]		
	[, <i>itemnum2</i> , <i>itemvalue2</i>]		
	[, <i>itemnum3</i> , <i>itemvalue3</i>]		
	[, <i>itemnum4</i> , <i>itemvalue4</i>]		
	[, <i>itemnum5</i> , <i>itemvalue5</i>]);		

The *itemnum/itemvalue* parameters must appear in pairs. Up to five items of information can be retrieved by specifying one or more *itemnum/itemvalue* pairs. FFILEINFO is designed to allow for future changes so that new file information can be defined and accessed.

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> MPE file number returned by FOPEN.
<i>itemnum</i>	<i>integer by value (optional)</i> Cardinal number of the item desired; this specifies which item value is to be returned. (Refer to "Item#" in Table 2-3.)
<i>itemvalue</i>	<i>byte array (optional)</i> Returns the value of the item specified by the corresponding <i>itemnum</i> ; the data type of the item value depends on the item itself. (Refer to "Item Value" in Table 2-3.)

CONDITION CODES

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

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

Table 2-3. Item Values Returned by FFILEINFO

ITEM#	ITEM VALUE	TYPE	UNITS
1	Filename	BA (see FGETINFO)	
2	Foptions	L (see FGETINFO)	
3	Aoptions	L (see FGETINFO)	
4	Record	I (see FGETINFO)	words/bytes
5	Device type	I (see FGETINFO)	
6	Ldev # of remote file	L (see FGETINFO)	
7	Hdaddr	L (see FGETINFO)	
8	File code	I (see FGETINFO)	
9	Record pointer	D (see FGETINFO)	
10	EOF	D (see FGETINFO)	
11	File limit	D (see FGETINFO)	records
12	Log count	D (see FGETINFO)	records
13	Physcount	D (see FGETINFO)	records
14	Block size	I (see FGETINFO)	records/bytes
15	Extent size	L (see FGETINFO)	sectors
16	Number of extents	I (see FGETINFO)	
17	User labels	I (see FGETINFO)	
18	Creator ID	BA (see FGETINFO)	
19	Label address	D (see FGETINFO)	
20	Blocking factor	I (see FOPEN)	
21	Physical block size	I	words
22	Data block size	I	words
23	Offset to data in blocks	I	words
24	Offset of Active Record Table	I (RIO files)	words
25	Size of Active Record Table within the block	I	words
26	Vol. ID(label tape)	BA	
27	Vol. set ID(label tape)	BA	
28	Expiration date(Julian)	I	
29	File sequence number	I	
30	Reel number	I	
31	Sequence type	I	
32	Creation date(Julian)	I	
33	Label type	I	
34	Current# of writers	I	
35	Current# of readers	I	
36	File Allocation Date	L (CALENDAR format)	
37	File Allocation Date	D (CLOCK format)	
38	SPOOLFILE Devicefile number	L	
	(Bit (0:1) = 0 Input Spoolfile		
	(0:1) = 1 Output Spoolfile		
	Bits (1:15) Devicefile Number)		

Table 2-3. Item Values Returned by FFILEINFO (Continued)

ITEM#	ITEM VALUE	TYPE	UNITS
39	RESERVED		
40	Disc or diskette device status	D	
41	Device type	I	
42	Device subtype	I	
43	Environment file name	BA	
44	Last disc extent allocated	I	
45	Filename from labeled tape HDR1 record	BA	
46	Tape density	I	
47	DRT number	I	
48	UNIT number	I	
49	Software interrupt PLABEL	I	
50	Real device number of the file	I	
51*	Remote environment number	I	
52	Last modification time (CLOCK format)	D	
53	Last modification date (CALENDAR format)	L	
54	File creation date (CALENDAR format)	L	
55	Last access date (CALENDAR format)	L	
56	# data blocks in a variable length file	I	
57	# of the user label written to the file	I	
58	Number of opens for output	I	
59	Number of opens for input	I	
60	Terminal type, defined as:	I	
	0-File's associated device is not a terminal		
	1-Standard hardware or multi-point terminal		
	2-The terminal is connected via a phone-modem		
	3-DS pseudo terminal		
	4-X.25 Packet Switching Network PAD		
	(Packet Assembler Disassembler) terminal		
61**	NS/3000 remote environment id name		

* If NS/3000 RFA (Remote File Access) is being used specify DSDEVICE *ldev#* when a DS (point-to-point or X.25) link is being used.

** If NS/3000 RFA is being used: DSDEVICE *ldev#* in ASCII if a point-to-point link is being used; the X.25 node name if an X.25 link is being used. Users of item 61 must provide a buffer for the node name (or *envid*). This buffer must be able to accommodate the required space of 52 bytes. If not you risk data corruption on variables whose DB-relative location follows that of the Item 61 buffer or FSERR 73, "BOUNDS VIOLATION" from FFILEINFO.

FGETINFO

INTRINSIC NUMBER 11

Requests access and status information about a file.

SYNTAX

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

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

PARAMETERS

filenum

integer by value (required)

A word supplying the file number of the file about which information is requested.

filename

byte array (optional)

A 28-character byte array to which the actual designator of the file being referenced is returned, in the format:

filename.groupname.accountname

When the actual designator is returned, unused bytes in the array are filled with blanks on the right. A nameless file will return an empty string.

Default: The actual designator is not returned.

foptions

logical (optional)

The *foptions* parameter returns seven different file characteristics by setting corresponding bit groupings in a 16-bit word. Correspondence is from right to left. The file characteristics returned (the bit settings are summarized in Figure 2-1) are as follows:

Bits (14:2) - Domain *foption*.

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

=00 The file is a new file.

=01 The file is an old permanent file.

=10 The file is an old temporary file.

=11 The file is an old file.

Bit (13:1) - ASCII/binary *foption*.

=0 Binary.

=1 ASCII.

Bit (10:3) - Default file designator *foption*.

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

=001 The actual file designator is \$STDLIST.

=010 The actual file designator is \$NEWPASS.

=011 The actual file designator is \$OLDPASS.

=100 The actual file designator is \$STDIN.

=101 The actual file designator is \$STDINX.

=110 The actual file designator is \$NULL.

Bits (8:2) - Record format *foption*.

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

=00 Fixed-length records.

=01 Variable-length records.

=10 Undefined-length records.

=11 Spoolfile.

Bit (7:1) - Carriage control *foption*.

=0 No carriage-control character expected.

=1 Carriage-control character expected.

Bit (6:1) - MPE tape label *foption*.

=0 Nonlabeled tape.

=1 Labeled tape.

Bit (5:1) - Disallow :FILE equation *foption*.

This option ignores any corresponding :FILE command, so that the specifications in the FOPEN call take effect (unless overridden by those in the file label). The following bit settings apply:

=1 Disallow :FILE command.

=0 Allow :FILE command.

Bits (2:3)- File type *foption*.

=000 Ordinary file.

=001 KSAM file.

=010 Relative I/O file.

=100 Circular file (discussed in Section III).

=110 Message file.

Bits (0:2) - Reserved for MPE. Should be set to zero.

Default: *Foptions* are not returned.

aoptions

logical (optional)

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

Bits (12:4) - Access type *aoptions*.

The type of access allowed users of this file follows:

=0000 Read access only.

=0001 Write access only.

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

=0011 Append access only.

=0100 Input/output access.

=0101 Update access.

=0110 Execute access.

Bit (11:1) - Multirecord *aoption*.

=0 Select nonmultirecord mode.

=1 Select multirecord mode.

Bit (10:1) - Dynamic locking *aoption*. The bit settings are:

=0 Disallow dynamic locking/unlocking.

=1 Allow dynamic locking/unlocking.

Bits (8:2) - Exclusive *aoption*.

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

=00 Default.

=01 Exclusive access.

=10 Semi-exclusive access.

=11 Shared access.

Bit (7:1) - Inhibit buffering *aoption*.

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

=0 Normal buffering.

=1 Inhibit buffering.

Bit (5:2) - Multiaccess mode *aoption*.

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

=00 Nonmultiaccess.

=01 Multiaccess.

=10 Inter-job multiaccess.

Bit (4:1) - NOWAIT I/O *aoption*.

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

=0 NOWAIT I/O not in effect.

=1 NOWAIT I/O in effect.

Bits (3:1) - File Copy Access *aoption*.

=0 Access in file's native mode.

=1 Access as standard sequential file.

Bits (0:3) Reserved for MPE.

Default: *Aoptions* are not returned.

recsize

integer (optional)

A word to which the logical record size associated with the file is returned. If the file was created as a binary type, this value is positive and expresses the size in words. If the file was created as ASCII type, this value is negative and expresses the size in bytes.

For Interprocess Communication (IPC) when a call to FCONTROL with a *controlcode* of 46 ("FCONTROL 46") is in effect, the value returned in *rec-size* will be the size of the user's data records, including the two-word header.

Default: The logical record size is not returned.

devtype

integer (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 disc 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 disc class.

Default: The device type and subtype are not returned.

ldnum

logical (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 disc file, the logical device number will be that of the first extent. If the file is spooled, *ldnum* will be a virtual device number which does not correspond to the system configuration I/O device list. If the file is located on a remote computer, 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, 0 is returned. For version G.02.00 or later, if either the DS device for the RFA or the *ldev* is 0, then the entire word is 0.

Default: The logical device number is not returned.

hdaddr

logical (optional)

A word to which the hardware address of the device is returned, where bits (0:8) 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.

Default: The hardware address is not returned.

filecode

integer (optional)

A word to which a disc file's file code is returned.

Default: The file code is not returned.

<i>recpt</i>	<p><i>double (optional)</i></p> <p>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.</p> <p>Default: The logical record pointer setting is not returned.</p>
<i>EOF</i>	<p><i>double (optional)</i></p> <p>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 disc, 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.</p> <p>Default: The number of logical records in the file is not returned.</p>
<i>flimit</i>	<p><i>double (optional)</i></p> <p>A double-word to which a double positive integer representing the number of the last logical record that could ever exist in the file, because of the physical limits of the file is returned. If the file does not reside on disc, this value will be zero.</p> <p>Default: The logical record count is not returned.</p>
<i>logcount</i>	<p><i>double (optional)</i></p> <p>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.</p> <p>Default: The logical record count is not returned.</p>
<i>physcount</i>	<p><i>double (optional)</i></p> <p>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.</p> <p>Default: The number of I/O operations is not returned.</p>
<i>blksize</i>	<p><i>integer (optional)</i></p> <p>A word to which the block size associated with the file is returned. 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.</p> <p>Default: The block size is not returned.</p>
<i>extsize</i>	<p><i>logical (optional)</i></p> <p>A word to which the disc extent size associated with the file (in sectors) is returned.</p> <p>Default: The disc extent size is not returned.</p>

<i>numextent</i>	<p><i>integer (optional)</i></p> <p>A word to which the maximum number of disc extents allowable for the file is returned.</p> <p>Default: The maximum allowable number of extents is not returned.</p>
<i>userlabels</i>	<p><i>integer (optional)</i></p> <p>A word to which the number of user header labels defined for the file when it was created is returned. If the file is not a disc file, this number is zero. When an old file is opened for overwrite output, the value of <i>userlabels</i> is not reset and old user labels are not destroyed.</p> <p>Default: The number of user labels is not returned.</p>
<i>creatorid</i>	<p><i>byte array (optional)</i></p> <p>A byte array to which the eight-byte name of the user who created the file is returned. If the file is not a disc file, blanks are returned.</p> <p>Default: The user name is not returned.</p>
<i>labaddr</i>	<p><i>double (optional)</i></p> <p>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 disc address. If the file is not a disc, zero is returned.</p> <p>Default: The sector address is not returned.</p>

CONDITION CODES

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

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

BITS	(0:2)	(2:3)	(3:1)	(6:1)	(7:1)	(8:2)	(10:3)	(13:1)	(14:2)
FIELD	(RE-SERVED)	FILE TYPE	DISALLOW :FILE	MPE TAPE LABELS	CAR-RIAGE CONTROL	RECORD FORMAT	DEFAULT FILE 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=Labled Tape	0=NOCTL 1=CTL	00=Fixed 01=Variable 10=Un-defined 11=Spoolfile	000=FILENAME 001=\$STDLIST 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 Permanent Or Temporary File

NOTE: Double lines indicate octal digit boundaries.

Figure 2-1. Foptions Bit Summary

BITS	(0:3)	(3:1)	(4:1)	(5:2)	(7:1)	(8:2)	(10:1)	(11:1)	(12:4)
FIELD	(RE-SERVED)	FILE COPY ACCESS	NOWAIT I/O	MULTI-ACCESS	INHIBIT BUF-FERING	EXCLU-SIVE ACCESS	DYNAMIC LOCKING	MULTI-RECORD ACCESS	ACCESS TYPE
MEANING		0=Access Native Mode 1=Access As Stand-ard Se-quential File	1=NOWAIT 0=Non-NOWAIT	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=Semi-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

NOTE: Double lines indicate octal digit boundaries.

Figure 2-2. Aoptions Bit Summary

FINDJCW

INTRINSIC NUMBER 86

Searches the Job Control Word Table for a specified Job Control Word (JCW).

SYNTAX

BALI

```
FINDJCW(jcwname, jcwvalue, status);
```

PARAMETERS

- jcwname*** *byte array (required)*
A byte array containing the name of the Job Control Word (JCW) to be found. May contain up to 255 alphanumeric characters, starting with a letter and ending with a nonalphanumeric character such as a blank.
- jcwvalue*** *logical (required)*
A word to which the value of *jcwname* is returned, if *jcwname* is found. If *jcwname* is not found, *jcwvalue* is unchanged.
- status*** *integer (required)*
A word to which a value denoting the execution status of the intrinsic is returned, as follows:
- 0 Successful execution, *jcwname* found.
 - 1 Error, *jcwname* greater than 255 characters long.
 - 2 Error, *jcwname* does not start with a letter.
 - 3 Error, *jcwname* not found in JCW table.
 - 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.
- Value 5 will only be returned if the fundamental operating software is version G.00.00 or later.

SPECIAL CONSIDERATIONS

There are three types of JCWs in the system: user-defined JCWs, system-defined JCWs, and system-reserved JCWs. FINDJCW can return the value of any type of JCW. The system-reserved JCWs are HPDATE, HPDAY, HPHOUR, HPMINUTE, HPMONTH, and HPYEAR.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Interprocess Communication" in Section V.

FINTEXTIT

INTRINSIC NUMBER 23

Causes the return from a user's interrupt procedure. (Available only on version G.01.00 or later.)

SYNTAX

0-VLV

```
FINTEXTIT(intstate);
```

The FINTEXTIT intrinsic is used to cause the return from the user's interrupt procedure. On the return, software interrupts are set according to *intstate*. If *intstate* is omitted, the software interrupts are enabled by default.

PARAMETERS

intstate

logical by value (optional)

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

=1 Enable software interrupts.

=0 Leave software interrupts disabled.

Default: (15:1)=1.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

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

SYNTAX

```
L          LV
oldstate:=FINTSTATE(intstate);
```

The software interrupt facility enables users to perform FREAD/FWRITE completion processing with their own interrupt procedure. An FREAD/FWRITE call is necessary to initiate the I/O request. Both of these intrinsics return to the user's process as soon as the request has been started. When the operation completes, the user's program is trapped (or "interrupted") and goes to a user chosen interrupt procedure. This performs whatever processing is necessary and then resumes the user's original program.

Soft interrupts are "armed" for a particular file by specifying the interrupt procedure's *plabel* in an FCONTROL call with a *controlcode* of 48. Calling "FCONTROL 48" with a parameter of 0 will disarm the software interrupt mechanism.

NOTE

MPE inhibits software interrupts just before entering an interrupt procedure. This is done to stop unwanted nesting of the interrupt procedures. Each interrupt procedure should call FINTEXT (Refer to FINTEXT in this section) to re-enable other interrupts just before it exits.

Software interrupts are normally automatically inhibited before a YC trap procedure. The trap procedure may elect to allow software interrupts, however, by calling the FINTSTATE intrinsic. The RESETCONTROL intrinsic will restore the interrupt state of the process to its pre-YC value (unless the trap procedure issues an FINTSTATE call, in which case RESETCONTROL makes no change).

When the software interrupt is executed, location Q-4 in the stack will contain the file number of the file that caused the interrupt.

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

FUNCTIONAL RETURN

oldstate

logical

The old state (enabled or disabled) of software interrupts is returned by this procedure.

PARAMETERS

intstate

logical by value (required)

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

=0 Disable software interrupts.

=1 Enable software interrupts.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

An uncompleted FREAD/FWRITE request may be aborted by issuing an FCONTROL call with a *control-code* 43 (abort NOWAIT I/O).

Limitations:

- Only message (MSG) files allow soft interrupts.
- No more than one uncompleted FREAD/FWRITE may be outstanding for a particular file.
- The interrupt is held off while the user is executing within MPE, with the following exceptions: PAUSE and IOWAIT will allow the interrupt. The interrupt handler's return stack marker in this case will be set to reinvoke the intrinsic.
- FINTSTATE may not be used with remote files.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

Returns information from the file label of a disc file. (Available version G.02.00 or later.)

SYNTAX

	BA	IV	I	IA	BA	IA
FLABELINFO	(<i>fname</i> ,	<i>mode</i> ,	<i>errorcode</i> ,	<i>items</i> ,	<i>itemvalues</i> ,	<i>itemerrors</i>);

The FLABELINFO intrinsic allows information on a specific disc file to be extracted from the file system whether or not the file is opened. The information returned is a subset of the information returned from the FFILEINFO intrinsic. FLABELINFO is not supported for remote files.

PARAMETERS

- fname*** *byte array (required)*
The name of the file terminated by a nonalphanumeric character other than a period (.) or a slash (/). The filename may include password, group, and account specifications. The file must be in the permanent file directory. The filename also may backreference a file equation and optionally be preceded by an asterisk.
- mode*** *integer by value (required)*
An integer specifying the valid backreferencing (to file equations) for the file. Valid mode values are:
- 0 Use file equation if one exists.
 - 1 Must use file equation (error if one does not exist).
 - 2 Ignore any existing file equations.
- errorcode*** *integer (required)*
Returns any general warnings or errors which affect the FLABELINFO call. Specifically, it will return an error number indicating that the call failed or that an error occurred in one of the specific items. An error code of zero indicates no errors, a positive error code indicates an error, and a negative error code indicates a warning.
- items*** *integer array (required)*
An array of item numbers terminated with a zero. Each item number describes which item is to be returned in the *itemvalues* array. (Refer to "Item#" in Table 2-4.)
- itemvalues*** *byte array (required)*
Information returned depending on the item number. (Refer to "Item Value" in Table 2-4.) The array contains all of the returned items. For example, if Item 1 and Item 2 are requested, the filename will be in the first eight bytes of the array, and the group name will be in the next eight bytes of the array.

itemerrors

integer array (required)

Array of error numbers which correspond to the items specified in the *item* array. If an element of the array is negative, a warning exists for that item. If the element is positive, an error was detected for that specific item. The absolute value of the error is the file system error number.

CONDITION CODES

CCE Request granted.

CCG Not returned by this intrinsic.

CCL Request denied because an error occurred. Refer to the *errorcode* and *itemerrors* parameters for specific error numbers.

ADDITIONAL DISCUSSION

The `FFILEINFO` discussion in this section.

The request for information from an Item# listed in Table 2-4 is granted or denied based on the access mode of the user:

- Read access to the file (R).
- Write access to the file (W).
- Read or write access to the file (R/W).
- Privileged call to intrinsic (P).
- Manager of file (M). (This is AM if file is within group, otherwise SM).
- Creator of the file (C).

Items 1 through 3 and 6 through 24 are available to any caller. Items 4 and 5 require either creator, privileged, or manager access. Item 25 requires privileged, manager, or read/write access.

Table 2-4. Item Values Returned By FLABELINFO

ITEM#	ITEM VALUE	TYPE	UNITS
1	File name in file label	BA	8 bytes
2	Group name in file label	BA	8 bytes
3	Account name in file label	BA	8 bytes
4	Creator name in file label	BA	8 bytes
5	Security matrix for access	D	
6	Creation date (calendar format)	L	
7*	Last access date (calendar format)	L	
8*	Last modified date (" ")	L	
9	File code	L	
10*	Number of user labels written	I	
11*	Number of user labels available	I	
12	File limit	D	
13	FOPTIONS	L	
14	Record size	I	bytes
15	Block size	D	words
16*	Number of extents	I	
17*	Last extent size	I	sectors
18	Extent size	I	sectors
19*	End of file record number	D	
20	File allocation time	D	
21	File allocation date	L	
22*	Number of open/close records (message files only)	D	
23	Device name	BA	8 bytes
24*	Last modify time	D	
25*	First user label (user label 0)	BA	256 bytes

* These items may not be up to date while the file is open.

FLOCK

INTRINSIC NUMBER 15

Dynamically locks a file.

SYNTAX

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

FLOCK provides a means of signaling that the caller temporarily wants exclusive use of a file.

PARAMETERS

- filenum*** *integer by value (required)*
A word supplying the file number of the file to be locked.
- lockcond*** *logical by value (required)*
A word specifying conditional or unconditional locking by setting bit (15:1) as follows:
- =0 Locking will take place only if the file's Resource Identification Number (RIN) is not currently locked. If the RIN is locked, control returns immediately to the calling process, with condition code CCG.
 - =1 Locking will take place unconditionally. If the file cannot be locked immediately, the calling process suspends until the file can be locked.

CONDITION CODES

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

- | | |
|-----|---|
| CCE | Request granted. |
| CCG | Not returned for this bit setting. |
| CCL | Request denied because this file was not opened with the "dynamic locking" <i>aoption</i> bit (10:1) specified in the FOPEN intrinsic, or the request was to lock more than one file and the calling process does not possess the Multiple RIN (MR) capability. |

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

- | | |
|-----|---|
| CCE | Request granted. |
| CCG | Request denied because the file was locked by another process. |
| CCL | Request denied because this file was not opened with the "dynamic locking" <i>aoption</i> bit (10:1) specified in the FOPEN intrinsic, or the request was to lock more than one file and the calling process does not possess the Multiple RIN (MR) capability. |

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Standard capability sufficient if only one file is to be locked dynamically. If more than one file is to be locked dynamically, the Multiple RIN (MR) capability is required.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

The FOPEN intrinsic in this section.

FLUSHLOG

NO INTRINSIC NUMBER ASSIGNED

Flushes the contents of the user logging memory buffer to the logging file.

SYNTAX

D I

FLUSHLOG(*index*,*status*);

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

PARAMETERS

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

status *integer (required)*
One of the following integers indicating the success/failure of the intrinsic call:

Message No.	Meaning
0	No error occurred for this call.
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. 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.

Message No.	Meaning
12	System is out of disc space , logging cannot proceed .
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user log file.
16	Invalid logging identifier.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capability required.

ADDITIONAL DISCUSSION

"User Logging" in Section III.

FMTCALENDAR

NO INTRINSIC NUMBER ASSIGNED

Converts any calendar date with the same format as the CALENDAR intrinsic into the format: "FRI, JAN 18, 1985".

SYNTAX

LV BA

FMTCALENDAR(*date*,*string*);

PARAMETERS

date *logical by value (required)*
A logical value representing any calendar date with the same format as the CALENDAR intrinsic:

Bits (7:9) - The day of the year.

Bits (0:7) - The year of the century.

string *byte array (required)*
A 17-character byte array in which the formatted calendar date is returned. If the day of the month is less than 10, an additional blank will precede it, for example: "FRI, JAN 8, 1985".

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Formatting Calendar Data and Time Information" in Section V.

FMTCLOCK

NO INTRINSIC NUMBER ASSIGNED

Converts the time of day with the same format as the CLOCK intrinsic, into the format: "12:39 AM".

SYNTAX

DVBA

FMTCLOCK(*time*,*string*);

PARAMETERS

time

double by value (required)

A double-word value representing the time of day in the same format as the CLOCK intrinsic:

Word 1:

Bits (8:8) - The minute of the hour.

Bits (0:8) - The hour of the day.

Word 2:

Bits (8:8) - The tenths of seconds.

Bits (0:8) - The seconds.

string

byte array (required)

An 8-character byte array in which the formatted time of day is returned. If the hour is a single digit (i.e. prior to 10:00) the array will contain a leading blank as follows: " 7:39 AM".

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Formatting Calendar Date and Time Information" in Section V.

FMTDATE

NO INTRINSIC NUMBER ASSIGNED

Converts calendar date and time of day with the same format as the CALENDAR and CLOCK intrinsics, into the format: "FRI, JAN 18, 1985, 12:39 AM".

SYNTAX

LV DV BA

FMTDATE(*date,time,string*);

PARAMETERS

date

logical by value (required)

A logical value with the same format as the CALENDAR intrinsic:

Bits (7:9) - The day of the year.

Bits (0:7) - The year of the century.

time

double by value (required)

A double value with the same format as the CLOCK intrinsic:

Word 1:

Bits (8:8) - The minute of the hour.

Bits (0:8) - The hour of the day.

Word 2:

Bits (8:8) - The tenths of seconds.

Bits (0:8) - The seconds.

string

byte array (required)

A 27-character byte array in which the formatted date and time are returned. If the day of the month is less than 10, an additional blank will precede it. Similarly, if the time is earlier than 10:00 (one digit), an additional blank will also precede the time, for example: "FRI, JAN 8, 1985, 7:39 AM".

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Formatting Calendar Date and Time Information" in Section V.

Used to establish access to a file and optionally, to define the physical characteristics of the file prior to access.

SYNTAX

```

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

```

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

FUNCTIONAL RETURN

filenum

integer

An integer file number used to identify the opened file in subsequent intrinsic calls.

PARAMETERS

formaldesignator

byte array (optional)

This array contains a string of ASCII characters interpreted as a formal file designator. This string must begin with a letter; contain alphanumeric characters, slashes, or periods; and terminate with any nonalphanumeric character except a slash (/) or a period (.). If the string is the name of a system-defined file, it can begin with a dollar sign (\$); if it is the name of a user-predefined file, it can begin with an asterisk (*). New KSAM files, unlike standard files, must be opened with a unique name. Effective with the G.02.00 release, the remote location of a device may be specified in *formaldesignator* as *filename:envid*.

Default: A temporary nameless file is assigned that can be read from or written to, but not saved.

foptions

logical by value (optional)

The *foptions* parameter allows you to specify different file characteristics by setting corresponding bit groupings in a 16-bit word. The correspondence is from right to left, beginning with bit 15. These characteristics are as follows, proceeding from the rightmost bit groups to the leftmost bit groups in the word. The bit settings are summarized in Figure 2-1, "Foptions Bit Summary, found in the description of the FGETINFO intrinsic.

Bits (14:2) - Domain *foption*.

The file domain to be searched by MPE to locate the file:

- =00 The file is a new file, created at this point. No search is necessary.
- =01 The file is an old file, and the system file domain should be searched.
- =10 The file is an old temporary file, and the job file domain should be searched.
- =11 The file is an old file, located first by searching the job file domain then, if it is not found, by searching the system file domain.

Bit (13:1) - ASCII/binary *foption*.

The code (ASCII or binary) in which a new file is to be recorded when it is written to a *device* that supports both codes. In the case of disc files, this also affects padding that can occur when a direct-write intrinsic call (FWRTEDIR) is issued to a record that lies beyond the current logical end-of-file indicator. In ASCII files, any dummy records between the previous end-of-file and the newly written record are padded with blanks. In binary files, such records are padded with binary zeros. By default, magnetic tape and serial disc files are treated as ASCII files; and foreign disc files are treated as binary. This bit has the following settings:

- =0 Binary file.
- =1 ASCII file.

Bit(10:3) - Default file designator *foption*.

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

- No explicit or implicit :FILE command equating the formal file designator to a different actual file designator occurs in the job or session.
- The "Disallow file equation" *foption* is true (bit (5:1)=1). Note that a leading "*" in a formal designator can effectively override the disallow file equation *foption*.

The bit settings are:

- =000 The actual file designator is the same as the formal file designator.
- =001 The actual file designator is \$STDLIST.
- =010 The actual file designator is \$NEWPASS.
- =011 The actual file designator is \$OLDPASS.
- =100 The actual file designator is \$STDIN.
- =101 The actual file designator is \$STDINX.
- =110 The actual file designator is \$NULL.

Bits (8:2) - Record format *foption*.

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

- =00 Fixed-length records. The file is composed of logical records of uniform length. Foreign discs always have fixed-length records.
- =01 Variable-length records. The file contains logical records of varying length. This format is restricted to records that are written in sequential order. The size of each record is recorded internally. The actual physical record size used is determined by multiplying the record size (specified or default) plus one by the blocking factor, and adding one word for the end-of-block indicator.

In the case of new files, this option is not allowed when NOBUF is specified. In such a case, the record format will be changed by these bit settings internally to undefined-length records, discussed below. If NOBUF is specified, then reads/writes are performed in terms of the entire block, not just the record; thus you must set up the variable structure before attempting an FWRITE. (Refer to record formats in the MPE File System Reference Manual (30000-90236).)

- =10 Undefined-length records. The file contains records of varying length that were not written using the variable-length *foption* (bits (8:2)=01). All files not on disc or magnetic tape are treated as containing undefined-length records by default. The file system makes no assumption about the amount of data that is useful. The user must determine how much data is 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; fixed- and variable-length records are supported by disc and magnetic tape devices only. To state this another way: disc and magnetic tape devices support all record formats, whereas all other devices support only undefined length records.

Bit (7:1) - Carriage control *option*.

If selected, this specifies that you will supply a carriage control directive in the calling sequence of each FWRITE call that writes records onto the file. This bit may be set as follows:

=0 No carriage control directive expected.

=1 Carriage control directive expected.

Carriage control is defined only for character-oriented (ASCII) files. This option and the "binary" option are mutually exclusive, and attempts to open new files with both binary and carriage control directives will result in an access violation. This option is a physical attribute of the file and its state cannot be modified when opening an old disc file.

A carriage-control character passed through the *control* parameter of FWRITE is recognized as such and acted upon for files that have carriage control specified in FOPEN. Embedded control characters are treated strictly as data on files for which no carriage control is specified, and they do not invoke spacing for such files. You may specify spacing action on files for which carriage control has been specified, either by embedding the control in the record, indicated with a *control* parameter of 1 in the call to FWRITE, or by sending the control code directly via the *control* parameter of FWRITE.

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

The control codes %400 through %403 are remapped to %100 through %103, so that they fit into one byte and thus can be embedded. Records written to the line printer with one of the above control codes should contain only control information.

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

For the purpose of computing record size, carriage control information is considered by the file system to be part of the data record. As such, specifying the carriage control option adds one byte to the record size when the file is originally created. For example, a specification of "REC=-132,1,F,ASCII;CTL" results in a *recsize* of 133 characters.

As a general rule you may read the entire file (the size of which is returned in the *recsize* parameter of FGETINFO or *item#* 6 (record) of FFILEINFO). (Refer to Table 2-3 found in the description of the FFILEINFO intrinsic.) However, on writes of files for which carriage control characters are specified the data transferred is limited to *recsize*-1 unless a control of 1 is passed, indicating the data record is prefixed with embedded carriage control characters.

Bit (6:1) - Labeled tape *foption*.

=1 Labeled tapes.

=0 No labeled tapes.

Bit (5:1) - Disallow :FILE equation *foption*.

This option ignores any corresponding :FILE command, so that the specifications in the FOPEN call take effect (unless pre-empted by those in the file label, for disc files). Note that a leading * in a formal designator can effectively override the disallow file equation *foption*.

=0 Allow :FILE.

=1 Disallow :FILE.

Bits (2:3) - File type *foption*.

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

=000 Ordinary file.

=001 KSAM file.

=010 Relative I/O file.

=100 Circular file.

=110 Message file.

NOTE

The default designator *foption*, bits (10:3), offers several choices for default file designators. Any value used other than 000 for "filename" will override the File Type field.

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

Bits (0:2) - Reserved for MPE. Should be set to zero.

aoptions

logical by value (optional)

The *aoptions* parameter permits you to specify up to seven different access options established by bit groupings in a 16-bit word. The correspondence is from right to left, beginning with bit 15. These access options are described below, proceeding from the rightmost bit groups to the leftmost bit groups in the word. The bit settings are summarized in Figure 2-2, "Aoptions Bit Summary", found in the description of the FGETINFO intrinsic.

Default: All bits are set to zero.

Bits (12:4) - Access Type *aoptions*.

The type of access allowed for this file is specified by the following bit settings:

- =0000 READ access only. FWRITE, FUPDATE, and FWRITEDIR intrinsic calls cannot reference this file. The end-of-file is not changed; the record pointer starts at 0.
- =0001 WRITE access only. Any data written in the file prior to the current FOPEN request is deleted. The FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file. The end-of-file is set to 0; the record pointer starts at 0. On magnetic tape an EOF mark will be written to the tape when the file is closed even if no data is written.
- =0010 WRITE access only, but previous data in the file is not deleted. The FREAD, FREADSEEK, FUPDATE, and FREADDIR intrinsic calls cannot reference this file. The end-of-file pointer is not changed, the record pointer starts at 0. Therefore, data will be overwritten if an FWRITE is done. The system will change this value to APPEND for message files.
- =0011 APPEND access only. The FREAD, FREADDIR, FREADSEEK, FUPDATE, FSPACE, FPOINT, and FWRITEDIR intrinsic calls cannot reference this file. The end-of-file pointer is used to set the record pointer prior to each FWRITE. For disc files it is updated (in an internal file system table) after each FWRITE. Thus, data cannot be overwritten.
- =0100 INPUT/OUTPUT access. Any file intrinsic except FUPDATE can be issued for this file. The end-of-file pointer is not changed, the record pointer starts at 0. Not valid for message files.
- =0101 UPDATE access. All file intrinsics, including FUPDATE can be issued for this file. The end-of-file pointer is not changed; the record pointer starts at 0. Not valid for message files.
- =0110 EXECUTE access. Allows user who is running in Privileged Mode input/output access to any loaded file. The end-of-file pointer is not changed, the record pointer starts at 0. Not valid for message files.

Bit (11:1) - Multirecord *aoption*.

Signifies that individual read or write requests are not confined to record boundaries. Thus, if the number of words or bytes to be transferred (specified in the *tcount* parameter of the read or write request) exceeds the size of the physical record (i.e. block) that is referenced, the remaining words or bytes are taken from subsequent successive records until the number specified by *tcount* has been transferred.

For message (MSG) files, the file system sets this bit to zero. This option is available only if the inhibit buffering *aoption*, described below, is also selected.

=0 Non-multirecord mode (NOMULTI).

=1 Multirecord mode (MULTI).

Bit (10:1) - Dynamic Locking *aoption* (disc file only).

Indicates use of the FLOCK and FUNLOCK intrinsics to dynamically permit or restrict concurrent access to the file by other processes at certain times. The user process can continue this temporary locking/unlocking until it closes the file. Dynamic locking/unlocking is made possible through a Resource Identification Number (RIN) assigned to the file and temporarily acquired by FOPEN. The calling process and other processes must use the RIN in cooperation to guarantee the integrity of the file, as discussed under "Resource Management" in Section III. Non-cooperating processes are allowed concurrent access at all times, unless other provisions prohibit this. You must have LOCK access at account, group, and file levels for FOPEN to grant the dynamic locking *aoption*. (LOCK access is available if LOCK, APPEND, or WRITE access is set for you at these levels.)

=0 Disallow dynamic locking/unlocking.

=1 Allow dynamic locking/unlocking. A disc file may be accessed concurrently only if all FOPEN requests for the file specify dynamic locking. An FOPEN request that disagrees with the current access, if any, will fail. This bit is ignored for files not residing on disc.

Bits (8:2) - Exclusive *aoption*.

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

=00 Default value. If the "READ access only" *aoption* is selected, SEMI-exclusive access (8:2)=10 takes effect. Otherwise, EXCLUSIVE access (8:2)=01 takes effect. Regardless of which access is selected, FGETINFO will report (8:2)=00.

=01 EXCLUSIVE access. After this file is opened, any additional FOPEN requests, 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 the FOPEN call is issued, CCL is returned to the calling process. If another FOPEN call is issued for this file while the "EXCLUSIVE access" *aoption* is in effect, an error code is returned to that calling process. The "EXCLUSIVE access" *aoption* can be requested only by users allowed the LOCK access mode by the security provisions for the file. For message files, there can be only one reader and one writer.

=10 SEMI-exclusive access. After the file is opened, concurrent output access to this file via 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 "INPUT/OUTPUT" or "UPDATE" *aoption* access type will obtain "READ" access. Other types of read access, however, are allowed. If a process already has OUTPUT access to the file when this FOPEN call is issued, a CCL error code is returned to the calling process. If another FOPEN call that violates the READ-only restriction is issued while the "SEMI-exclusive" *aoption* is in effect, that call fails and an error code is returned to the calling process. Semi-exclusive access can be requested only by users allowed the LOCK access mode by the security provisions for the file. For message files there can be multiple readers, but only one writer.

=11 SHARE access. After the file is opened, permits concurrent access to this file by any process, in any access mode, subject to other basic MPE security provisions in effect. For any message file there can be multiple readers and multiple writers.

Bit (7:1) - Inhibit buffering *aoption*.

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

=0 Allow normal buffering (BUF).

=1 Inhibit buffering (NOBUF).

NOBUF access is oriented to physical block transfer rather than logical record transfer. If this option is specified with the variable record format and the file is not variable length record format, the format will be changed internally to undefined length record format. Thus, you are responsible for buffer management. When performing an FWRITE, you will have to set up the variable structure. (Refer to the File System Reference Manual (30000-90236) for a discussion of record formats.)

With NOBUF access, you have 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 manipulated under NOBUF access follow the same rules as those files that are created using buffering. The default blocking factor for a file created under NOBUF is 1.

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

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

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

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

Non-RIO access to a RIO file can be indicated by specifying the NOBUF option. In this case the physical block size from FFILEINFO should be used to determine the maximum transfer length. (Refer to Table 2-3, Item #21 found in the description of the FFILEINFO intrinsic.) The FGETINFO *blksize* parameter may also be used.

For message files, the file system sets bit (7:1)=0. However, readers may open a message file with NOBUF if they are in copy mode; this determines whether they will be accessing the file record by record or by block. Thus for readers of message files, bit (7:1) has the following settings:

=0 Read by logical record.

=1 Read by physical block.

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

Bits (5:2) - MULTI access mode *aoption*.

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

=00 No MULTI access.

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

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

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

For message files, the file system changes bits (5:2)=00 to bits (5:2)=10 to allow global MULTI access.

Bit (4:1) - NOWAIT I/O *aoption*.

This bit allows the accessor to initiate an I/O request and to have control returned before the completion of the I/O. The NOWAIT I/O *aoption* implies the NOBUF *aoption*; if you do not specify NOBUF, the file system does it for you. Also, multirecord access is not available. This option is not available if the file is located on a remote computer.

=1 NOWAIT I/O in effect.

=0 NOWAIT I/O not in effect.

You must be running in Privileged Mode to use NOWAIT.

Bits (3:1) - File copy *aoption*.

This feature permits any file to be treated as a standard sequential file so that the file can be copied by logical record or physical block to another file.

=0 The file will be accessed in its native mode, i.e. a message file will be treated as a message file, a KSAM file as a KSAM file.

=1 The file is to be treated as a standard, sequential file with variable length records. For message files this allows nondestructive reading of an old message file at either the logical record or physical block record level. Only block level access is permitted if the file has message-file format to prevent incorrectly formatted data from being written to the message file while it is unprotected. In order to access a message file in copy mode, a process must have exclusive access to the file.

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

recsize

integer by value (optional)

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

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

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

For either ASCII or binary files with fixed or undefined length records, the record size is rounded up to the nearest word boundary. For example, a *recsize* specified as -106 for an ASCII file is 106 characters (53 words) in length. A *recsize* of -113 for a binary file is 114 characters (57 words) in length. The rounded sizes should be used in computations for *blockfactor* or block size. When a foreign disc is opened, *recsize* is forced to 128 words. (IBM diskettes are forced to 64 words.)

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

device

byte array (optional)

Contains a string of ASCII characters terminating with any 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 disc 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 disc file, *device* must be either a foreign disc class name or the *ldev* of a disc in a foreign disc class. If you specify *ldev*, the disc should be mounted on the drive prior to the FOPEN. Otherwise it may be assumed to be a serial disc by the system. 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 the *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* 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 disc; specification of any device other than disc opens the device. When this occurs, the file is no longer a message file.

Default: DISC.

formmsg

byte array (optional)

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

This array is also used for tape label information if bit (6:1) of the *foptions* parameter is set. The tape label information is formatted as follows:

. [volumeid [, type [, exp [, seq]]]] ;

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

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

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

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

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

seq One of the following methods to specify the position of the file in relation to other files on the tape:

A 0 which causes a search of all volumes until the file is found.

An unsigned integer (1-9999) that specifies the position of the file relative to the current file on the tape.

ADDF causes the tape to be positioned so as to add a new file on the end of the volume (or last volume in a multi-volume set).

NEXT will position 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 will remain at the beginning of the previous file.

Default: NEXT.

userlabels

integer by value (optional)

An integer specifying the number of user-label records to be written for this file. Applicable to new disc files only. The maximum number of user labels allowed varies from file to file. It depends on the final *blockfactor* and *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: The default number of user-label records is zero.

blockfactor

integer by value (optional)

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

Default: Calculated by dividing the specified *recsize* (in words) into the configured block size. This value is rounded downward to an integer that is never less than 1.

numbuffers

integer by value (optional)

A 16-bit word whose bits specify the number of buffers, number of copies, and output priority.

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

Bits (11:5) - Number of buffers.

Specifies the number of buffers to be allocated to the file. This parameter is not used for files representing interactive terminals, because a system-managed buffering method is always used in such cases. If omitted, set to zero, the default value of 2 is set by MPE.

For message files, a value between 2 and 31; default is 2. This parameter must not exceed the physical capacity of the file.

Bits (4:7) - Number of Copies.

For spooled output devices only, specifies the number of copies of the entire file to be produced by the spooling facility. This can be specified for a file already opened (for example, \$STDLIST), in which case the highest value supplied before the last FCLOSE will take effect. The copies do not appear contiguously if the System Operator intervenes or if a file of higher output priority becomes READY before the last copy is complete. This parameter is ignored for nonspooled output devices.

Default: The value is 1.

Bits (0:4) - Output Priority. For spooled devices only.

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

Default: 8.

filesize

double by value (optional)

A double-word integer specifying the maximum file capacity. For variable- and undefined-length records the capacity is expressed in blocks. For fixed-length records the capacity is expressed in logical records. The maximum file size is 2^{21} 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 disc files is set to the maximum physical size of the disc 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 disc sectors) that can be dynamically allocated to the file as logical records are written to it. The size of each extent is always calculated in terms of physical records. When the file is type F (fixed) *filesize* is the number of logical records; thus it will be divided by the *blockfactor* to determine the number of physical records (blocks). If the file is variable length or undefined length, *filesize* is the number of physical records. Then, the number of physical records required for the system file label, and user labels (if any), are added to the number of physical records required for data. To determine extent size, this number is divided by the requested *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 disc 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 disc, a single disc device class or a logical device number must be used in the *device* parameter.

initialloc

integer by value (optional)

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

Default: 1 extent.

filecode

integer by value (optional)

An integer recorded in the file label and made available for general use to anyone accessing the file through the FGETINFO intrinsic. For this parameter, any user can specify a positive integer ranging from 0 to 1023. This parameter is used for new files only when it is positive and the process is running in user mode. If your process is running in Privileged Mode, you can specify a negative integer for *filecode* when initially opening a file. Then, any future accesses of the "privileged" file must be requested in Privileged Mode. A process running in user mode cannot open a file that has a negative *filecode*. Also, if the process supplies a non-zero parameter, the *filecode* must match the one originally specified for the file. The following integers have meanings predefined by Hewlett-Packard:

Integer	Mnemonic	Meaning
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	XLSAV	Cross Loader ASCII File (SAVE)
1041	XLBIN	Cross Loader Relocated Binary File
1042	XLDSP	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 WorkFile
1059	TDPXQ	TDP WorkFile (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
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/3000 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 File

Integer	Mnemonic	Meaning
1156	DSTOR	RAPID/3000 DICTDBU Utility Store File
1157	TCODE	Code File for Transact/3000 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
1168	MARPA	ARPA Messages File
1169	MARPD	ARPA Distribution List
1170	MCMND	HPDESK Abbreviated Commands File
1171	MFRTM	Reserved
1172		Reserved
1173	MEFT	Reserved
1174	MCRPT	Reserved
1175	MSERL	Reserved
1176	VCSF	Reserved
1177	TTYPE	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
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	Revised Form DCA Document
1271	FFDCA	Final Form DCA Document
1272	DIU	Document Interchange Unit
1273	PDOC	HPWORD/150 Document
1401	CWPTX	Reserved
1421	MAP	HPMAP/3000 Map Specification File
1422	GAL	Reserved
1425	TTX	Reserved
3333		Reserved

Default is the unreserved file code of 0.

CONDITION CODES

CCE	Request granted. The file is open.
CCG	Not returned by this intrinsic.
CCL	Request denied. This may be because another process already has EXCLUSIVE or semi-exclusive access for this file, or an initial allocation of disc space cannot be made due to lack of disc space. The file number value returned by FOPEN is 0 if the file is not opened successfully. The FCHECK intrinsic should be called for more details.

ADDITIONAL DISCUSSION

"File Device Relationships" and "How to Use Files" in Section IV.

FPARSE

NO INTRINSIC NUMBER ASSIGNED

Parses and validates file designators. (Available version G.02.00 or later.)

```
BA      IA      LA      O-V
FPARSE (string,result,items,vectors);
```

FPARSE is called to ensure that the formal file designator is syntactically correct. It eliminates the need to write your own modules to parse and validate file names. FPARSE also eliminates any ambiguities about the correct syntax of the file designators and presents a more flexible and modular programming approach. It is recommended that all new programs use FPARSE to validate and parse file designators and that many of the existing application programs and subsystems enhance their code to use FPARSE.

PARAMETERS

string

byte array (required)

A buffer which contains the file reference string to be parsed. The string can be delimited by any nonalphanumeric character except a slash (/), period (.), or colon (:).

result

integer array (required)

A two-word array. The first 16-bit word of the array will contain the value indicating the result of the parse. If the value is positive, the file *string* is syntactically correct and the value indicates the type of file reference being made. The second word is reserved for future use. The following values are valid:

0 - Regular file designator.

1 - Backreference ("*" is the first character in *string*).

2 - System file (" \$" is the first character in *string*).

If negative, it is one of the following error codes, indicating a syntax error in the file reference:

Error# Meaning

- | | |
|----|--|
| -1 | Bad item values. |
| -2 | Parameter bounds violation. |
| -3 | Illegal delimiter; misuse of "." "/" or ":". |
| -4 | User specified only one of items or vectors array. |
| -5 | Illegal item value in items array. |

Error# Meaning

- 6 Item list not terminated by the 0 terminator.
- 7 Undefined system file.
- 8 Lockword disallowed in backreferenced (*) file.
- 9 NS/3000 not present, but user specified *envid*.
- 101 First character of the file name not alpha.
- 102 File name expected in the string.
- 103 File name identifier too long.
- 104 First character of lockword not alpha.
- 105 Lockword expected in the string.
- 106 Lockword identifier too long.
- 107 First character of the group name not alpha.
- 108 Group name expected in the string.
- 109 Group name identifier too long.
- 110 First character of the account name not alpha.
- 111 Account name expected in the string.
- 112 Account name identifier too long.
- 113 First character of *envidname* not alpha.
- 114 The *envidname* expected in the string.
- 115 Identifier for *envidname* too long.

System-defined designators, i.e. file names starting with "\$", are part of the file designator extension; a file name starting with ":" is not part of the file designator extension and is considered an illegal delimiter.

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

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

In case of an error the first element of the *vectors* array returns the byte offset of the invalid item in *string*. The second word will be zero.

There may be up to 3 identifiers separated by a period (.) in an *envidname*. Therefore, each of the errors, -113 through -115, may be referring to one of the 3 identifiers. The error pointer will point to the location of the error.

If the NS/3000 subsystem is not installed on the system, FPARSE will return error code -9 for file designators with *envid* after the file name. This error implies that everything else, up to the point where *envid* was detected, is valid. However, FPARSE will not return a parsed *vectors* array.

items

logical array (optional)

This array contains the item code, one-per-word, categorizing to what the corresponding *vectors* array should point:

0 - End of item list in array.

1 - File name.

2 - Lockword.

3 - Group name.

4 - Account name.

5 - NS/3000 *envid* name.

vectors

double array (optional)

An array which contains the vector element, one per double-word, for the requested items. The first word will contain the byte offset, from the base of the string array to the start of the item name, and the second word will contain the length. A length of zero will indicate that the item was not specified in the string array. There will be a one-to-one mapping with the *items* array, i.e. one *items* array word to one *vectors* array double-word. For Item 0, which terminates the *items* list, the corresponding entry in the *vectors* array will contain the total length of the file designator *string* in the second word and the first word will contain the default designator type (as in bits (10:3) of the FOPEN *options*) if the result parameter indicates a system file.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Parsing and Validating File Designators" in Section IV.

NS/3000 User/Programmer Reference Manual (32344-90001).

Sets the logical record point for a disc file.

SYNTAX

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

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

PARAMETERS

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

recnum *double by value (required)*
A positive double integer representing the relative logical record (or block number for a NOBUF file) at which the logical record pointer is to be positioned. The number of the first first logical record is zero.

On disc files, the end-of-file 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 point position is unchanged because of one of the following: <ul style="list-style-type: none">• The <i>recnum</i> was <0.• Invalid <i>filenum</i> parameter.• Input/output is pending on a NOWAIT request.• The file is spooled or is not a direct-access disc file.• The file does not contain fixed-length or undefined-length records.• Not allowed with append access.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Not applicable to message files.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

Reads a logical record or portion of a record from a file to the user's stack.

SYNTAX

```

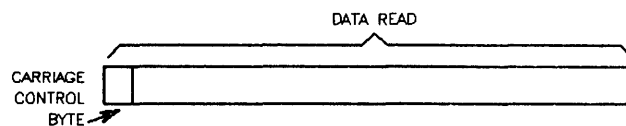
      I      IV      LA      IV
lgth:=FREAD(filenum,target,tcount);
  
```

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

When the logical end-of-data is encountered during reading, CCG is returned to the user process. On magnetic tape, the end-of-data can be denoted by a physical indicator such as a tape mark. When a file is read that spans more than one volume of labeled magnetic tape, the user program is suspended until the operator has completed mounting the next tape. CCG is not returned when end-of-tape is encountered. On disc, the end-of-data occurs when attempting to read beyond 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 MPE commands, the end-of-data is indicated when an :EOD command is encountered, but the :EOD command itself is not returned to the user. The end-of-data is indicated by a hardware end-of-file, including :EOF;; on \$STDIN by any record beginning with a colon; or on \$STDINX by :EOD. In addition, on the standard input device for a job, as opposed to a session, :JOB, :EODJ, or :DATA indicate end-of-data.

When a message file is empty and there are no writers, the process will wait if there is an FCONTROL 45 in effect. It will also wait 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.

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



FUNCTIONAL RETURN

lgth

integer (optional)

An integer value showing the length of the information transferred. If the *tcount* parameter in the FREAD call is positive, the positive value returned represents a word count; if the *tcount* parameter is negative, the positive value returned represents a byte count. FREAD always returns zero if NOWAIT I/O is specified. In this case, the actual record length is returned in the *tcount* parameter of the INWAIT intrinsic.

PARAMETERS

filenum

integer by value (required)

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

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

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

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

CONDITION CODES

CCE	The information was read.
CCG	The logical end-of-data was encountered during reading. When reading a labeled magnetic tape file that spans more than one volume, CCG is not returned when end-of-tape (EOT) is encountered. Instead, CCG is returned at actual end-of-file, with a transmission log of 0 if an attempt is made to read past end-of-file.
CCL	The information was not read because an error occurred, a terminal read was terminated by a special character or time-out interval as specified in the FCONTROL intrinsic, or a tape error was recovered and the FSETMODE option was enabled.

NOTE

The condition codes should be checked both in normal I/O and in NOWAIT I/O.

SPECIAL CONSIDERATION

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Using FREAD and FWRITE with \$STDIN and \$STDLIST" in Section IV.

FREADBACKWARD

NO INTRINSIC NUMBER ASSIGNED

Reads a logical record backward from the current record pointer. Data is presented to the user as if read forward.

SYNTAX

I	IV	LA	IV
<i>lgth</i> :=FREADBACKWARD(<i>filenum</i> , <i>target</i> , <i>tcount</i>);			

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

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

1. It may only be used with magnetic tape drives on HP-IB Systems.
2. The magnetic tape must be accessed NOBUF.

FUNCTIONAL RETURN

lgth

integer (optional)

An integer value showing the length of the information transferred. If the *tcount* parameter in the FREADBACKWARD call was positive, the positive value returned represents a word count; if the *tcount* parameter was negative, the positive value returned represents a byte count. FREADBACKWARD always returns zero if NOWAIT I/O is specified. In this case, the actual record length is returned in the *tcount* parameter of the IOWAIT intrinsic.

PARAMETERS

filenum

integer by value (required)

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

target

logical array (required)

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

tcount

integer by value (required)

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

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

CONDITION CODES

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

NOTE

The condition code should be checked both in normal I/O and in NOWAIT I/O.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

Reads a specific logical record or portion of a record from a direct-access disc file to the user's data stack.

SYNTAX

IVLAIVDV

```
FREADDIR(filenum,target,tcount,recnum);
```

The FREADDIR intrinsic reads a specific logical record, or a portion of such a record, from a disc file to the user's data stack. This intrinsic differs from the FREAD intrinsic in that the FREAD intrinsic reads only the record already pointed to by the logical record pointer. The FREADDIR intrinsic may be issued only for direct-access disc files composed of fixed-length records. If RIO access is used, FREADDIR will input the specified logical record. If the record is inactive, the contents of the inactive record will be transmitted and a CCE will be returned. In this case there is no indication whether the block contains some inactive records. (FCHECK returns a nonzero error number to distinguish active and inactive records. If a RIO file is accessed using the nonRIO method, (NOBUF) FREADDIR will input 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 *tcount* parameter specified is greater than one logical record. For example, if you read all of record 11 and half of record 12 in a file, the logical record pointer is set to the beginning of record 13 after the FREADDIR intrinsic executes. Thus the second half of record 12 is skipped.

PARAMETERS

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

recnum

double by value (required)

A double-word integer indicating the relative number, in the file, of the logical record or block number for NOBUF files to be read. The first record is indicated by "0D" (double-word zero in SPL notation).

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Not applicable to message files.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

Reads a user file label.

SYNTAX

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

The FREADLABEL intrinsic reads a user-defined label from a disc file or magnetic tape file. Once a disc file has been opened, user labels may be read from or written to in any order at any time, regardless of the opener's access to the rest of the file. A disc file can have up to 254 128-word user labels. A magnetic tape file, if labeled at all, must be labeled with an ANSI-standard or IBM-standard label. MPE automatically skips over any unread user labels when the first FREAD intrinsic call is issued for files. Therefore for labeled tape files, the FREADLABEL intrinsic should be called immediately after the FOPEN intrinsic has opened the file. The user-defined label must be 40 words in length to conform to the length of the ANSI or IBM-standard label.

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file whose label is to be read.
<i>target</i>	<i>logical array (required)</i> An array in the stack to which the label is to be transferred. This array should be large enough to hold the number of words specified by <i>tcount</i> .
<i>tcount</i>	<i>integer by value (optional)</i> An integer specifying the number of words to be transferred from the label. The limit is 128 words. Default: 128 words.
<i>labelid</i>	<i>integer by value (optional)</i> An integer specifying the label number. For labeled tapes <i>labelid</i> is ignored. The next sequential label is read. Default: Zero is assigned.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

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

SYNTAX

<div style="text-align: center;">IV DV</div> <div>FREADSEEK(<i>filenum</i>,<i>recnum</i>);</div>

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

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

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file to be read.
<i>recnum</i>	<i>double by value (required)</i> A double-word integer in SPL notation indicating the relative number of the logical record to be read. The first record is indicated by "0D" (double-word zero in SPL notation).

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Not applicable to message files.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

FREEDSEG

INTRINSIC NUMBER 131

Releases an extra data segment.

SYNTAX

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

A process can release an extra data segment assigned to it by using the FREEDSEG intrinsic. If this is a private data segment, or if it is a sharable (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 which is currently assigned to any other process, it is deleted from the calling process but continues to exist in the job/session.

PARAMETERS

<i>index</i>	<i>logical by value (required)</i> A word containing the logical index assigned to the data segment, obtained from the GETDSEG intrinsic call.
<i>id</i>	<i>logical by value (required)</i> The identity assigned to the segment. If none is assigned, enter a zero.

CONDITION CODES

CCE	Request granted. The data segment is deleted from the job/session.
CCG	Request granted. The data segment is deleted from the calling process but continues to exist in the job/session because it is being shared by another process.
CCL	Request denied. Either the <i>index</i> is invalid or <i>index</i> and <i>id</i> do not specify the same extra data segment.

SPECIAL CONSIDERATIONS

Data Segment Management (DS) capability required.

ADDITIONAL DISCUSSION

The GETDSEG intrinsic in this section, and "Deleting an Extra Data Segment" in Section III.

Frees all local Resource Identification Numbers (RINs) from allocation to a job.

SYNTAX

```
FREELOCRIN;
```

The FREELOCRIN intrinsic frees all local Resource Identification Numbers (RINs) currently reserved from your job.

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

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Resource Management" in Section III.

FRELATE

INTRINSIC NUMBER 18

SYNTAX

```

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

```

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

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

The FRELATE intrinsic applies to files on all devices.

FUNCTIONAL RETURN

intordup *logical (optional)*
A word indicating whether the two files referenced are interactive and/or
duplicative. It contains two bits of importance:

Bit (15:1)

=0 The *infilenum* and *listfilenum* do not form an interactive pair.

=1 The *infilenum* and *listfilenum* form an interactive pair.

Bit (0:1)

=0 The *infilenum* and *listfilenum* do not form a duplicative pair.

=1 The *infilenum* and *listfilenum* form a duplicative pair.

PARAMETERS

<i>infilenum</i>	<i>integer by value (required)</i> A word supplying the file number of the input file.
-------------------------	---

<i>listfilenum</i>	<i>integer by value (required)</i> A word supplying the file number of the list file.
---------------------------	--

CONDITION CODES

CCE	Request granted.
CCG	Request denied because <i>infilenum</i> and/or <i>listfilenum</i> corresponds to \$NULL. \$NULL is considered to be a logical file which contains no data. No data can be read from this file and all data written to it is discarded. The <i>infilenum</i> and <i>listfilenum</i> functions, therefore, are illogical for the \$NULL file. Interactive or duplicative functions do not apply.
CCL	Request denied because an error occurred.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

FRENAME

INTRINSIC NUMBER 17

Renames a disc file.

SYNTAX

IV BA

FRENAME(*filenum*,*newfilereference*);

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

The file to be renamed must be either:

- A new file.
- An existing file, opened for EXCLUSIVE access, for which you have WRITE access (specified by the security provisions of the file). If the file is a permanent file, you must be the creator.

PARAMETERS

filenum

integer by value (required)

A word supplying the file number of the file to be renamed.

newfilereference

byte array (required)

Contains an ASCII string specifying the new name of the file. The maximum number of characters allowed in the string is 36. The format of *newfilereference* is:

filename/lockword.group.account

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

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

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

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

The ASCII string contained in *newfilereference* must begin with a letter; can contain up to eight alphanumeric characters for each of the *filename*, *lockword*, *group*, and *account* fields. The string must end with a nonalphanumeric character including a blank, but not a slash (/) or a period (.) since these are used as field delimiters within the string.

CONDITION CODES

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

ADDITIONAL DISCUSSION

None.

FSETMODE

INTRINSIC NUMBER 14

Activates or deactivates the file access modes.

SYNTAX

IVLV

```
FSETMODE(filenum,modeflags);
```

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 disc 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 disc write completion until the physical write operation has completed. Therefore, the BLOCKONWRITE option guarantees commitment of the transaction to disc 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 disc caching with minimal performance impact. Many disc writes can be performed with BLOCKONWRITE=NO, Serial Write Queue enabled, and guarantee integrity. On the final disc 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.

PARAMETERS

filenum

integer by value (required)

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

modeflags

logical by value (required)

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

Bit (15:1) - Serial Write Queue. This bit may be set as follows:

=0 Disabled. Serial I/O is not guaranteed.

=1 Enabled. All physical I/O performed for this file will be guaranteed to be output to disc in the order in which the I/Os were sent. This bit is effective only on systems with disc caching enabled. With disc caching, it is possible for I/Os to be posted in a different order than sent.

Bit (14:1) - BLOCKONWRITE. This bit has the following settings:

=0 Disabled. Output is not verified.

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

Bit (13:1) - Terminal control by the user.

This bit has the following settings:

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

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

Carriage return, line feed is not issued in the case where FREAD *filenum*, *target*, or *tcount* is satisfied (*tcount* characters are typed in). If **RETURN** is pressed, however, a carriage return is echoed but no line feed is sent. This also applies to the READ intrinsic.

Bit (12:1) - Tape error recovery.

This bit may be set as follows:

=0 Tape error and report condition code CCE.

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

The remaining bits (0:12) are reserved for MPE and must be set to zero.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Declaring Access-Mode Options" in Section IV.

MPE File System Reference Manual (30000-90236).

MPE V System Operation and Resource Management Reference Manual (32033-90005).

Point-to-Point Workstation I/O Reference Manual (30000-90250).

Moves a physical record pointer forward or backward on a tape or disc file.

SYNTAX

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

You can space forward or backward on a fixed-length or undefined-length file by using the FSPACE intrinsic to reset the logical record pointer. The FSPACE intrinsic applies to files on disc and magnetic tape devices (including serial discs) only. On magnetic tape devices FSPACE spaces physical rather than logical records.

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

Refer to the MPE File System Reference Manual (30000-90236) for special considerations on magnetic tape files.

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file on which spacing is to be done.
<i>displacement</i>	<i>integer by value (required)</i> A signed integer indicating the number of logical records for buffered disc files, or blocks for NOBUF files and all tape files, to be spaced over, relative to the current position of the logical record pointer. A positive value signifies forward spacing, a negative value signifies backward spacing. The maximum positive value is 32767, the maximum negative value is -32768. If RIO access is used, the <i>displacement</i> includes all records regardless of activity state (that is, active or deleted). Attempts to backspace beyond the beginning of the file will be ignored by the system. The logical record pointer will point to record 0 (the first record) and no error codes will be returned.

CONDITION CODES

CCE	Request granted.
CCG	An end-of-file indicator was encountered during spacing. For disc files, this is the file limit, and the logical record pointer is not changed. For magnetic tape files, it is the end-of-file mark, and the logical record pointer points to the (logical) end-of-file. The magnetic tape, however, is positioned to one record past the file mark on the tape. For labeled tape the logical record pointer is at the file mark.
CCL	Request denied because an error occurred; for example, the file resides on a device that prohibits spacing. Not allowed with APPEND access.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

Dynamically unlocks a file.

SYNTAX

IV

FUNLOCK(*filenum*);

The FUNLOCK intrinsic dynamically unlocks a file that has been locked with the FLOCK intrinsic.

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i>
	A word supplying the file number of the file to be unlocked.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

FUPDATE

INTRINSIC NUMBER 4

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

SYNTAX

IV LA IV

FUPDATE(*filenum*,*target*,*tcount*);

The FUPDATE intrinsic updates a logical record in a disc file. This intrinsic affects the logical record (or block for NOBUF files) last referenced by any intrinsic call for the file named except for FPOINT which affects the record prior to the last record referenced. FUPDATE moves the specified information from the user's stack into this record. The file containing this record must have been opened with the update *aoption* specified in the FOPEN call, and must not have variable-length records. If RIO access is used, the modified record is set to the active state.

FUPDATE is functionally equivalent to, but faster than, FSPACE(*filenum*, -1); followed by an FWRITE to *filenum*.

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file to be updated.
<i>target</i>	<i>logical array (required)</i> Contains the record to be written in the updating.
<i>tcount</i>	<i>integer by value (required)</i> An integer specifying the number of words or bytes to be written from the record. If this value is positive, it signifies words; a negative value signifies bytes; and a zero indicates that no transfer occurs. To have all contents of a record written, <i>tcount</i> must be set at \geq RECSIZE.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Not applicable to message files.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

FWRITE

INTRINSIC NUMBER 3

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

SYNTAX

IV	LA	IV	LV
FWRITE(<i>filenum</i> , <i>target</i> , <i>tcount</i> , <i>control</i>);			

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

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

When the FWRITE intrinsic is executed, the logical record pointer is set to the record immediately following the record just written, or the first logical record in the next block for NOBUF files. If RIO access is used, the modified record is set to the active state.

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

If a magnetic tape or serial disc is unlabeled (as specified in the FOPEN intrinsic or :FILE command) and a user program attempts to write over or beyond the physical or simulated end-of-tape (EOT) marker, the FWRITE intrinsic returns CCL. The actual data has been written to the tape, and a call to FCHECK reveals a file error indicating end-of-tape. All writes to the tape after the EOT tape marker has been crossed transfer the data successfully but return CCL until the tape crosses the EOT marker again in the reverse direction (rewind or backspace).

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

For message files and circular files this intrinsic logically appends the user's record to the end of the file. If 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 will wait if there is an FCONTROL 45 in effect. It will also wait if this the first FWRITE after the writer's FOPEN. Otherwise, CCG is returned.

PARAMETERS

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

target *logical array (required)*
Contains the record to be written.

tcount

integer by value (required)

An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; a negative value, bytes; and a zero indicates that no transfer occurs. If *tcount* is less than the *recsize* parameter associated with the record, only the first *tcount* words or bytes are written.

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

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

control

logical by value (required)

A logical value representing a carriage control code, effective if the file is transferred to a line printer or terminal (including a spooled file whose ultimate destination is a line printer or a terminal). This parameter is effective only for files opened with carriage control specified.

The options are:

- 0 Print the full record transferred, using single spacing. This results in a maximum of 132 characters per printed line.
- 1 Use the first character of the data written to satisfy space control, and suppress this character on the printed output. This results in a maximum of 132 characters of data per printed line. Permissible control characters are shown in Table 2-5.

Any octal code from Table 2-5 can be used to determine space control and print the full record transferred. This results in a maximum of 132 characters per printed line.

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

The effect of the FWRITE *control* parameter in combination with the FOPEN carriage control *foption* (or overriding :FILE command CCTL/NOCCTL parameter) upon the data written is summarized in Figure 2-3.

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

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

For non-spoiled devicefiles, all of the Carriage Control Codes listed in Table 2-5 may be used as the value of the *param* parameter in FCONTROL (when *controlcode* = 1), regardless of whether the file is opened with CCTL or NOCCTL. When the file is opened with CCTL, these carriage control codes may be used in either of the following ways via FWRITE:

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

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

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

Point-to-Point Workstation I/O Reference Manual (30000-90250).

Table 2-5. Carriage Control Directives

OCTAL CODE	ASCII SYMBOL	CARRIAGE ACTION
%40	" "	Single space (with or without automatic page eject).
%53	"+"	No space, return (next printing at column 1). This cannot be used more than once on the HP 2608A/S without losing data.
%55	"_"	Triple space (with or without automatic page eject).*
%60	"0"	Double space (with or without automatic page eject).*
%61	"1"	Conditional page eject (form feed) is performed by the software. If the printer is not already at the top of the form, a page eject is performed. Ignored if: Post-space mode: The current request has a transfer count of 0 and the previous request was FOPEN, FCLOSE, or FWRITE which specified a carriage control directive of %61. Pre-space mode: Both the current request and the previous request have transfer counts of 0, and the current request and previous request are any combination of FOPEN, FCLOSE, or FWRITE specifying a carriage control of %61.
%62		Skip to one line before top of form. This specification is valid only for the HP 2608S and 2563A printers.
%63		A conditional page eject form (form feed) is performed by the printer. If the printer is not already at the top of form, perform a page eject. This specification is valid only for the HP 2608S and 2563A printers.
%2nn		Space <i>nn</i> lines (no automatic page eject); <i>nn</i> 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).

*Note: If these codes are selected with automatic page eject in effect (by default or following an octal code of %102 or %402), the resulting skip is to a location absolute to the page. A code of %60 is replaced by %303, and a code of %55 is replaced by %304. Thus, the resulting skip may be less than two or three lines, respectively.

If automatic page eject is not in effect, a true double or triple space results, but the perforation between pages is not automatically skipped. For the HP 2608S and 2563A, if auto-eject and feature mode are in effect, a code of %60 will be replaced by two codes of %302, and a code of %55 is replaced by three codes of %302. The resulting skip will be double or triple space with auto-eject, respectively.

Table 2-5. Carriage Control Directives (Continued)

OCTAL CODE	ASCII SYMBOL	CARRIAGE ACTION
NOTE: Channel assignments shown below are the Hewlett-Packard standard defaults.		
%300		Skip to top of form (page eject).
%301		Skip to bottom of form.
%302		Single spacing with automatic page eject.
%303		Skip to next odd line with automatic page eject.
%304		Skip to next third line with automatic page eject.
%305		Skip to next 1/2 page.
%306		Skip to next 1/4 page.
%307		Skip to next 1/6 page.
%310		Skip to bottom of form.
%311		User option (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.
%320		No space, no return (next printing physically follows this).
%2 - %37		
%41 - %52		
%54		
%56-%57		Same as %40.
%62-%77		
%104-%177		
%310-%317 (HP 2607)		
%314-%317 (HP 2613/17/18/19)		
%321-%377		
%400 or %100		Sets post-space movement option; this first prints, then spaces. If previous option was pre-space movement, the driver outputs a line with a skip to VFC Channel 3 (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 pre-space movement option; this first spaces, then prints.
%402 or %102		Sets single-space option, with automatic page eject (60 lines per page).
%403 or %103		Sets single-space option, without automatic page eject (66 lines per page).

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

EFFECT ON DATA OUTPUT

Figure 2-3. Carriage Control Summary

FWRITEDIR

INTRINSIC NUMBER 8

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

SYNTAX

IVLAIVDV

FWRITEDIR

filenum, *target*, *tcount*, *recnum*

);

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

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

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

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

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

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

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file to be written on.
<i>target</i>	<i>logical array (required)</i> Contains the record to be written. This array should be large enough to hold all of the information to be transferred.
<i>tcount</i>	<i>integer by value (required)</i> An integer specifying the number of words or bytes to be written to the record. If this value is positive, it signifies words; a negative value, bytes; and a zero indicates that no transfer occurs. If <i>tcount</i> is less than the <i>rec-size</i> parameter associated with the record, and NOBUF was specified, only the first <i>tcount</i> words or bytes are written.

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

recnum

double by value (required)

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

CONDITION CODES

CCE	Request granted.
CCG	The physical end-of-file was encountered.
CCL	Request denied because an error occurred.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Not applicable to message files.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

FWRITELABEL

INTRINSIC NUMBER 20

Writes a user file label.

SYNTAX

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

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

PARAMETERS

<i>filenum</i>	<i>integer by value (required)</i> A word supplying the file number of the file to be labeled.
<i>target</i>	<i>logical array (required)</i> Contains the label to be written. If the file is a labeled magnetic tape file, this label must be 40 words in length.
<i>tcount</i>	<i>integer by value (optional)</i> The number of words to be transferred from the array. The default is 128 words.
<i>labelid</i>	<i>integer by value (optional)</i> The number of the label to be written. The first label is 0. This parameter is ignored for labeled tapes. The next sequential tape label is written. Zero is assigned as the default.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

MPE File System Reference Manual (30000-90236).

GENMESSAGE

NO INTRINSIC NUMBER ASSIGNED

Accesses the MPE message system.

SYNTAX

```
      I      0-V      IV      IV      IV      BA      IV      LV
msglen:=GENMESSAGE(filenum,setnum,msgnum,buff,buffsize,paramask,
                  LV      LV      LV      LV      LV      IV      I
                  param1,param2,param3,param4,param5,msgdest,errnum);
```

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

NOTE

The catalog file must be opened with *foptions* "old, permanent, ASCII" (*foptions* 5), and *aoptions* "NOBUF and MULTirecord access" (*aoptions* %420).

FUNCTIONAL RETURN

msglen *integer (optional)*
The length of the message is returned (in bytes).

PARAMETERS

filenum *integer by value (required)*
A word supplying the file number of the message catalog.

setnum *integer by value (required)*
A positive integer no greater than 62 specifying the message set number within the catalog.

msgnum *integer by value (required)*
A positive integer, specifying the message number within the message set.

buff *byte array (optional)*
A byte array to which the assembled message is returned.

Default: Message is not returned to calling program.

buffsize *integer by value (optional)*
When *buff* is specified, *buffsize* is the size, in bytes, of the buffer. When *buff* is not specified, *buffsize* is the length, in bytes, of the records written to the destination file.

Default: 72 bytes.

paramask *logical by value (optional)*
A 16-bit logical mask indicating parameter types for *param1*, *param2*, *param3*, *param4*, and *param5*. The bit settings are as follows:

Bits (13:3) - Defines *param5* type.

=000 Parameter is a string, terminated by an ASCII null (0).

=001 Parameter is an integer.

=010 Parameter is double by reference.

=011 Ignore the parameter.

Bits (10:3) - *Param4* type (types same as for *param5*).

Bits (7:3) - *Param3* type (types same as for *param5*).

Bits (4:3) - *Param2* type (types same as for *param5*).

Bits (1:3) - *Param1* type (types same as for *param5*).

Bit (0:1)

=1 Ignore rest of word and parameters *param1* through *param5*.

=0 Rest of word, in 3-bit groupings, will specify parameter types for *param1*, *param2*, *param3*, *param4*, and *param5*.

Default: Parameters *param1* through *param5* will be ignored.

param1 *logical by value (optional)*
Parameter to be inserted into message. If *paramask* specifies type 0 (string), *param1* must pass the byte address (that is, *@stringarray*) of the byte array containing the string. If *paramask* specifies type 2 (double by reference), *param1* must pass the word address (that is, *@doublename*) of the double-word identifier containing the value.

param2 *logical by value (optional)*
Parameter to be inserted into message. Description is the same as for *param1*.

param3 *logical by value (optional)*
Parameter to be inserted into message. Description is the same as for *param1*.

param4 *logical by value (optional)*
Parameter to be inserted into message. Description is the same as for *param1*.

param5 *logical by value (optional)*
Parameter to be inserted into message. Description is the same as for *param1*.

msgdest *integer by value (optional)*
Integer value specifying the destination of the assembled message. Enter the file number of the destination file if the file is not \$STDLIST; for \$STDLIST, enter 0.

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

errnum *integer (optional)*
Integer identifier to which an error number is returned. Values returned are as follows:

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

CONDITION CODES

CCE	Successful execution.
CCL	Intrinsic did not execute because of file system error.
CCG	Intrinsic did not execute. May have missing required parameter, invalid parameter, or invalid file number of catalog or destination file. CCG is also returned if <i>setnum</i> or <i>msgnum</i> is not found.

ADDITIONAL DISCUSSION

"Using GENMESSAGE to Insert Parameters in Messages" in Section V.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

Creates an extra data segment.

SYNTAX

L I LV

`GETDSEG(index,length,id);`

The GETDSEG intrinsic creates or acquires an extra data segment. The number of extra data segments that can be requested, and the maximum size allowed these segments, are limited by parameters specified when the system is configured. When an extra data segment is created, the GETDSEG intrinsic returns a logical index number to the calling process. This index number is assigned by MPE and allows this process to reference the segment in later intrinsic calls.

The GETDSEG intrinsic is also used to assign the segment the identity that either allows other processes in the job or session to share the segment, or that declares it private to the calling process. If the segment is sharable, other processes can obtain its logical index (through GETDSEG) and use this index to reference the segment. Thus, the logical index is a local name that identifies the segment throughout any process that obtained the index with the GETDSEG call. The logical index need not be the same value in all processes sharing the data segment. The identity, on the other hand, is a job-wide or session-wide name that permits any process to determine the logical index of the segment. If the intrinsic is called in User Mode, then the data segment is initially filled with zeros.

When GETDSEG is called in User Mode, all subsequent calls to intrinsics that use *index* must now be in User Mode. When GETDSEG is called in Privileged Mode, all subsequent calls to intrinsics that use *index* must now be in Privileged Mode (version G.01.00 or later).

PARAMETERS

index

logical (required)

A word to which the logical index of the data segment, assigned by MPE, is returned. When GETDSEG is called in User Mode, *index* is a logical index of the assigned data segment; if an error is found, *index* will be set to %2000-%2004. When GETDSEG is called in Privileged Mode, *index* is the actual Data Segment Table (DST) entry number for the data segment that was assigned.

length

integer (required)

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

id

logical by value (required)

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

CONDITION CODES

CCE	Request granted. A new segment was created.
CCG	Request granted. An extra data segment with this identity exists already.
CCL	Request denied. An illegal <i>length</i> was specified (<i>index</i> is set to %2000). The process requested more than the maximum allowable number of data segments (<i>index</i> is set to %2001). Sufficient storage was not available for the data segment (<i>index</i> is set to %2002). A stack expansion necessary to satisfy the request could not be done because the stack was frozen (<i>index</i> set to %2003). A stack expansion is usually not necessary to get an extra data segment. There is not enough room in the job definition table to make an entry for the extra data segment (<i>index</i> set to %2004).

SPECIAL CONSIDERATIONS

Data Segment Management (DS) capability required.

If the *index* parameter will be used with the SWITCHDB intrinsic, GETDSEG must be called in Privileged Mode (PM).

ADDITIONAL DISCUSSION

"Creating an Extra Data Segment" in Section III.

Retrieves the *info* string and *parm* value from the :RUN command or CREATEPROCESS intrinsic. (Available G.02.00 version or later.)

SYNTAX

```
0-V      BA      I      I
GETINFO (info,info'length,parm);
```

The GETINFO intrinsic provides user programs with the ability to retrieve the *info* string and the *parm* value entered in the :RUN command, or given in the CREATEPROCESS intrinsic. The *info* string is returned in the byte array *info* and the *parm* value is returned in *parm*. If the *info* string is longer than the length specified by *info'length* only the amount of the string up to the *info'length* is returned.

FUNCTIONAL RETURN

result

Integer. The possible values returned are:

- 0: The intrinsic executed successfully.
- 1: Error. This occurs when the *info* array is passed in, but the length is invalid or omitted.

PARAMETERS

info

byte array (optional)

Specifies where the *info* string is to be returned. If not provided, the *info* string will not be returned.

info'length

integer (optional)

This parameter both passes and receives a length. When calling, this should contain a positive integer specifying the length, in bytes, of *info*. This is only required if *info* is specified.

When returning from the call, the real length of the *info* string will be in *info'length*. When returning from the call, *info'length* will contain the smaller of either the real length of the *info* string or the original *info'length* value.

parm

integer (optional)

Specifies where the *parm* value will be returned.

CONDITION CODES

The condition code remain unchanged.

ADDITIONAL DISCUSSION

The CREATEPROCESS discussion in this section.

MPE V Commands Manual (32033-90006).

Returns the value of the system-defined Job Control Word JCW.

SYNTAX

```
  L  
JCW:=GETJCW;
```

The GETJCW intrinsic returns the value of the system-defined Job Control Word to the calling process.

FUNCTIONAL RETURN

JCW

logical (optional)

A logical variable containing the Job Control Word. This word is structured for a desired purpose by the calling program through the SETJCW intrinsic or PUTJCW intrinsic.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"User-Defined Job Control Words" in Section V.

GETLOCRIN

INTRINSIC NUMBER 30

Acquires local RINs.

SYNTAX

LV
GETLOCRIN(*rincount*);

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

PARAMETERS

<i>rincount</i>	<i>logical by value (required)</i> 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.
-----------------	--

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.

ADDITIONAL DISCUSSION

"Resource Management" in Section III.

Determines the source of an activation call for a process.

SYNTAX

I
source := GETORIGIN;

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

FUNCTIONAL RETURN

source

integer (optional)

This intrinsic returns one of the following codes:

- 0 Was not activated by the father nor the son.
- 1 Activated by the father.
- 2 Activated by a son.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Determining Source of Activation" in Section III.

GETPRIORITY

INTRINSIC NUMBER 120

Changes the priority of a process.

SYNTAX

0-V	IV	LV	IV
<code>GETPRIORITY(<i>pin</i>,<i>priorityclass</i>,<i>rank</i>);</code>			

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

NOTE

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

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

PARAMETERS

pin

integer by value (required)

An integer specifying the process whose priority is to be changed. If this is a son process, the integer is the Process Identification Number (PIN) of the process. If this is the calling process, the integer is zero.

priorityclass

logical by value (required)

A 16-bit word that contains two ASCII characters describing the priority class in which the process is rescheduled. This may be AS, BS, CS, DS, or ES. For users running in Privileged Mode, the *priorityclass* parameter may be specified as an absolute number by αA , where α is an 8-bit priority number and A is the ASCII character "A". For example, a request for a *priorityclass* of 31 in the master queue would be requested as %017501. Note that an absolute priority must be specified in order to overcome the MAXPRI setting of an account.

NOTE

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

rank

integer by value (optional)

This parameter is used only for the "AS" priority class. It will be added to the "AS" priority value if supplied and is ignored in all other cases.

CONDITION CODES

CCE Request granted.

CCG Request denied because the process specified does not exist.

CCL Request denied because an illegal PIN was specified.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Process Handling (PH) capability required.

Must be running in Privileged Mode to specify absolute priority.

ADDITIONAL DISCUSSION

"Rescheduling Processes" in Section III.

GETPRIVMODE

INTRINSIC NUMBER 200

Dynamically enters Privileged Mode.

SYNTAX

O-P
GETPRIVMODE;

The GETPRIVMODE intrinsic switches a temporarily privileged program from the non-Privileged Mode to the Privileged Mode. This intrinsic sets the Privileged Mode bit in the status register bit (9:1)=1, but leaves the Privileged Mode bit in the Code Segment Table (CST) entry for the executing segment unchanged. The status register, rather than the CST, determines a mode change when running in Privileged Mode. Thus, if additional segments are to be run as part of the program, they will be run in Privileged Mode unless GETUSERMODE is called specifically to return to the non-Privileged Mode. The calling process is aborted if the program file does not possess the Privileged Mode (PM) capability, and the CST indicates non-Privileged Mode.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Privileged Mode (PM) capability required.

CAUTION

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

ADDITIONAL DISCUSSION

"Entering Privileged Mode" in Section III.

Requests PIN of a son process.

SYNTAX

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

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

FUNCTIONAL RETURN

pin *integer (optional)*
The PIN of the specified son process.

PARAMETERS

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

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

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Determining Son Process" in Section III.

GETPROCINFO

INTRINSIC NUMBER 110

Requests status information about a father or son process.

SYNTAX

D

IV

statinfo := GETPROCINFO(*pin*);

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

FUNCTIONAL RETURN

statinfo

double (optional)

A double-word message denoting information about a father or son process.
The words contain information in the following manner:

Word 1:

Bits (8:8) - The priority number of the process in the master queue.

Bits (0:8) - Reserved for MPE.

These bits are set to zero by the system.

Word 2:

Bit (15:1) - Activity state.

This bit has the following settings:

=0 The process is suspended.

=1 The process is active.

Bits (13:2) - Suspension condition.

The following settings apply only if bit (15:1)=0:

=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 the son.

Bits (9:4) - Reserved for MPE.

These bits are set to zero by the system.

Bits (7:2) - Origin of the last ACTIVATE intrinsic call denoted by the following bit settings:

=00 The process was activated by MPE.

=01 The process was activated by the father.

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

CONDITION CODES

CCE

Request granted.

CCG

Request denied because the process is being terminated.

CCL

Request denied because an illegal PIN was specified.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Determining Process Priority and State" in Section III.

GETUSERMODE

INTRINSIC NUMBER 201

Dynamically returns a program to non-Privileged Mode.

SYNTAX

```
GETUSERMODE;
```

The GETUSERMODE intrinsic changes a temporarily privileged program from Privileged to non-Privileged Mode.

This intrinsic changes the Privileged Mode bit in the status register, bit (9:1)=0, and is the complement of the GETPRIVMODE intrinsic.

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Privileged Mode (PM) capability required.

ADDITIONAL DISCUSSION

"Entering Non-Privileged Mode" in Section III.

Initializes a USL file to the empty state.

SYNTAX

I IV I

errnum := INITUSLF(*uslfnum*, *rec0*);

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

FUNCTIONAL RETURN

errnum *integer (optional)*
If no error occurs, no value is returned. If an error occurs, one of the following is returned:

- 0 An unexpected end-of-file was encountered when writing to *uslfnum*.
- 1 Unexpected input/output error occurred.

PARAMETERS

uslfnum *integer by value (required)*
A word supplying the file number of the USL file.

rec0 *integer array (required)*
A 128-word buffer, corresponding to the first record of the USL file (record 0), to be initialized to the empty state. This buffer should be set to all zeros. The intrinsic will set certain values in record 0 before returning to the calling program.

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied; an error number is returned to <i>errnum</i> .

ADDITIONAL DISCUSSION

MPE Segmenter Reference Manual (30000-90011).

IODONTWAIT

NO INTRINSIC NUMBER ASSIGNED

Initiates completion operations for an I/O request.

SYNTAX

I	O-V	IV	LA	I	L
<i>fnum</i> := IODONTWAIT(<i>filenum</i> , <i>target</i> , <i>tcount</i> , <i>cstation</i>);					

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

FUNCTIONAL RETURN

fnum *integer (optional)*
An integer representing the file number for which the completion occurred. If no completion occurred, zero is returned.

PARAMETERS

filenum *integer by value (required)*
A word supplying the file number for which there is a pending I/O request. If zero is specified, the IODONTWAIT intrinsic will check for any I/O completion.

target *logical array (optional)*
A word pointer specifying the DB-relative address of the user's input buffer. This buffer must be large enough to contain the input record. It should be the same buffer specified in the original I/O request if that request was a read.

tcount *integer (optional)*
A word to which a positive integer representing the length of the received or transmitted record is returned. If the original request specified a byte count, the integer represents bytes; if the request specified words, the integer represents words. Note that this parameter is pertinent only if the original request was a read. The FREAD intrinsic always returns zero as its functional return if NOWAIT I/O is specified. In this case, the actual record length is returned in the *tcount* parameter of IODONTWAIT.

Default: The length of the record is not returned.

cstation *logical (optional)*
Used for distributed systems to return the number of the calling station when completed.

CONDITION CODES

CCE	Request granted. If the functional return is not zero then I/O completion occurred with no errors. A 0 returned to <i>fnum</i> indicates that no I/O has completed.
CCG	An end-of-file condition was encountered.
CCL	Request denied. Normal I/O completion did not occur because there were no I/O requests pending, a parameter error occurred, or an abnormal I/O completion occurred.

SPECIAL CONSIDERATIONS

You must be running in Privileged Mode to specify `FOPEN aoption NOWAIT I/O`.

ADDITIONAL DISCUSSION

"Using the IOWAIT Intrinsic" in Section IV.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

INTRINSIC NUMBER 22

SYNTAX

```

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

```

FUNCTIONAL RETURN

PARAMETERS

cstation

logical (optional)

Used for distributed systems to return the number of the calling station when completed.

CONDITION CODES

CCE Request granted. I/O completion occurred with no errors.

CCG An end-of-file condition was encountered.

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

SPECIAL CONSIDERATIONS

You must be running in Privileged Mode to specify FOPEN *aoption* NOWAIT I/O.

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Using the IOWAIT Intrinsic" in Section IV.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

JOBINFO

INTRINSIC NUMBER 180

Provides access to job/session related information. (Available on version G.01.00 or later).

SYNTAX

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

JOBINFO provides access to information related to any job/session that is current to the system. This intrinsic is expandable, and is written so the addition of further functions will be straightforward.

PARAMETERS

jsind

integer by value (required)

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

- 1 *JS#nnn* is a session.
- 2 *JS#nnn* is a job.

JS#nnn

double (required)

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

status

logical array (required)

A two-word logical array to report the overall success/failure of the call. Only the first word contains significant information. The success/failure of the call is indicated by the following returns:

- 0 Successful call. All *errornums* equal zero.
- 1 Semi-successful call. One or more *errornum(s)* were returned with nonzero values.
- 2 Unsuccessful call. All *errornums* were returned with nonzero values.
- 3 Unsuccessful call. Syntax error in calling sequence.
- 4 Unsuccessful call. Unable to retrieve *JS#nnn*.
- 5 Process terminated. The process terminated during the start of retrieval.

itemnum *integer by value (optional)*
Cardinal number of the item desired. This specifies which item value is to be returned (Refer to "Item#" in Table 2-6).

item *logical array (optional)*
Name of a reference parameter (whose data type corresponds to the data type for the desired information) to which the desired information is returned (Refer to "Item Value" in Table 2-6).

All possible *itemnum* and *item* parameters are output parameters with one exception. Item Number 1 can be used for an input or output parameter. Item Number 1 is an input parameter only if the user is identifying a job or session by passing a character string containing the logon ID, [*jsname*, *username.acctname*. Otherwise, it is an output parameter. The maximum number of characters returned is twenty-six. The returned string will be left-justified and padded with blanks.

errornum *integer (optional)*
A returned integer specifying the success or failure of the retrieval of each item. The returned values are:

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

SPECIAL CONSIDERATIONS

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

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Identifying a Job or Session with JOBINFO" in Section V.

Table 2-6. Item Values Returned By JOBINFO

ITEM#	ITEM VALUE (Information Returned)	DATA TYPE
1	[jsname,]user.account (See Note 1)	LA
2	Session/job name (See Note 2)	LA
3	User name (See Note 2)	LA
4	User logon group (See Note 2)	LA
5	User account (See Note 2)	LA
6	User home group (See Note 2)	LA
7	Session/job introduction time (See Note 3)	LA
8	Session/job introduction date (See Note 4)	LA
9	Input ldev/class name (See Note 2)	LA
10	Output ldev/class name (See Note 2)	LA
11	Current job step (See Note 5)	LA
12	Current number of active jobs	I
13	Current number of active sessions	I
14	Job input priority	I
15	Job/session number	D
16	JOBFENCE	I
17	Job output priority	I
18	Number of copies	I
19	Job limit (system)	I
20	Session limit (system)	I
21	Job deferred (See Note 6)	I
22	Main PIN - CI PIN for job/session	L
23	Original job-spoiled (See Note 6)	L
24	RESTART option (See Note 6)	L
25	Sequenced - job (See Note 6)	L
26	Term code (See Note 7)	L
27	CPU limit	L
28	Session/job state (See Note 8)	L
29	User's local attributes	L
30	\$STDIN spoolfile number (See Notes 9 & 10)	D
31	\$STDIN spoolfile status (See Notes 9 & 11)	I
32	\$STDLIST spoolfile number (See Notes 9 & 10)	I
33	\$STDLIST spoolfile status (See Notes 9 & 11)	I
34	Length of current job step of Item Number 11	I
35	:SET STDLIST=DELETE invoked (See Note 12)	L
36	Job Information Table data segment number	L

Table 2-6. Item Values Returned By JOBINFO (Continued)

1. Can be used as an input or output 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 of [jsname,]user.account. The wildcard character @ is not allowed. If used as an output parameter, the logical array must be 13 words long. Output is left-justified and padded with blanks.
2. An ASCII output parameter. Logical arrays must be four words long. Output is left-justified and padded with blanks.
3. Returns a 32-bit double-word in a form to be used by the FMTCLOCK intrinsic. If for a scheduled job, it will be the time when the job enters the WAIT state.
4. Returns a 16-bit logical word in a form to be used by the FMTCALNDAR intrinsic. If for a scheduled job, it will be the date when the job enters the WAIT state.
5. Returns a maximum of 283 ASCII characters, and is the image of the command currently executing. The logical array must be long enough to accommodate the expected command image.
6. Returns the values: 0 - No
1 - Yes
7. Returns the values: 0 - Regular terminal
1 - Regular terminal with special logon
2 - APL terminal
3 - APL terminal
8. Returns the values: 2 - Executing
4 - Suspending
32 - Wait
48 - Initialization
56 - Scheduled
9. Returns data for current jobs and sessions. \$STDIN/\$STDLIST files only.
10. Returns the spoolfile number as an integer.
11. Returns the values: 0 - Active
1 - Ready
2 - Open
3 - Reserved
12. Returns the values: 0 - \$STDLIST will be saved
1 - :SET STDLIST=DELETE is invoked

KILL

INTRINSIC NUMBER 102

Deletes a son process.

SYNTAX

IV
KILL(*pin*);

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

PARAMETERS

<i>pin</i>	<i>integer by value (required)</i> A word containing the Process Identification Number (PIN) of the process to be deleted.
-------------------	---

CONDITION CODES

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

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Deleting Processes" in Section III.

Dynamically loads a library procedure.

SYNTAX

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

LOADPROC dynamically loads a library procedure, and any external procedures it has referenced.

FUNCTIONAL RETURN

identnum *integer (optional)*
An identity number required for use in unloading the procedure. If a loader error occurs, the identity number represents a loader error code.

PARAMETERS

procname *byte array (required)*
Contains the name of the procedure to be loaded. The name must be terminated by a blank.

lib *integer by value (required)*
An integer value of 0, 1, or 2, to request library searching for the procedure residing in the logon group of the user, as follows:

- 0 Search the system library only.
- 1 Search the account library, then the system library.
- 2 Search the group library first, the account library second, and the system library last.

plabel *integer (required)*
The word to which the procedure's label (*plabel*) is returned. This is the external *plabel* so that the SPL construct ASSEMBLE(PCAL 0) may be used.

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied. The value returned to <i>identnum</i> is a loader error code.

ADDITIONAL DISCUSSION

"Dynamic Loading and Unloading of Library Procedures" in Section V.

LOCKGLORIN

INTRINSIC NUMBER 34

Locks a global RIN.

SYNTAX

IV	L	BA
LOCKGLORIN(<i>rinnum</i> , <i>lockcond</i> , <i>rinpassword</i>);		

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

To use the LOCKGLORIN intrinsic, you must know both the RIN number and the RIN password. Multiple RIN (MR) capability is required to lock two or more global RINs simultaneously. An attempt by a user with standard capabilities to lock two or more RINs simultaneously aborts the process.

PARAMETERS

rinnum

integer by value (required)

A word specifying the RIN number of the resource to be locked. This is the RIN number furnished in the :GETRIN command. Refer to the MPE V Commands Reference Manual (32033-90006) for a description of the :GETRIN command.

lockcond

logical (required)

A word specifying conditional or unconditional RIN locking, through bit (15:1). Bit (15:1) has the following settings:

- =0 Locking takes place only if the RIN is immediately available. If the RIN is not immediately available, control returns to the calling process immediately with the condition code CCG.
- =1 Locking will take place unconditionally. If the RIN is not available, the calling process suspends until it becomes available.

All other bits are ignored.

rinpassword

byte array (required)

Contains the RIN password assigned through the :GETRIN command. This array must be a minimum of 10 bytes in length and must be terminated by a nonalphanumeric ASCII character (a blank is recommended).

CONDITION CODES

The condition codes possible if *lockcond* bit (15:1)=1 are:

CCE	Request granted. If the calling process had already locked the RIN, <i>lockword</i> bit (15:1)=1. If the RIN was free, <i>lockword</i> bit (15:1)=0.
CCG	Not returned.
CCL	Request denied because of invalid RIN. <i>Rinnum</i> is not a global RIN or the value is out of bounds for the RIN table.

The condition codes possible if *lockcond* bit (15:1)=0 are:

CCE	Request granted. If the calling process had already locked the RIN, <i>lockword</i> bit (15:1)=0. If the RIN was free, <i>lockword</i> bit (15:1)=1.
CCG	Request denied because the RIN was locked by another job.
CCL	Request denied because of invalid RIN. <i>Rinnum</i> is not a global RIN or the value is out of bounds for the RIN table.

SPECIAL CONSIDERATIONS

Multiple RIN (MR) capability is required if you are doing the following:

- Locking more than one global RIN at a time within a process.
- Locking one RIN within a process tree.
- Locking any files (for example, database).
- Using local RINs with global RINs.

ADDITIONAL DISCUSSION

"Resource Management" in Section III.

LOCKLOCRIN

INTRINSIC NUMBER 32

Locks a local RIN.

SYNTAX

IVL

`LOCKLOCRIN(rinum,lockcond);`

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

PARAMETERS

rinum

integer by value (required)

A number specifying one of the previously allocated local RINs, designated by an integer from 1 to the value specified in the *rincount* parameter of the GETLOCRIN intrinsic.

lockcond

logical (required)

A word specifying conditional or unconditional locking, through bit (15:1). The settings for bit (15:1) are as follows:

=0 Locking takes place only if the RIN is immediately available. If it is not, control returns to the calling process immediately with CCG.

=1 Locking takes place unconditionally. If the RIN is not available, the calling process suspends until the RIN becomes available.

All other bits are ignored.

CONDITION CODES

The condition codes possible if *lockcond* bit (15:1)=1 are:

CCE

Request granted. If the calling process had already locked the RIN, *lockcond* bit (15:1)=1. If the RIN was free, *lockcond* bit (15:1)=0.

CCG

Not returned.

CCL

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

The condition codes possible if *lockcond* bit (15:1)=0 are:

CCE	Request granted. If the calling process had already locked the RIN, <i>lockcond</i> bit (15:1)=1. If the RIN was free, <i>lockcond</i> bit (15:1)=0.
CCG	Request denied because the RIN was locked by another process.
CCL	Request denied because the RIN was invalid; the <i>rinum</i> was too large, no local RIN was allocated, or <i>rinum</i> specified a number less than or equal to zero.

ADDITIONAL DISCUSSION

"Resource Management" in Section III.

LOCRINOWNER

INTRINSIC NUMBER 36

Determines PIN of process that has locked a local RIN.

SYNTAX

IIV

```
pin := LOCRINOWNER(rinnum);
```

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

FUNCTIONAL RETURN

pin

integer (optional)

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

PARAMETERS

rinnum

integer by value (required)

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

CONDITION CODES

CCE

Request granted.

CCG

Request denied. The local RIN specified by *rinnum* is not currently locked by any process.

CCL

Request denied. The *rinnum* parameter was invalid (i.e. *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).

ADDITIONAL DISCUSSION

"Resource Management" in Section III.

Provides information about an opened user's logging file (whole file set). (Available on version G.02.00 or later.)

SYNTAX

```

O-V      DV      I      IV      BA
LOGINFO (index,status [,itemnum1,itemval1]
          [,itemnum2,itemval2]
          [,itemnum3,itemval3]
          [,itemnum4,itemval4]);

```

The LOGINFO intrinsic is used to obtain information about the whole logging file set. Up to four items of information can be retrieved by specifying one or more *itemnum/itemval* pairs. The *itemnum/itemval* parameters must appear in pairs.

PARAMETERS

index *double by value (required)*
The index parameter returned from OPENLOG. Identifies the user access to the logging file.

status *integer (required)*
One of the following integers indicating the success/failure of the intrinsic call:

Message No.	Meaning
0	No error occurred for this call.
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. Must have User Logging (LG) and System Supervisor (OP) capabilities to use a logging intrinsic.

Message No.	Meaning
8	Incorrect password passed to logging intrinsic.
9	Error occurred while writing logging file.
12	System is out of disc space; logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user logging file.
16	Invalid logging identifier.
17	Either <i>itemnum</i> or <i>itemval</i> is missing.
18	Invalid item number.

itemnum *integer by value (optional)*
Cardinal number of the item desired; this specifies which item value is to be returned. (Refer to Item# in Table 2-7.)

itemval *byte array (optional)*
The value of the item specified by the corresponding item number; the data type of the item value depends on the item itself. (Refer to Item Value in Table 2-7.)

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capabilities required.

ADDITIONAL DISCUSSION

"User Logging" in Section III.

Table 2-7. Item Values Returned By LOGINFO

ITEM#	ITEM VALUE	TYPE
1	Total # of records written in current file	D
2	Current file size	D
3	Current file space left	D
4	Number of users	I
5	Total records written in whole set	D
6	Current logfile name	36 bytes
7	Current logfile type: disc, tape, sdisc, ctape	I
8	Previous logfile name	36 bytes
9	Previous logfile type: disc, tape, sdisc, ctape	I
10	CHANGELOG allowed	L
11	AUTO allowed	L
12	Current file sequence number	I
13	Log status	I
	0 = Inactive	
	1 = Active	
	2 = CHANGELOG pending	
	3 = STOP pending	

LOGSTATUS

INTRINSIC NUMBER 214

Provides information about a current opened user logging file.

SYNTAX

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

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

PARAMETERS

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

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

Words 0 and 1 - Total records written in current logging file.

Words 2 and 3 - The size, in records, of the current logging file.

Words 4 and 5 - The space, in records, remaining in the current logging file.

Word 6 - The number of users using the logging system.

status *integer (required)*
One of the following integers indicating the success/failure of the intrinsic call:

Message No.	Meaning
0	No error occurred for this call.
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.

Message No.	Meaning
--------------------	----------------

7	Illegal capability. Must have User Logging (LG) and System Supervisor (OP) capabilities to use a logging intrinsic.
8	Incorrect password passed to logging intrinsic.
9	Error occurred while writing logging file.
12	System is out of disc space, logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user logging file.
16	Invalid logging identifier.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capabilities required.

ADDITIONAL DISCUSSION

"User Logging" in Section III.

MAIL

INTRINSIC NUMBER 106

Tests mailbox status.

SYNTAX

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

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

FUNCTIONAL RETURN

status

logical (optional)

One of the following which indicates the status of the mailbox:

Status Returned	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 <i>count</i> .
3	An error occurred because an invalid <i>pin</i> was specified or a bounds check failed.
4	The mailbox is temporarily inaccessible because other intrinsics are using it in the preparation or analysis of mail.

PARAMETERS

pin

integer by value (required)

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

count

integer (required)

A word to which an integer denoting the length, in words, of any incoming mail in the mailbox is returned.

CONDITION CODES

CCE	Request granted. The mailbox <i>status</i> was tested.
CCG	Request denied because an illegal <i>pin</i> parameter was specified. The value of 3 is returned to the calling process through <i>status</i> .
CCL	Not returned by this intrinsic.

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Interprocess Communication" in Section III.

MYCOMMAND

INTRINSIC NUMBER 71

Parses (delineates and defines parameters for) a user-defined command image.

SYNTAX

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

You can extract and format the parameters of a command, other than an MPE command, for execution by using the MYCOMMAND intrinsic within your program. This intrinsic also allows you to request the searching of a byte array, serving as a command dictionary, for a specified command.

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

FUNCTIONAL RETURN

entryno

integer (optional)

An integer specifying the command entry number. If the *dict* parameter was not specified, this number is 0.

PARAMETERS

comimage

byte array (required)

Contains either:

- A command name (expected if the *dict* parameter is specified), followed by parameters, followed by a carriage-return character (%15). The command name is delimited by the first nonalphanumeric character, and cannot be preceded by any leading blanks. The parameters are formatted and referenced in the *params* array. Also, *comimage* is converted to uppercase and the byte array specified by *dict* is searched for a name matching the command.
- Only command parameters (expected if the *dict* parameter is not specified), followed by a carriage-return character (%15). These parameters are formatted. Leading and trailing blanks are ignored. Lowercase is upshifted. In the byte array named for the *comimage* parameter, the first character of the parameter list may be a leading blank.

delimiters

byte array (optional)

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

Default: If this parameter is omitted, the delimiter array ",=; (carriage return)" is used.

maxparms *integer by value (required)*
 An integer specifying the maximum number of parameters expected in *comimage*.

numparms *integer (required)*
 The number of parameters found in *comimage*.

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

Word 1: Contains the byte pointer to the first character of the parameter. If the parameter is empty or all blanks, the pointer indicates the location of the delimiter.

Word 2: Contains bits that describe the parameter:

Bits (11:5) -
 The zero-relative position of the delimiter in *delimiters*. For example, if the default *delimiters* array is used, and the current parameter is delimited by a semicolon, this field will contain 2.

Bit (10:1) - Special characters indicator bit.
 This bit may be set as follows:

=0 The parameter contains no special characters.

=1 The parameter contains special characters other than those listed in *delimiters*.

Bit (9:1) - Numeric character indicator bit.
 The settings for this bit are as follows:

=0 The parameter does not contain numeric characters.

=1 The parameter contains numeric characters.

Bit (8:1) - Alphabetic characters indicator bit.
 This bit may be set as follows:

=0 The parameter does not contain alphabetic characters.

=1 The parameter contains alphabetic characters.

Bits (0:8) - These bits contain the length of the parameter in bytes.
 This value is zero if the parameter is omitted.

dict

byte array (optional)

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

Default: 0 is returned.

defn

byte pointer (optional)

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

Default: The corresponding information is not returned.

CONDITION CODES

CCE

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

CCG

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

CCL

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

ADDITIONAL DISCUSSION

"Formatting Command Parameters" in Section V.

Provides access to the user logging facility.

SYNTAX

D LA LA I I

```
OPENLOG(index,logid,pass,mode,status);
```

The OPENLOG intrinsic provides access to the user logging facility. Effective with version G.02.00, the number of users and log entries are independent of the number of times OPENLOG is called. There must already be an active process for this logging identifier. If the process is active, it will get an entry in the logging buffer. Previously each call to OPENLOG obtained an entry and incremented the user count in the logging buffer table. This occurred whether or not the user already had an entry in the table. This caused applications to exceed the limit for users per process or log entries prematurely.

Now a logging buffer entry is obtained and the user count is incremented only if this is the first OPENLOG call for this user. A counter is used to keep track of the number of times a user has performed OPENLOG and CLOSELOG. The counter is incremented for every OPENLOG and decremented for every CLOSELOG. This is done to ensure that the entry in LOGBUFF is released only if this is the last CLOSELOG call for this user (i.e. counter = 0).

PARAMETERS

<i>index</i>	<i>double (required)</i> A double-word returned which identifies logging access. The <i>index</i> parameter is used to check the validity of subsequent calls to the other user logging intrinsics.
<i>logid</i>	<i>logical array (required)</i> An array of up to eight characters which supplies user logging identification. The array contains alphanumeric characters. Arrays of less than eight characters must be terminated with a nonalphanumeric character.
<i>pass</i>	<i>logical array (required)</i> An array in which a password associated with the logging identifier by using the :GETLOG command is assigned.
<i>mode</i>	<i>integer (required)</i> An integer used to indicate whether or not a process should be suspended if a request for service cannot be completed immediately. Enter a zero if you want to wait for service; enter a one if you do not want to wait.
<i>status</i>	<i>integer (required)</i> One of the following integers that the logging system uses to return information on the status of the intrinsic call to the user:

Message No.	Meaning
0	No error occurred for this call.
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. 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 is out of disc space, logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user logging file.
16	Invalid logging identifier.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capabilities required.

ADDITIONAL DISCUSSION

"User Logging" in Section III.

Suspends calling process for a specified number of seconds.

SYNTAX

R

PAUSE(*interval*);

PARAMETERS

interval

real (required)

A positive real value specifying the amount of time, in seconds, that the process will pause. The maximum time allowed is approximately 2,147,484 seconds (almost 25 days).

CONDITION CODES

CCE	Request granted.
CCG	Request denied because of insufficient system table (Timer Request List) space.
CCL	Request denied because a negative value was specified for interval or the value is too large.

ADDITIONAL DISCUSSION

"Suspending the Calling Process" in Section V.

PRINT

INTRINSIC NUMBER 65

Prints character string on job/session listing device.

SYNTAX

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

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

PARAMETERS

message

logical array (required)

Contains the character string to be output.

NOTE

SPL programmers can avoid 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)

An integer denoting the length of the character string to be transmitted. If length is positive, it specifies the length in words; if length is negative, it specifies the length in bytes. Note that if length exceeds the configured record length of the device, successive records will be written only on terminals.

control

integer by value (required)

An integer representing a Carriage Control Code as shown in Table 2-5, "Carriage Control Directives", found in the description of the FWRITE intrinsic.

CONDITION CODES

CCE

Request granted.

CCG

End-of-data was encountered.

CCL

Request denied because of input/output error. Further error analysis through the FCHECK intrinsic is not possible.

ADDITIONAL DISCUSSION

"Writing Output to the Job/Session List Device" in Section V.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

PRINTFILEINFO

INTRINSIC NUMBER 21

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

SYNTAX

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

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

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

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

PARAMETERS

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

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Writing a File System Error-Check Procedure" in Section IV.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

Prints a character string on the System Console.

SYNTAX

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

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

PARAMETERS

<i>message</i>	<i>logical array (required)</i> The array from which the character string is output. The character string contained in <i>message</i> is limited to 56 characters. Nonvideo enhancing escape sequences are stripped out.
<i>length</i>	<i>integer by value (required)</i> An integer denoting the <i>length</i> of the output string to be transmitted. If <i>length</i> is positive, it specifies the length in words; if <i>length</i> is negative, it specifies the length in bytes.
<i>control</i>	<i>integer by value (required)</i> The value 0 or %320.

NOTE

If a null character (%000) is embedded in an array to be printed on the System Console via 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 by this intrinsic.
CCL	Request denied because of a physical input/output error. Further error analysis through the FCHECK intrinsic is not possible.

ADDITIONAL DISCUSSION

"Writing Output to the System Console" in Section V.

PRINTOPREPLY

INTRINSIC NUMBER 67

Prints a character string on the System Console and solicits a reply.

SYNTAX

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

The PRINTOPREPLY intrinsic transmits a string of ASCII characters from an array in your program to the System Console and solicits a reply. While the reply is pending, the string can be viewed by using the :RECALL command.

FUNCTIONAL RETURN

lgth

integer (optional)

A positive integer indicating the *length* of the reply from the System Operator. This *length* represents a word count if *expectedl* is positive or a byte count if *expectedl* is negative.

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

If an error occurs, the value returned is zero.

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

PARAMETERS

message

logical array (required)

The array from which the characters are output to the System Console. The character string is limited to 50 characters.

length

integer by value (required)

An integer denoting the *length* of the output string to be transmitted. If *length* is positive, it specifies the *length* in words; if *length* is negative, it specifies the length in bytes. This parameter should never specify a length of more than 50 bytes.

zero

integer by value (required)

This parameter is not used by MPE but it must be specified. Typically it is assigned the value of zero.

reply

logical array (required)

The array into which the input characters are read from the System Console.

expectedl

integer by value (required)

An integer specifying the maximum *length* of the message to be read into the array *reply*. If *expectedl* is positive, it specifies a word count; if it is negative, it specifies a byte count. This parameter should never specify a reply *length* of more than 31 bytes.

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Writing Output to the System Console and Requesting a Reply" in Section V.

PROCINFO

INTRINSIC NUMBER 111

Provides access to process information.

SYNTAX

0-V	I	I	IV	I	BA
PROCINFO(<i>error1</i> , <i>error2</i> , <i>pin</i> [, <i>itemnum1</i> , <i>item1</i>]					
[, <i>itemnum2</i> , <i>item2</i>]					
[, <i>itemnum3</i> , <i>item3</i>]					
[, <i>itemnum4</i> , <i>item4</i>]					
[, <i>itemnum5</i> , <i>item5</i>]					
[, <i>itemnum6</i> , <i>item6</i>]);					

PARAMETERS

- error1*** *integer (required)*
An integer indicating the success or failure of the intrinsic call as described in Figure 2-4.
- error2*** *integer (required)*
An integer which supplies additional information concerning an error reported in *error1* as described in Figure 2-4.
- pin*** *integer by value (required)*
An integer specifying the Process Identification Number (PIN) for which information is to be returned. A *pin* value of zero will return information about the calling process. This parameter is not compatible with the *pin* parameter of the GETPROCINFO intrinsic.
- itemnum*** *integer (optional)*
An integer containing the item number (in any order) of an information option as defined in Figure 2-5. The user may request up to 6 options to be returned.
- item*** *byte array (optional)*
Arrays (in the same order as the *itemnums*) of returned information as specified in Figure 2-5.

The parameters *error1*, *error2*, and *pin* are required. The *itemnum* and *item* parameters are optional. The actual number included depends upon the information desired. The *itemnums* and the *items* are paired such that the *nth itemnum* corresponds to the *nth item*. An *itemnum* contains the option number of the desired information. The information is returned in the corresponding *item* or is stored using the *item* element as a pointer, depending on the information desired. See Note 3 in Figure 2-5 for the user capabilities required for the use of any option.

CONDITION CODES

CCE Successful call. All error codes set to zero.

CCG Not used.

CCL Unsuccessful call with error codes set accordingly.

ADDITIONAL DISCUSSION

None.

Error1	Meaning	Error2
0	Successful execution - no error.	0
1	Insufficient capability to return request information.	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>item#</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> pair.
Note 1:	The process will abort if <i>error1</i> parameter address is illegal or if the intrinsic is called in split-stack mode.	
Note 2:	If an error condition is detected while processing an information request, the index of the <i>itemnum</i> where the offending option was located is stored in <i>error2</i> .	

Figure 2-4. Error Codes Returned From PROCINFO

ITEM#	INFORMATION RETURNED	ITEM
0	Ignored.	Ignored.
1	Process Identification Number of calling process.	Integer where PIN will be returned.
2	Process Identification Number of the father of the specified process.	Integer where PIN will be returned. (See Note 3.)
3	Number of sons of the specified process (direct descendants).	Integer where the number of sons will be returned. (See Note 3.)
4	Number of descendants (both direct and indirect) of the specified process.	Integer where the number of descendants will be returned. (See Note 3.)
5	Number of generations (number of levels in the process tree substructure) the specified process has including itself.	Integer where number of generations will be returned. (See Note 3.)
6	Process Identification Numbers of all sons (direct descendants).	Integer array where son PINs will be returned. (See Note 1, 3.)
7	Process Identification Numbers of all descendants (both direct and indirect).	Integer array where descendent PINs will be returned. (See Note 1, 3.)
8	Priority number in the Master Queue of specified process.	Integer where priority will be returned (same as Word 1 of GETPROCINFO intrinsic).
9	State and activation information of the specified process.	Logical where information will be returned (same as Word 2 of GETPROCINFO intrinsic).
10	Program name where the specified process is currently executing.	Byte array where the fully qualified program name will be stored. (See Note 2, 3.)
<p>Note 1: Options 6/7 return a variable number of PINs. In these cases <i>item</i> should be set by the calling process to point to an integer where the PINs will be returned. The first word of the array should be set by the calling process to indicate the array size in words and the array size should include the array size word (i.e. if the user desires four pins, the first entry that contains the array size should be 5). PINs will be stored into the array, one PIN per word, starting with the second word and continuing until the array is filled or all PINs have been returned. If the array is not filled, the remaining unused locations will be padded with zeros.</p> <p>Note 2: The byte array for the program name must be a minimum of 28 bytes long. The name will be returned in the form of "f.g.a" where "f" will be the local file name, "g" will be the group name, and "a" will be the account name of the file containing the program that the specified process is currently executing. The name will be returned left-justified with the unused locations filled with blanks.</p> <p>Note 3: If the calling process is executing in Privileged Mode, requests for information will be honored for any process. Otherwise, requests will be honored as follows:</p> <ol style="list-style-type: none"> 1. Complete information will be returned for sons of the calling process itself. 2. Item 10 will be returned only if the calling process has read access to the program file. 3. Information returned for indirect descendants and processes directly above the calling process will be limited to items 2 through 7 and 10 only. <p>Process Handling capability will also be required for any user mode call unless the calling process is requesting information about itself.</p>		

Figure 2-5. Information Options For PROCINFO

Returns the accumulated CPU time for a process.

SYNTAX

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

The PROCTIME intrinsic is used to obtain the amount of CPU time, in milliseconds, that a process has accumulated. This is the basis on which CPU time is charged. (Refer to the :REPORT command in the MPE V Commands Reference Manual (32033-90006) for additional information.)

FUNCTIONAL RETURN

<i>time</i>	<i>double (optional)</i>
	A double integer value which shows the number of milliseconds that the process has been running.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Obtaining Process Run Time" in Section V.

PTAPE

INTRINSIC NUMBER 191

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

SYNTAX

```
PTAPE(IVfileum1,IVfileum2);
```

When using terminals with attached tape readers (such as the ASR-33), you can read data program-matically 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 Y^C 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 disc file (*fileum2*) to the proper position in the file. If you are reading more than one tape, you should specify, in the FOPEN intrinsic call that opens the disc file, the append-only *aooption* and a variable-length record format, before the first PTAPE call. In addition, set the end-of-file pointer to zero, if necessary, before issuing the first PTAPE intrinsic call.

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

PARAMETERS

<i>fileum1</i>	<i>integer by value (required)</i> A word supplying the file number of the user's terminal. This is the value returned by FOPEN when the terminal file was opened.
<i>fileum2</i>	<i>integer by value (required)</i> A word supplying the file number of the disc file to which the data is to be written.

CONDITION CODES

CCE	Request granted.
CCG	Request denied because an error occurred while writing to the specified disc file.
CCL	Request denied because the input file specified is not a terminal or does not belong to the calling process, or because insufficient resources, such as disc space or main memory, are available to satisfy the request. (PTAPE requires 128 contiguous sectors of work area on <i>ldev1</i> .)

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

Appendix B, "DEVICE CHARACTERISTICS".

Point-to-Point Workstation I/O Reference Manual (30000-90250).

PUTJCW

INTRINSIC NUMBER 85

Assigns the value of a particular Job Control Word (JCW) in the JCW Table.

SYNTAX

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

The PUTJCW intrinsic assigns a specified value to a Job Control Word (JCW) in the JCW Table. If an entry of the same name already exists in the table, only its value is entered. If no entry exists for this name, an entry is created and its value is entered.

PARAMETERS

jcwname

byte array (required)

A byte array containing the name of the JCW. This array may 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 (required)

A word containing the value of the JCW.

status

integer (required)

A word used by the system to return a value denoting the execution status of the intrinsic, as follows:

- 0 Successful execution. Value entered in JCW.
- 1 Error, *jcwname* greater than 255 characters long.
- 2 Error, *jcwname* does not start with a letter.
- 3 Error, JCW table overflow. No JCW with this name exists in table and unable to create new entry.
- 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.

Value 5 will only be returned if the fundamental operating software is version G.00.00 or later.

SPECIAL CONSIDERATIONS

There are three types of JCWs in the system; user-defined JCWs, system-defined JCWs, and system-reserved JCWs. Attempts to assign a value to a system-reserved JCW will result in an error. The system-reserved JCWs are HPDATE, HPDAY, HPHOUR, HPMONTH, HPMINUTE, and HPYEAR.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"User-Defined Job Control Words" in Section V.

QUIT

INTRINSIC NUMBER 76

Aborts a process.

SYNTAX

IV
QUIT(*num*);

From within any process in a user program structure, you can abort the process by using the QUIT intrinsic. The QUIT intrinsic also transmits an abort message to the calling process output device, and sets the job/session in an error state. In batch jobs not containing the :CONTINUE command this results in job termination. For additional information refer to the MPE V Commands Reference Manual (32033-90006).

PARAMETERS

num

integer by value (required)

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

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

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

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Affects the system Job Control Word.

ADDITIONAL DISCUSSION

"Aborting a Process" in Section V.

Aborts the entire user process structure.

SYNTAX

IV
QUITPROG(*num*);

This intrinsic destroys all sons of the job/session main process. The job/session main process is set in the error state. In batch jobs not containing the :CONTINUE command this terminates the job. Refer to the MPE V Commands Reference Manual (32033-90006) for additional information.

An abort message is transmitted to the job/session list device.

PARAMETERS

num *integer by value (required)*
Any arbitrary number. When the QUITPROG intrinsic is executed, *num* is output as part of the abort message, as follows:

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

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

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Affects the system Job Control Word.

ADDITIONAL DISCUSSION

"Aborting a Program" in Section V.

READ

INTRINSIC NUMBER 64

Reads an ASCII string from \$STDIN into an array.

SYNTAX

I	LA	IV
<i>lgth</i> := READ(<i>message</i> , <i>expectedl</i>);		

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

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

READ differs from READX in how it interprets an end-of-file. READ reads a record and if there is a colon in the first column end-of-file is set.

FUNCTIONAL RETURN

lgth

integer (optional)

A positive integer value representing the length of the ASCII string which was read. If *expectedl* is positive, the length specified is words; if *expectedl* is negative, the length specified is bytes.

PARAMETERS

message

logical array (required)

The array into which the ASCII characters are read.

expectedl

integer by value (required)

An integer specifying the maximum length of the *message* array. If *expectedl* is positive, it specifies the length in words; if *expectedl* is negative, it specifies the length in bytes. When the record is read the first *expectedl* characters are input. If *expectedl* equals or exceeds the size of the physical record, the entire record is transmitted.

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Transmitting Program Input/Output from Job/Session Input/Output Devices" in Section V.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

READX

NO INTRINSIC NUMBER ASSIGNED

Reads an ASCII string from \$STDINX into an array.

SYNTAX

I	LA	IV
<i>lgth</i> := READX(<i>message</i> , <i>expectedl</i>);		

This intrinsic is similar to issuing an FREAD intrinsic call against the file \$STDINX. The READ intrinsic is limited in its usefulness, however, in that the full capability of the file system is not available to a user of this intrinsic. For example, :FILE commands are not allowed and certain file intrinsics cannot be used because the *filenum* parameter, obtained from the FOPEN intrinsic, is not available to normal users of the READ intrinsic.

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

READX differs from READ in how it interprets an end-of-file. READX interprets :EOD, :EOF:, or in a job, :EOJ, :JOB, or :DATA as an end-of-file indication.

FUNCTIONAL RETURN

<i>lgth</i>	<i>integer (optional)</i> A positive integer value representing the length of the ASCII string which was read. If <i>expectedl</i> is positive, the length is specified in words; if <i>expectedl</i> is negative, the length is specified in bytes.
-------------	---

PARAMETERS

<i>message</i>	<i>logical array (required)</i> The array into which the ASCII characters are read.
----------------	--

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

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Transmitting Program Input/Output from Job/Session Input/Output Devices" in Section V.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

RECEIVEMAIL

INTRINSIC NUMBER 108

Receives mail from another process.

SYNTAX

```

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

```

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

FUNCTIONAL RETURN

status

logical (optional)

One of the following mailbox status codes:

Status Returned	Meaning
0	The mailbox was empty and <i>waitflag</i> 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 an invalid <i>pin</i> was specified or a bounds check failed.
4	The request was denied because <i>waitflag</i> specified that the receiving process wait for mail if the mailbox is empty, but the other process sharing the mailbox is already suspended, waiting for mail. If both processes were suspended, neither could activate the other, and they may be deadlocked.

PARAMETERS

pin

integer by value (required)

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

location

logical array (required)

The array (buffer) in the stack where the *message* is to be written.

waitflag

logical by value (required)

A word which specifies the action to be taken if the mailbox is empty. This is determined by the setting of bit (15:1). Bit (15:1) has the following settings:

=0 Return immediately to the calling process.

=1 Wait until incoming mail is ready for collection.

CONDITION CODES

CCE Request granted. The mail was collected (value 2 is returned to *status*) or the mail was not collected because the mailbox contained outgoing mail from the receiving process (value 1 is returned to *status*).

CCG Request denied because of an illegal *pin* parameter (value 3 is returned to *status*).

CCL Request denied because the bounds check revealed that the *location* parameter did not define a legal stack address (value 3 is returned to *status*) or because both sending and receiving processes would be awaiting incoming mail causing a deadlock (value 4 is returned to *status*).

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Interprocess Communication" in Section III.

DSN/DS Reference Manual (32190-90001).

RESETCONTROL

INTRINSIC NUMBER 55

Resets terminal to accept CONTROL-Y signal.

SYNTAX

```
RESETCONTROL;
```

The RESETCONTROL intrinsic is used to reset a terminal so it can accept a CONTROL-Y signal. To take effect, this intrinsic may be called at any time from any procedure, after the CONTROL-Y trap has been invoked.

CONDITION CODES

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

ADDITIONAL DISCUSSION

"CONTROL-Y Traps" in Section V.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

Disables the abort stack analysis facility.

SYNTAX

```
RESETDUMP;
```

CONDITION CODES

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

ADDITIONAL DISCUSSION

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

MPE V Commands Manual (32033-90006).

SEARCH

INTRINSIC NUMBER 70

Searches an array for a specified entry or name.

SYNTAX

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

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

FUNCTIONAL RETURN

entryno *integer (optional)*
The entry number of the word in *dict* which matches *target*. If the name specified in *target* is not found, a zero is returned.

PARAMETERS

target *byte array (required)*
Contains the name for which the search is to be performed.

length *integer by value (required)*
An integer specifying the length, in bytes, of the byte array *target*.

dict *byte array (required)*
The specially formatted array which is to be searched for *target*.

defn *byte pointer (optional)*
A word to which the address of the definition portion of the entry sought in the array is returned.

Default: If *defn* is omitted, the address is not returned.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Searching Arrays" in Section V.

Sends mail to another process.

SYNTAX

```

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

```

A process sends mail to its father or sons by using the SENDMAIL intrinsic. If the receiving process mailbox contains an uncollected message from the calling process, the action taken depends on the *waitflag* parameter specified in the SENDMAIL intrinsic call. If the mailbox currently is being used by other intrinsics, the SENDMAIL intrinsic waits until the mailbox is free and then sends the mail.

FUNCTIONAL RETURN

status

logical (optional)

One of the following status codes indicating the success of the intrinsic:

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

PARAMETERS

<i>pin</i>	<i>integer by value (required)</i> An integer specifying the process to receive the mail. If a son process is specified, the integer is the Process Identification Number (PIN) of that process. If a father process is specified, the integer is zero.
<i>count</i>	<i>integer by value (required)</i> An integer specifying the length of the message, in words, transmitted from the stack of the sending process. If zero is specified, SENDMAIL empties the mailbox of any incoming or outgoing mail.
<i>location</i>	<i>logical array (required)</i> The array (buffer) in the stack containing the message.
<i>waitflag</i>	<i>logical by value (required)</i> A word specifying in bit (15:1) the action to be taken if the mailbox contains previously sent mail. Bit (15:1) has the following settings: =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.

CONDITION CODES

CCE	Request granted. The mail was sent (value 0 or 1 is returned to <i>status</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>status</i>).
CCG	Request denied because of an illegal <i>count</i> parameter (value 5 is returned to <i>status</i>), an illegal <i>pin</i> parameter was specified (value 3 is returned to <i>status</i>), or storage for the mail data segment was not available (value 6 is returned to <i>status</i>).
CCL	Request denied because the bounds check revealed that the <i>count</i> or <i>location</i> parameters did not define a legal stack area (value 3 is returned to <i>status</i>), or both processes are waiting to send mail (value 4 is returned to <i>status</i>).

SPECIAL CONSIDERATIONS

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Interprocess Communication" in Section III.

Enables the stack analysis facility.

SYNTAX

LV
SETDUMP(*flags*);

PARAMETERS

flags

logical by value (required)

A logical word whose bits specify the following if set to =1:

Bit (15:1) - Specifies a DL to Q initial dump.

Bit (14:1) - Specifies a Q initial to S dump.

Bit (13:1) - Specifies a Q-63 to S dump. This bit is ignored if bit (14:1)=1.

Bit (12:1) - Causes an ASCII dump of the octal content along with the octal values.

If bits (12:4)=0, *flags* will specify a display of registers and stack marker trace only.

Bits (0:12) are reserved for MPE.

CONDITION CODES

CCE Request granted.

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

CCL Not returned by this intrinsic.

ADDITIONAL DISCUSSION

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

SETJCW

INTRINSIC NUMBER 72

Sets bits in the system Job Control Word, JCW.

SYNTAX

LV

SETJCW(*word*);

You can establish the bit contents of the system Job Control Word, JCW, with the SETJCWintrinsic.

PARAMETERS

word

logical by value (required)

A 16-bit word whose contents are established by the user for Interprocess Communication. The form is:



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

PROGRAM TERMINATED IN ERROR STATE (CIERR 976)

In batch mode, the job is terminated unless the :CONTINUE command is used. If you set JCW 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)

Refer to the MPE V Commands Reference Manual (32033-90006) for a discussion of :CONTINUE. Bits (1:15) may be used for any purpose.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"User-Defined Job Control Words" in Section V.

MPE V Commands Reference Manual (32033-90006).

Dumps selected parts of stack to a file.

SYNTAX

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

or (from SPL only):

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

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.

Default: Dump is sent to \$STDLIST.

idnumber

integer (optional)

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 (optional)

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. Bounds violation occurred and the dump was not completed. Record size was not between 32 and 256 words.

CCL Request denied. File system error occurred during opening, writing to, or closing the file. The file error number is returned in *idnumber*.

ADDITIONAL DISCUSSION

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

Initiates a session on the specified terminal. (Available only on version G.01.00 and later.)

SYNTAX

IV BA I D IA

STARTSESS(*ldev*,*logonstr*,*jsid*,*jsnum*,*errorstat*);

STARTSESS initiates a session on the specified terminal using the given logon string. The terminal on which the session is to be created must be available; no other user may be logged on. The target terminal will not be speed sensed, so it must be set at the configured baud rate.

When the session is created, nothing will be printed to the terminal until the **(RETURN)** key is pressed unless ;NOWAIT is specified. To override the need for a **(RETURN)**, the ;NOWAIT keyword parameter may be entered with the logon string when executing the :STARTSESS command or invoking the STARTSESS intrinsic. For example, `:STARTSESS 21;MANAGER/pass.SYS;NOWAIT` or passing a byte array of "MANAGER/pass.SYS; NOWAIT (%15)" to the STARTSESS intrinsic will start a session and logon immediately without waiting for a **(RETURN)**. The ;NOWAIT parameter should never be used 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 the invoker must have System Manager (SM) capability. All other valid terminals may be specified by the invoker with only Programmatic Sessions (PS) capability.

PARAMETERS

ldev

integer by value (required)

The logical device number of the terminal on which the session is to be created. This must be a real physical device, and it must be a hardwired terminal. The terminal must be configured as TYPE 16 and SUBTYPE 0 or 4.

logonstr

byte array (required)

The character holding the logon parameters in the same format expected in the :STARTSESS command. For more details on the :STARTSESS command, refer to the MPE V Commands Reference Manual (32033-90006). The first characters in the string should be the session name, if specified, or the user name if no session name is assigned; *logonstr* must be terminated with a carriage return character (%15).

Passwords must be supplied in the *logonstr*. There will be no prompting for passwords. If passwords are required but not supplied, STARTSESS will fail.

jsid

integer (required)

Indicates the type of Command Interpreter (CI) process. If the STARTSESS call was successful, the value returned will be 1, if unsuccessful, the value will be 0. When the value returned to *jsid* is 1, *jsid* and *jsnum* can be used as input to JOBINFO to check on the various attributes of the session.

jsnum

double (required)

A 32-bit value which when used with *jsid* uniquely identifies the created session.

errorstat

integer array (required)

A two-element array in which the status of the call is returned. The second element is reserved for future use, and will always contain a zero. The first element will contain a zero if no errors occurred. If an error occurs one of the following error values is returned in the first element:

Error No.	Meaning
0	Successful call.
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	Logical device specified by <i>ldev</i> is not a real device.
7009	Caller lacks Programmatic Sessions (PS) capability.
7014	Session was aborted before logging on.
7015	Logon failed because system is at its session limit.
7016	Logon failed because the session's INPRI was less than or equal to the system's JOBFENCE.
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.
7019	Logon failed because the system was unable to obtain a DST entry for the session's JIT.
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 in the JPCNT table for the session.
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 would only happen 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.
7028	Logon failed because a disconnect was detected on the terminal during logon processing.
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.

Error No.	Meaning
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.
7042	Group is not specified, and there is no default home group.
1444	Password is required but not specified in <i>logonstr</i> .
1424	Missing or invalid user name in <i>logonstr</i> .
1426	Missing or invalid account name in <i>logonstr</i> .
1438	No such user in the account.
1437	No such account in the system.
1436	No such group in the account.
1431	Specified session user lacks Interactive Access (IA) capability.
1412	Logon failed due to JMAT overflow.
1411	Logon failed due to IDD overflow.
-1458	Bad TERMTYPE specified in <i>logonstr</i> , default used.
-1459	Invalid PRI specified in <i>logonstr</i> , default used.
-1479	Invalid TIME specified in <i>logonstr</i> , no time limit is used.
-1461	HIPRI then INPRI specified, INPRI used.
-1464	INPRI then HIPRI specified, HIPRI used.
-1462	INPRI too low, lowest valid INPRI used.
-1463	INPRI too high, highest valid INPRI used.
-1465	OUTCLASS specified, ignored.
-1473	RESTART specified, ignored.
-1452	Unknown parameter found in <i>logonstr</i> , ignored.
-1451	Ignored delimiter.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Programmatic Sessions (PS) capability required.

ADDITIONAL DISCUSSION

None.

SUSPEND

INTRINSIC NUMBER 103

Suspends a process.

SYNTAX

O-V LV IV

SUSPEND(*susp*,*rin*);

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

PARAMETERS

susp

logical by value (required)

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

Bit (15:1) - Father activation bit.

=0 Process does not expect to be activated by its father.

=1 Process expects to be activated by its father.

Bit (14:1) - Son activation bit.

=0 Process does not expect to be activated by one of its sons.

=1 Process expects to be activated by one of its sons.

If (14:1)=1 and (15:1)=1, the suspended process can be activated by either father or sons. Bits (0:14) are reserved by MPE and should be set to zero.

rin

integer by value (optional)

An integer specifying a Resource Identification Number (RIN). If *rin* is specified, it represents a local RIN that is locked by the process but that will be released when this process is suspended. This facility can be used to synchronize processes within the same job.

Default: If omitted, no RIN is unlocked when the process suspends.

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because the <i>susp</i> parameter is not valid, the specified RIN is not owned by this process, or the specified RIN was not locked.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

Process Handling (PH) capability required.

ADDITIONAL DISCUSSION

"Suspending Processes" in Section III.

SWITCHDB

INTRINSIC NUMBER 139

Switches DB register pointer.

SYNTAX

L0-PLV

```
logindex := SWITCHDB(index);
```

The SWITCHDB intrinsic changes the DB register so that it points to the base of an extra data segment instead of the base of the stack.

FUNCTIONAL RETURN

<i>logindex</i>	<i>logical (optional)</i> 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

<i>index</i>	<i>logical by value (required)</i> Specifies 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 <i>index</i> to ensure that the process has acquired access to this segment previously. For an extra data segment, <i>index</i> must be a positive, nonzero integer. To switch back to the stack segment, <i>index</i> must be zero.
--------------	--

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because an illegal data segment was specified.

SPECIAL CONSIDERATIONS

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.

ADDITIONAL DISCUSSION

"Moving the DB Pointer" in Section III.

The GETDSEG intrinsic in this section.

Terminates a process.

SYNTAX

TERMINATE;

A process and all of its descendants (sons), including any extra data segments belonging to them, can be deleted by using the TERMINATE intrinsic.

All files still open by the process and its descendants are closed and assigned the same disposition they had when opened.

CONDITION CODES

The process that calls this intrinsic is terminated and no return is made.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Deleting Processes" in Section III and "Terminating a Process" in Section V.

TIMER

INTRINSIC NUMBER 40

Returns system timer information.

SYNTAX

```
D  
count:=TIMER;
```

A 31-bit logical quantity representing the current system timer and overflow count can be returned to your program with the TIMER intrinsic.

The resolution of the system timer is one millisecond. Thus, readings taken within a one-millisecond period may be identical.

FUNCTIONAL RETURN

<i>count</i>	<i>double (optional)</i>
	A 31-bit logical quantity representing the actual millisecond count since the midnight preceding the last system coldload.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

Split-stack calls permitted.

ADDITIONAL DISCUSSION

"Obtaining System Timer Information" in Section V.

Dynamically unloads a library procedure.

SYNTAX

IV
UNLOADPROC(*procid*);

The UNLOADPROC intrinsic dynamically unloads a procedure and its referenced external procedures.

PARAMETERS

<i>procid</i>	<i>integer by value (required)</i> A word containing the procedure's identity number, which was obtained from the LOADPROC intrinsic call.
----------------------	---

CONDITION CODES

CCE	Request granted.
CCG	Not returned by this intrinsic.
CCL	Request denied because of invalid <i>procid</i> .

ADDITIONAL DISCUSSION

"Dynamic Loading and Unloading of Library Procedures" in Section V.

UNLOCKGLORIN

INTRINSIC NUMBER 35

Unlocks a global RIN.

SYNTAX

IV
UNLOCKGLORIN(*rinnum*);

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

PARAMETERS

<i>rinnum</i>	<i>integer by value (required)</i> A word supplying the number of any RIN locked by the calling process. If <i>rinnum</i> does not specify a RIN locked by the calling process, no action is taken.
---------------	--

CONDITION CODES

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

ADDITIONAL DISCUSSION

"Resource Management" in Section III.

Unlocks a local RIN.

SYNTAX

IV
UNLOCKLOCIN(*rinnum*);

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

PARAMETERS

<i>rinnum</i>	<i>integer by value (required)</i> A word supplying the locked RIN, designated by an integer from 1 to the value specified in the <i>rincount</i> parameter of the GETLOCIN intrinsic call.
---------------	--

CONDITION CODES

CCE	Request granted.
CCG	Request denied because the RIN specified is not locked by the calling process.
CCL	Request denied because the specified RIN is not allocated to this process.

ADDITIONAL DISCUSSION

"Resource Management" in Section III.

WHO

INTRINSIC NUMBER 69

Returns information about a user.

SYNTAX

```
0-V   L       D       D   BA   BA   BA   BA   L
WHO(mode,capability,lattr,usern,groupn,acct,homen,term);
```

The WHO intrinsic supplies the access mode and attributes of the user running the program.

PARAMETERS

mode

logical (optional)

A word to which the current user's access mode is returned. In this word, the bits have the following meanings:

Bit (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 dialog can be established between a program, displaying information on the list device, and a person responding through the input device.

Bit (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.

Bits (12:2)

=00 Is not used.

=01 The user is accessing the system through a session.

=10 The user is accessing the system through a job.

=11 Is not used.

Bits (0:12) - Reserved for MPE. The WHO intrinsic sets these bits to zero.

Default: The user's access mode is not returned.

capability

double (optional)

A double-word to which the user's file access, user, and capability class attributes are returned. In the first word, possession of the following file access and user attributes is indicated by the corresponding bit being =1.

File access attributes:

Bit (15:1)-Ability to save files (declare them permanent) (SF).

Bit (14:1)-Ability to acquire nonsharable devices (ND).

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

Bit (12:1)-Node Manager (NM). (On version G.01.00 or later.)

Bit (11:1)-Network Administrator (NA). (On version G.01.00 or later.)

Bits (9:2)-Reserved for MPE. The WHO intrinsic sets these bits to zero.

Bit (8:1)- User Logging (LG).

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

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

User Attributes:

Bit (5:1)- System Supervisor (OP).

Bit (4:1)- Diagnostician (DI).

Bit (3:1)- Group Librarian (GL).

Bit (2:1)- Account Librarian (AL).

Bit (1:1)- Account Manager (AM).

Bit (0:1)- System Manager (SM).

In the second word, possession of the user's capability-class attributes is indicated by the corresponding bit being =1.

Bit (15:1)-Process Handling (PH).

Bit (14:1)-Extra Data Segments (DS).

Bit (13:1)-Reserved for MPE. The WHO intrinsic sets this bit to zero.

Bit (12:1)-Multiple RINs (MR).

Bit (11:1)-Reserved for MPE. The WHO intrinsic sets this bit to zero.

Bit (10:1)-Programmatic Sessions (PS). (On version G.01.00 or later.)

Bit (9:1)- Privileged Mode operation (PM).

Bit (8:1)- Interactive (session) access (IA).

Bit (7:1)- Batch (job) access (BA).

Bit (0:7)- Reserved for MPE. The WHO intrinsic sets these bits to zero.

Default: The user's file access, user, and capability-class attributes are not returned.

lattn

double (optional)

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

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

usern

byte array (optional)

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

Default: The user's name is not returned.

groupn

byte array (optional)

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

Default: The user's logon group is not returned.

acctn

byte array (optional)

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

Default: The user's logon account is not returned.

homen

byte array (optional)

An 8-byte array to which the name of the user's home group is returned. If a home group is not assigned, this array is filled with blanks.

Default: This information is not returned.

term

logical (optional)

A word to which the logical device number of the job/session input device is returned. If this is a spooled (:STREAM) batch job, then the logical device number will be the virtual device.

A virtual device simulates a spooling device. The actual spooling device cannot be owned by users, so the virtual device allows users access to spooling. Each virtual device will be 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. The logical device number of the virtual device can also be obtained with FFILINFO.

Default: The logical device number is not returned.

CONDITION CODES

The condition code remains unchanged.

ADDITIONAL DISCUSSION

"Determining the User's Access Mode and Attributes" in Section V.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

Writes a record to a logging file.

SYNTAX

D LA I I I

WRITELOG(*index*,*data*,*len*,*mode*,*status*);

The WRITELOG intrinsic journalizes data base and subsystem file additions and modifications to the user logging file.

PARAMETERS

- index*** *double (required)*
The parameter returned from OPENLOG that identifies your access to the logging file.
- data*** *logical array (required)*
The array in which the information to be logged is passed. A log record contains 128 words of which 119 are available to you. Thus the most efficient use of log file space is to structure your arrays with lengths in multiples of 119 words.
- len*** *integer (required)*
The length of the data in *data*. A positive number indicates words, a negative number indicates bytes. If the length is greater than 119 words, the information in *data* is divided into two or more physical log records.
- mode*** *integer (required)*
An integer which you use to indicate whether or not your process should be suspended if your request for service cannot be completed immediately. Enter a zero if you want to wait for service, enter a one if you do not want to wait. If a two is entered it will cause the logging buffer to get written to the disc logfile (disc logging) or disc buffer file (serial logging) at the first opportunity. This is the only intrinsic that allows a setting for *mode* that is greater than 1.
- status*** *integer (required)*
One of the following integers indicating the success/failure of the call:

Message No.	Meaning
0	No error occurred for this call.
1	User requested NOWAIT mode and the logging process is busy.
2	Parameter out of bounds in logging intrinsic.

Message No.	Meaning
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 logging file.
10	Invalid DST passed to a logging system intrinsic.
12	System is out of disc space, logging cannot proceed.
13	No more logging entries.
14	Invalid access to logging file.
15	End-of-file on user log file.
16	Invalid logging identifier.

CONDITION CODES

The condition code remains unchanged.

SPECIAL CONSIDERATIONS

User Logging (LG) and System Supervisor (OP) capabilities required.

ADDITIONAL DISCUSSION

"User Logging" in Section IV.

Enables or disables the user-written software arithmetic trap.

SYNTAX

```

      IV      IV      I      I
XARITRAP(mask,plabel,oldmask,oldplabel);

```

The XARITRAP intrinsic enables you to replace the trap handler in MPE with your own trap handler routine.

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)

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, or PSL.
- Privileged in PROG, GSL, or PSL, *plabel* may be privileged or nonprivileged in PROG, GSL, or PSL.
- Privileged or nonprivileged in SSL, *plabel* may be in any non-MPSSL-segment.

PARAMETERS

mask

integer by value (required)

A word *mask* that selects which hardware traps will invoke the software trap, and which will not. Only the 14 right-most bits of the word forming the *mask* are used. The setting of the other bits is not significant, but it is recommended that they be set to zero. Thus, octal values up to %37777 are allowed for this parameter.

If a bit is on (=1), the corresponding hardware trap activates the software trap. If a bit is off (=0), the corresponding hardware trap does not activate the software trap. If all bits are set to zero, the software trap is disabled. Hardware traps are activated by the following relationships:

Bit	Hardware Error Trap
(15:1)	Floating Point Divide By Zero.
(14:1)	Integer Divide By Zero.
(13:1)	Floating Point Underflow.

(12:1)	Floating Point Overflow.
(11:1)	Integer Overflow.
(10:1)	Extended Precision Overflow.
(9:1)	Extended Precision Underflow.
(8:1)	Extended Precision Divide By Zero.
(7:1)	Decimal Overflow.
(6:1)	Invalid ASCII Digit.
(5:1)	Invalid Decimal Digit.
(4:1)	Invalid Source Word Count.
(3:1)	Invalid Decimal Operand Length.
(2:1)	Decimal Divide By Zero.

plabel

integer by value (required)

The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disabled. The external-type label of the procedure, which resides in the segment transfer table of the procedure's code segment, is passed as a parameter (in SPL) by placing an @ before the procedure name.

oldmask

integer (required)

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

oldplabel

integer (required)

A word in which the previous *plabel* is returned to the user's program. If no *plabel* existed previously, zero is returned.

CONDITION CODES

CCE	Request granted. Software trap enabled.
CCG	Request granted. Software trap disabled.
CCL	Request denied because of an invalid <i>plabel</i> .

ADDITIONAL DISCUSSION

"Enabling and Disabling Traps" in Section V.

XCONTRAP

INTRINSIC NUMBER 54

Enables or disables the CONTROL-Y trap.

SYNTAX

```
          IV      I
XCONTRAP(plabel,oldplabel);
```

When a session is initiated, the CONTROL-Y trap is disabled. The XCONTRAP intrinsic enables this trap. This intrinsic takes effect on the file \$STDINX, and also on \$STDIN (when \$STDIN is defined as a terminal).

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)

If, when a trap procedure is being enabled, the code of the caller is one of the following states:

- 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 SSL, *plabel* may be in any non-MPSSL segment.

PARAMETERS

<i>plabel</i>	<i>integer by value (required)</i> The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disabled.
----------------------	---

<i>oldplabel</i>	<i>integer (required)</i> A word in which the previous <i>plabel</i> is returned to the user's program. If no <i>plabel</i> existed previously, zero is returned.
-------------------------	--

CONDITION CODES

CCE	Request granted. Trap enabled.
CCG	Request granted. Trap disabled.
CCL	Request denied because of illegal <i>plabel</i> , or because \$STDIN is not defined as a terminal.

ADDITIONAL DISCUSSION

"Enabling and Disabling Traps" in Section V.

Point-to-Point Workstation I/O Reference Manual (30000-90250).

XLIBTRAP

INTRINSIC NUMBER 52

Enables or disables the software library trap.

SYNTAX

```
           IV      I
XLIBTRAP(plabel,oldplabel);
```

When a program begins execution, the software library trap is disabled automatically. You can enable (or subsequently disable) this trap with the XLIBTRAP intrinsic call.

PARAMETERS

<i>plabel</i>	<i>integer by value (required)</i> The external-type label of the user's trap procedure. If the value of this entry is 0, the trap is disabled.
<i>oldplabel</i>	<i>integer (required)</i> A word in which the previous <i>plabel</i> is returned to the user's program. If no <i>plabel</i> existed previously, zero is returned.

CONDITION CODES

CCE	Request granted. Trap enabled.
CCG	Request granted. Trap disabled.
CCL	Request denied because of an illegal <i>plabel</i> .

ADDITIONAL DISCUSSION

"Enabling and Disabling Traps" in Section V.

Enables or disables the system trap.

SYNTAX

IVI

XSYSTRAP(*plabel*,*oldplabel*);

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

You can enable (or subsequently disable) the system trap by using the XSYSTRAP intrinsic.

PARAMETERS

<i>plabel</i>	<i>integer by value (required)</i> The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disabled.
<i>oldplabel</i>	<i>integer (required)</i> A word in which the previous <i>plabel</i> is returned to the user's program. If no <i>plabel</i> existed previously, zero is returned.

CONDITION CODES

CCE	Request granted. Trap enabled.
CCG	Request granted. Trap disabled.
CCL	Request denied because of an illegal <i>plabel</i> .

ADDITIONAL DISCUSSION

"Enabling and Disabling Traps" in Section V.

ZSIZE

INTRINSIC NUMBER 136

Alters current Z to DB area.

SYNTAX

IIV

actsize := ZSIZE(*size*);

The ZSIZE intrinsic alters the size of the current Z to DB area by adjusting the register offset of the Z address from the DB address (Z to DB).

The ZSIZE intrinsic moves the Z address forward (expansion) or backward (contraction). If the Z to DB area size requested exceeds the maximum size permitted for the Z to DL (stack) area, only the maximum size allowed is granted.

All changes to the Z to DB area are made in increments or decrements of 128 words, and hence the *size* actually granted may differ from the size requested.

FUNCTIONAL RETURN

<i>actsize</i>	<i>integer (optional)</i> This intrinsic returns the size in words actually granted.
----------------	---

PARAMETERS

<i>size</i>	<i>integer by value (required)</i> An integer value greater than or equal to zero that specifies the desired register offset (in words) for Z to DB.
-------------	---

CONDITION CODES

CCE	Request granted.
CCG	The requested size exceeded the maximum limits of the Z to DL (stack) area. The maximum limit is granted, and this value is returned to the calling process as the value of <i>actsize</i> .
CCL	An illegal <i>size</i> parameter, less than $(S - DB) + 64$ words, was specified. This minimum value is assigned by default.

ADDITIONAL DISCUSSION

"Changing Stack Sizes" in Section V.

OPTIONAL CAPABILITIES

SECTION

III

Some MPE functions can only be performed by users with the optional capabilities mentioned in Section I. This section 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`.

Optional Capabilities

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` *options* 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` *options* 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.

Optional Capabilities

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 MAXPRI=BS. Normally, priority for a BS process is 100. However, by specifying a rank >0 in the GETPRIORITY intrinsic, the process may be set in the Master Queue at *min* 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 MPE V Commands Reference Manual (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 MPE V Commands Reference Manual (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

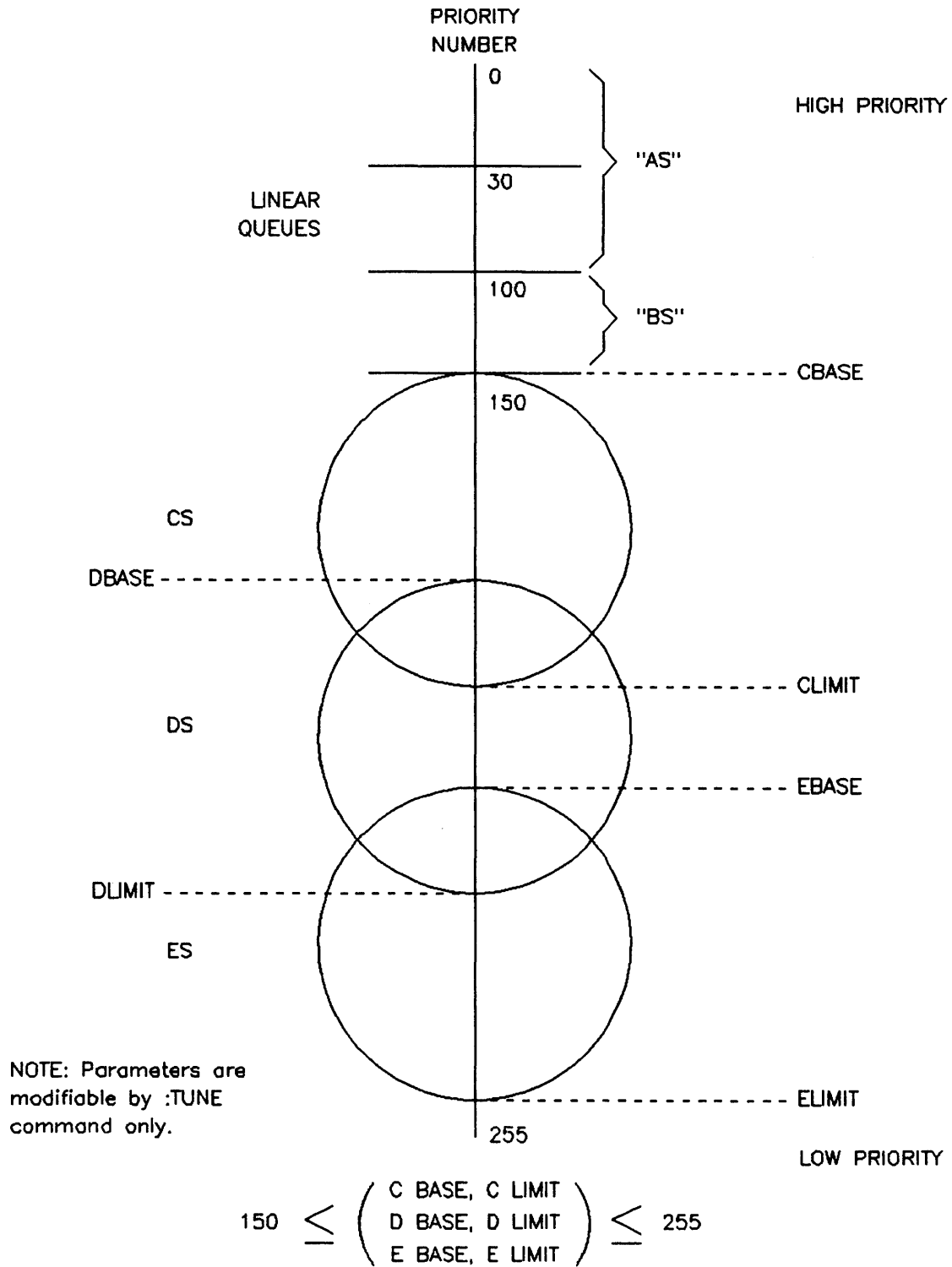


Figure 3-2. Master Queue Structure

Optional Capabilities

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 disc I/O, terminal I/O, pre-emption), its new priority is determined so that it may be requested 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.

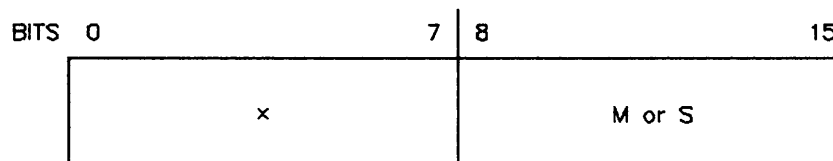
The priority class of a process can be specified by the user with PH capability. In the two-character string that comprises a priority class reference, the first character refers to the location of a subqueue within the Master Queue (in alphabetical order) and the second character specifies whether the 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:

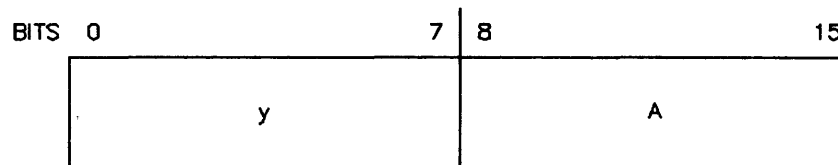
- A string of two ASCII characters describing the standard priority class (subqueue) in which the process is to be scheduled: AS, BS, CS, DS, or ES.
- An ASCII character or characters (*x*) specifying a valid subqueue, followed immediately by the single-character string M, indicating the Master Queue, 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.

Optional Capabilities

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 Figures 3-3, 3-4, and 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.

Optional Capabilities

```

PAGE 0001  HEWLETT-PACKARD 32100A.08.1  SPL[4W]  TUE, OCT 28, 1982, 4:35 PM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1  INTEGER IN,OUT,LGTH,MONTH,DAY,YEAR,DATE:=1,DSLGTH:=372;
00006000 00005 1  LOGICAL DSINDX;
00007000 00005 1  ARRAY READ(0:14):="GENERATE CALENDAR DATA SEGMENT";
00008000 00017 1  ARRAY BUFR(0:1):=2(" ");
00009000 00001 1  BYTE ARRAY BBUF(*)=BUFR;
00010000 00001 1  ARRAY REQST(0:5):="ENTER YEAR: ";
00011000 00006 1  INTEGER ARRAY MAXDAY(0:11):=31,28,31,30,31,30,
00012000 00006 1                                     31,31,30,31,30,31;
00013000 00014 1  INTEGER ARRAY CALENDAR (0:371):=372(-1);
00014000 00001 1      DEFINE CCL = IF < THEN QUIT#,
00015000 00001 1      CCNE= IF <> THEN QUIT#;
00016000 00001 1
00017000 00001 1  INTRINSIC FOPEN,FREAD,FWRITE,GETDSEG,DMOVOUT,BINARY,QUIT;
00018000 00001 1
00019000 00001 1  <<END OF DECLARATIONS>>
00020000 00001 1
00021000 00001 1      IN:=FOPEN(INPUT,%45); CCL(1);      <<$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,DSLGTH,"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 UNTIL MAXDAY(MONTH)-1 DO<<INDEX DAYS/EA MO>>
00036000 00127 1              BEGIN
00037000 00132 2                  CALENDAR(MONTH*31+DAY):=DATE; <<SET JULIAN DATE>>
00038000 00140 2                  DATE:=DATE+1                <<INCR JULIAN DATE>>
00039000 00141 2              END;
00040000 00143 1
00041000 00143 1          DMOVOUT(DSINDX,0,372,CALENDAR);    <<JULIAN 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, 1982, 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)

Optional Capabilities

```

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  BYTE ARRAY NAME(0:7):="TERMIO# ";
00004000 00005 1  BYTE ARRAY DEV(0:4):="TERM ";
00005000 00004 1  INTEGER FNO,LGTH,MONTH,DAY,DSLGTH,JDATE,CURRENT:=1;
00006000 00004 1  LOGICAL 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;
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. 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 00005000 in Figure 3-3).
<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:

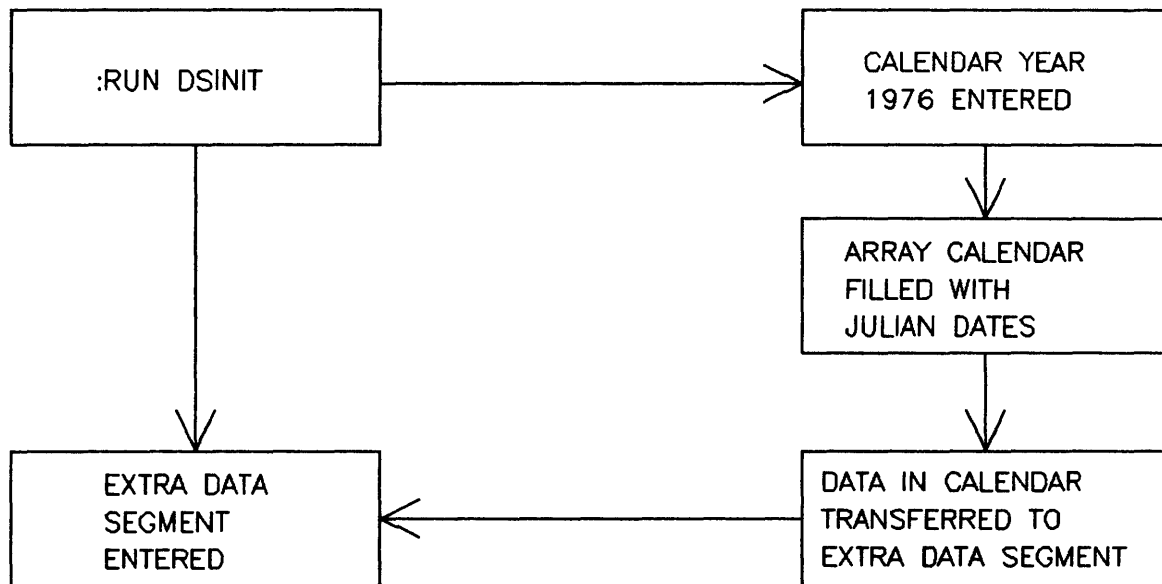
```
DMOVOUT (DSINDEX,0,372,CALENDAR)
```

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.

Optional Capabilities

At this point, the following events have occurred:



When the `:RUN DSBOS` command is entered, the program in Figure 3-4 (DSBOS) 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);
```

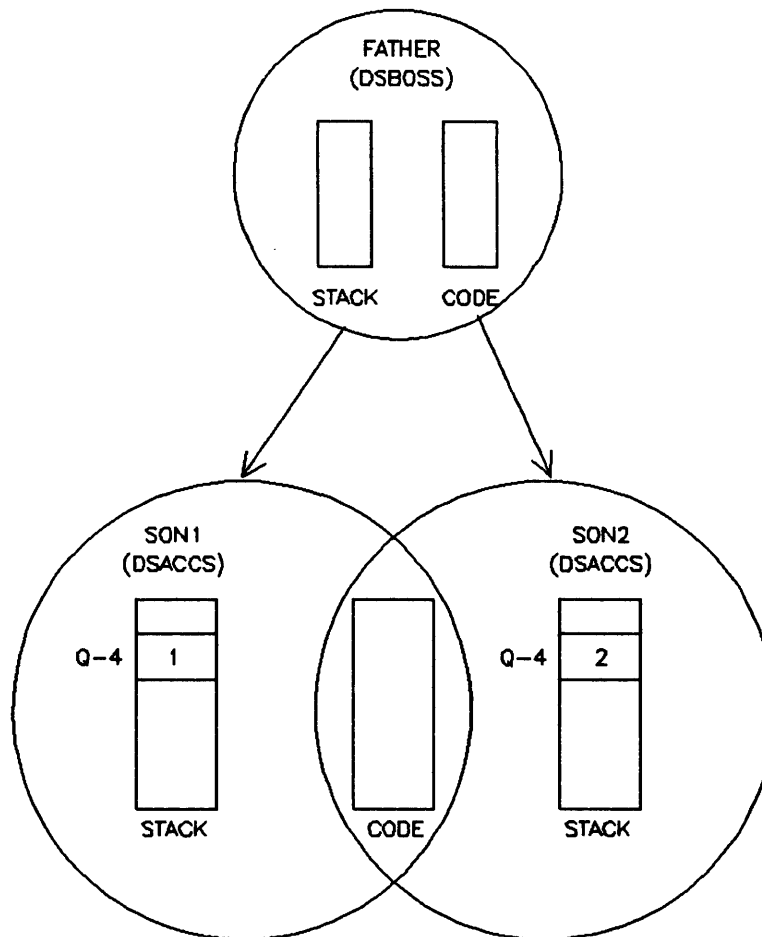
creates a process. The parameters specified are:

<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 **DSBOSS**, 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.

Optional Capabilities

Statement 00017000 of Figure 3-5:

```
FND:=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(G):=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 disc files, where each formal designator must be unique in its domain (temporary or permanent), two or more devices can be opened with the same formal designator. For example, 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 36, specifying 36 words.

device DEV, a byte array specifying the device class ("TERM").

All other parameters are omitted from the FOPEN intrinsic call.

The shared data segment is acquired with the statement:

```
GETDSEG(DSINDX,DSLGT,"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(DSINDX,(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 Figures 3-3, 3-4, and 3-5:

EXAMPLE 1:

```
:RUN DSINIT
```

```
GENERATE CALENDAR DATA SEGMENT
```

```
ENTER YEAR: 1983
```

```
END OF PROGRAM
```

```
:RUN DSBOSS
```

```
WHEN ALL JULIAN DATE OPERATIONS ARE COMPLETE TYPE: DONE.
```

```
? DONE
```

```
END OF PROGRAM
```

Optional Capabilities

EXAMPLE 2:

```
:DATA FIELD.SUPPORT;TERMI01
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;TERMI02
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 DSB0SS command causes DSB0SS to execute. DSB0SS 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 TERMI01 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 section.

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 section.

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, disc storage for a new segment is obtained by calling the `GETDSEG` intrinsic. Sufficient virtual space is allocated by the system to accommodate the original length of the data segment. This virtual space 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 disc storage is obtained, you can use the `ALTDSEG` intrinsic to reduce the storage required by the segment when it is moved into main memory, then later expand it as needed for more efficient use of memory. In no case, however, can `ALTDSEG` be used to increase segment size beyond the virtual space originally allocated through `GETDSEG`.

The form of the `ALTDSEG` intrinsic call is:

```
ALTDSEG<INDEX, INC, SIZE>;
```

The parameters specified are:

<i>index</i>	<code>INDEX</code> is a word containing the logical index of the extra data segment, obtained through the <code>GETDSEG</code> intrinsic call.
<i>inc</i>	<code>INC</code> 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>	<code>SIZE</code> 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 section. 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, 1982, 4:35 PM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1  INTEGER IN,OUT,LGTH,PIN;
00006000 00005 1  ARRAY REQST(0:8):=%6412,"PROGRAM FILE = ";
00007000 00011 1  ARRAY NAME(0:13);
00008000 00011 1  BYTE ARRAY BNAME(*)=NAME;
00009000 00011 1  ARRAY CRMSG(0:6):="CREATE PROCESS";
00010000 00008 1  ARRAY ACTMSG(0:7):="ACTIVATE PROCESS";
00011000 00010 1  DEFINE CCL=IF < THEN QUIT#,
00012000 00010 1  CCG=IF > THEN QUIT#,
00013000 00010 1  CCNE=IF <> THEN QUIT#;
00014000 00010 1
00015000 00010 1  INTRINSIC FOPEN,FREAD,FWRITE,CREATE, ACTIVATE,QUIT;
00016000 00010 1
00017000 00010 1  <<END OF DECLARATIONS>>
00018000 00010 1
00019000 00010 1  IN:=FOPEN(INPUT,%45);  <<$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 3-6. 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 FILE = SPL.PUB.SYS
CREATE PROCESS
ACTIVATE PROCESS

HEWLETT-PACKARD 32100A.08.1 SPL[4W] TUE, OCT 28, 1982, 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 FILE = SEGDVR.PUB.SYS
CREATE PROCESS
ACTIVATE PROCESS
SEGMENTER SUBSYSTEM (C.0)
-USL \$OLDPASS
-PREPARE \$NEWPASS
-EXIT

PROGRAM FILE = \$OLDPASS
CREATE PROCESS
ACTIVATE PROCESS
* TEST PROCESS EXECUTING *

PROGRAM FILE = (RETURN)

END OF PROGRAM
:

When SPL.PUB.SYS is entered in response to the "PROGRAM FILE=" request, the program displays:

CREATE PROCESS

ACTIVATE PROCESS

then suspends itself and the SPL compiler subsystem is accessed. (This process has been created and activated because of the SPL.PUB.SYS response by the user.)

A short program is entered from the terminal and the SPL compiler is exited, reactivating PROG at the statement following the ACTIVATE intrinsic call, and causing the "PROGRAM FILE=" message to be displayed again.

Optional Capabilities

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 FILE=" request and the program terminates.

The example below uses PROG to create and activate a duplicate of itself:

```
:RUN PROG
```

```
PROGRAM FILE = PROG  
CREATE PROCESS  
ACTIVATE PROCESS
```

```
PROGRAM FILE = PROG  
CREATE PROCESS  
ACTIVATE PROCESS
```

```
PROGRAM FILE = RETURN **3**
```

```
PROGRAM FILE = RETURN **2**
```

```
PROGRAM FILE = RETURN **1**
```

```
END OF PROGRAM  
:
```

When **PROG** is entered in response to the "PROGRAM FILE=" 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 FILE =
```

A carriage return (see **(RETURN) **3**** in the example) causes this process to stop executing and control returns to **PROG1**. **PROG1** displays:

```
PROGRAM FILE =
```

A second carriage return (see **(RETURN) **2**** in the example) causes this process to stop executing and control returns to **PROG**.

PROGRAM FILE = 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.

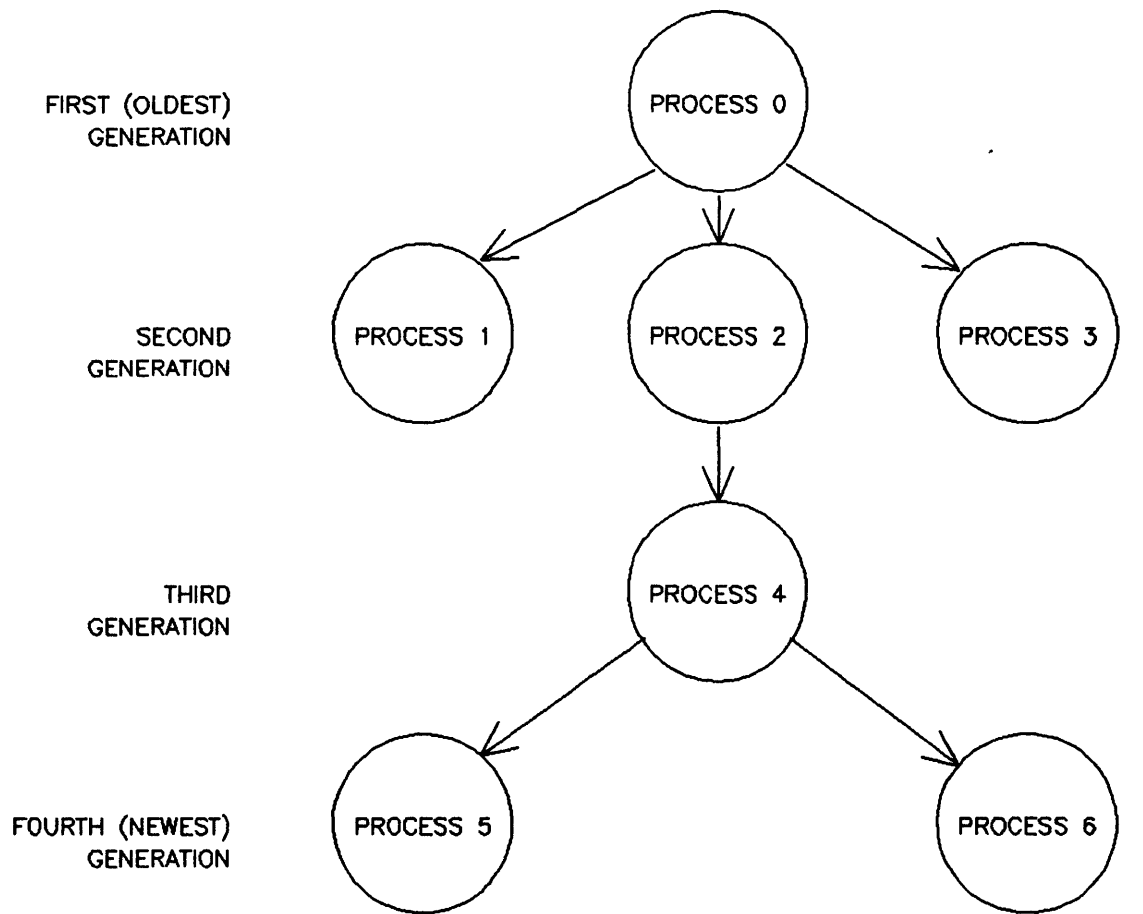
The intrinsic call:

```
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 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 3-7. 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.)

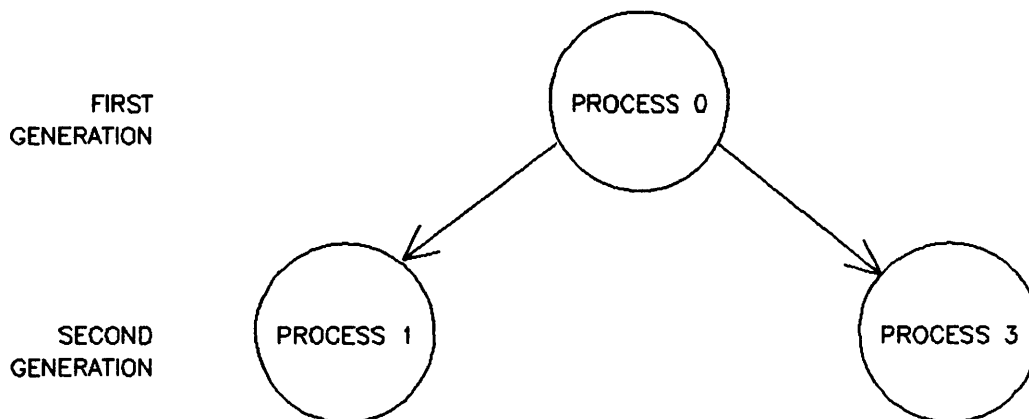


Figure 3-7. Process Deletion

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.

Interprocess Communication

You can direct the communication of information between processes. This information transfer, however, is restricted to upward or downward paths through the process tree structure, so that any process can communicate only with its father or sons. Between any father/son pair, only one such transfer is allowed at any particular time. 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.
<i>waitflag</i>	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.

Optional Capabilities

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);
```

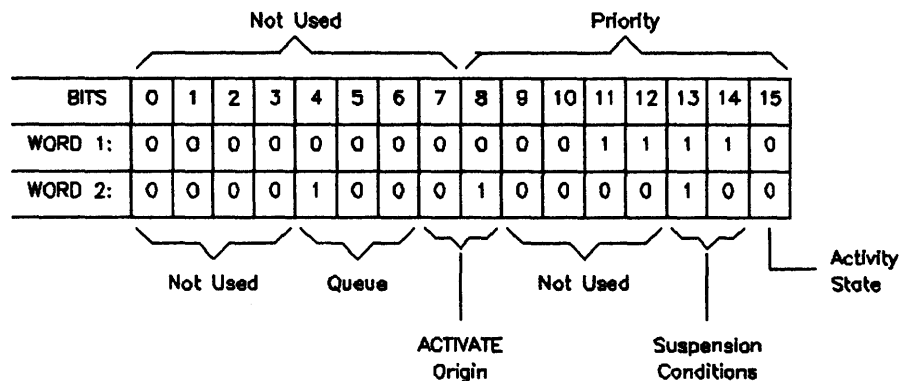
Determining Process Priority and State

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 Section II). 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:rinnum`

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, 1982, 4:35 PM
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1  BYTE ARRAY INPUT(0:5):="INPUT ";
00004000 00004 1  BYTE ARRAY OUTPUT(0:6):="OUTPUT ";
00005000 00005 1  BYTE ARRAY NAME(0:8):="BOOKFILE ";
00006000 00006 1  BYTE ARRAY PASSWD(0:7):="BOOKRIN ";
00007000 00005 1  INTEGER IN, OUT, BOOK, LGTH, ACCNO, RIN;
00008000 00005 1  LOGICAL DUMMY, COND:=TRUE;
00009000 00005 1  ARRAY BUFR(0:35);
00010000 00005 1  BYTE ARRAY BBUFR(*)=BUFR;
00011000 00005 1  ARRAY HEAD(0:13):="LIBRARY INFORMATION PROGRAM.";
00012000 00016 1  ARRAY REQUEST(0:7):=%6412,"ACCESSION NO: ";
00013000 00010 1  ARRAY CHANGE(0:9):="          NEW LOCATION: ";
00014000 00012 1  EQUATE RINBASE=2, RECD$'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/RECD$'PER'RIN);  <<COMPUTE RIN NO>>
00031000 00105 1          IF NOT(RINBASE<=RIN<=MAXRIN) THEN GO LOOP;<<BNDS CHK RIN>>
00032000 00117 1          LOCKGLORIN(RIN,COND,PASSWD);        <<LOCK FILE SUBSET>>
00033000 00123 1          FREADDIR(BOOK,BUFR,36,DOUBLE(ACCNO));CCL(7);<<READ DATA>>
00034000 00135 1          IF > THEN GO AGAIN;                  <<EOF, TRY AGAIN>>
00035000 00136 1          FWRITE(OUT,BUFR,36,0); CCNE(8);      <<DISPLAY DATA>>
00036000 00146 1          FWRITE(OUT,CHANGE,10,%320); CCNE(9);  <<REQUEST A CHANGE>>
00037000 00156 1          BUFR(19):=" ";
00038000 00161 1          MOVE BUFR(20):=BUFR(19),(16);        <<BLANK OLD LOCN>>
00039000 00167 1          LGTH:=FREAD(IN,BUFR(19),17); CCNE(10);<<READ NEW LOCN>>
00040000 00201 1          IF LGTH>0 THEN                        <<NEW LOCN ENTERED>>
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 3-8. 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 3-9. 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:

```
ACCND:=BINARY(BBUFR,LGTH);
```

The RIN to be locked is computed with the statement:

```
RIN:=RINBASE+(ACCND/RECD$ 'PER' RIN);
```

Optional Capabilities

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):

$$\begin{aligned} \text{RIN} &= 2 + (3/4) \\ &= 2 + 0 \end{aligned}$$

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:

```
FWRITEDIR(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 disc 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 LOCKLOCRIN intrinsic call within that process. When this is done, other processes within the job that attempt to lock this RIN are suspended until the locked RIN is released.

For example, to lock RIN number 6 (acquired by GETLOCRIN) unconditionally, the following call could be used:

```
LOCKLOCRIN(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.

Optional Capabilities

To unlock the same RIN, the intrinsic call:

```
UNLOCKLOCRIN(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 LOCKLOCRIN and UNLOCKLOCRIN 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,...);
•	•
•	•
•	•
GETLOCRIN(3);	LOCKLOCRIN(1,TRUEVAL);
FWRITE(LP,...);	FWRITE(LP,...);
LOCKLOCRIN(1,TRUEVAL);	•
CREATE(DESCEND,...);	•
FWRITE(LP,...);	FWRITE(LP,...);
•	•
•	•
•	•
UNLOCKLOCRIN(1);	UNLOCKLOCRIN(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:

```

equate whichsonrin = 1,
        synchronin  = 2,
        waitforfather= 1,
        waitforson   = 2,

```

FATHER PROCESS

```

•
•
•
soncount := 0;
while soncount <= maxsons do
begin
    SUSPEND(waitforson, synchronin);
    LOCKLOCRIN (synchronin);
    owner := LOCRINOWNER (whichsonrin);
    •
    •
    •
    soncount := soncount + 1;
    ACTIVATE (owner);
end;
•
•
•

```

SON PROCESS

```

•
•
•
LOCKLOCRIN (whichsonrin)
LOCKLOCRIN (synchronin);
ACTIVATE (father);
SUSPEND (waitforfather, synchronin);
UNLOCKLOCRIN (whichsonrin);
•
•
•

```

FREEING LOCAL RINS. To free all local RINs currently reserved for your job, enter:

```
FREELOCRIN;
```

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 disc, tape, serial disc, 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 disc destination file (disc 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.

Optional Capabilities

- **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** obtains information from the logging buffer and passes it to the user.

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 disc buffer file on disc. (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 disc. 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 disc, 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 filename digits preceding this three digit numeric remain 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 disc buffer file (when logging to a serial device) or the disc destination file (when logging to disc) if:

- A user calls **FLUSHLOG**, **BEGINLOG**, **ENDLOG**, or **WRITELOG** in *mode*=2. (Refer to the **WRITELOG** intrinsic in Section II 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 disc 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.

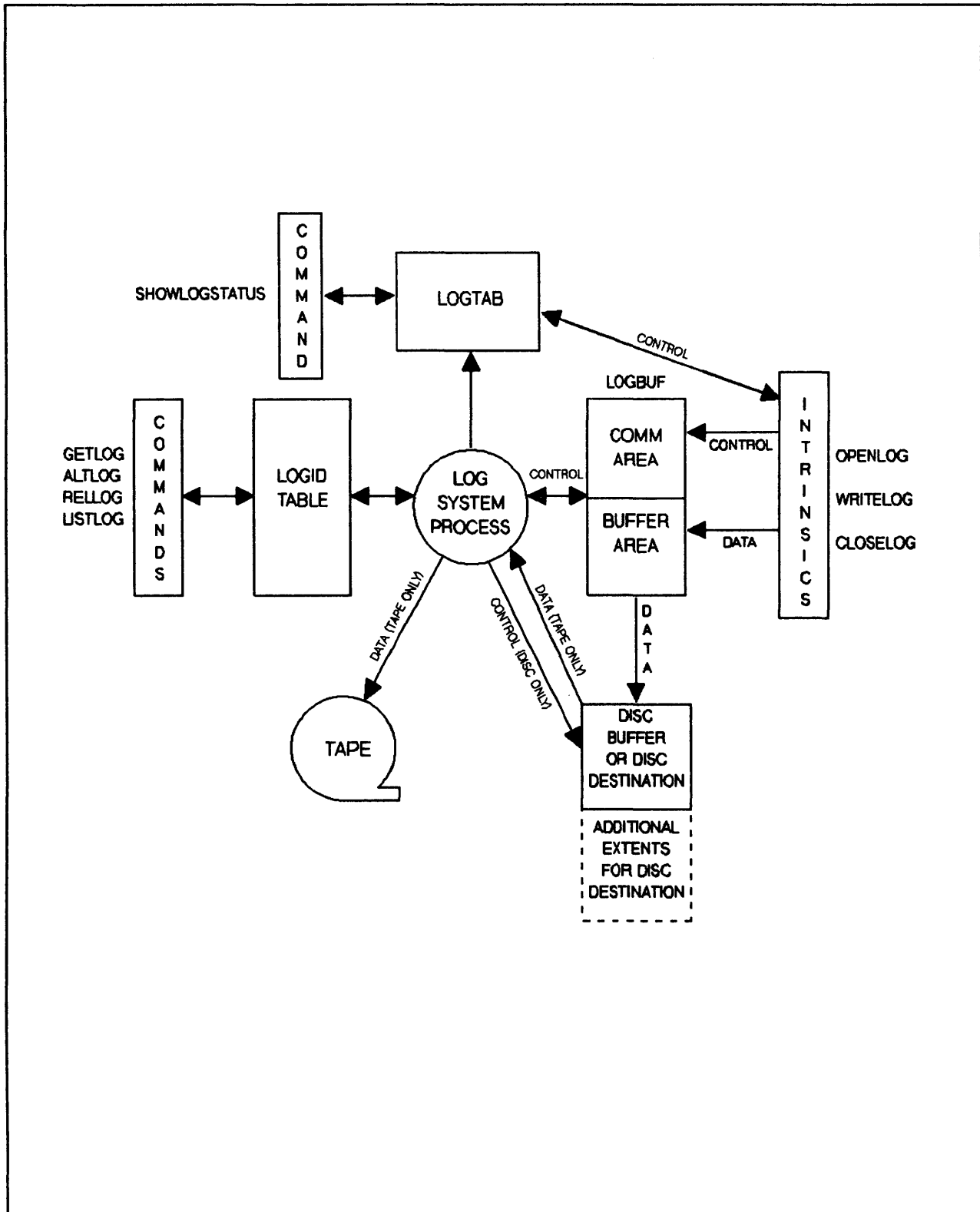


Figure 3-10. User Logging Facility

Optional Capabilities

When a disc file is specified, the logging process and logging buffer function differently. Logging records are loaded into the buffer area of LOGBUFF (the logging data segment). The records are moved to the disc destination file when the buffer area is full. A message is displayed on the Console when the destination file becomes full. The process will then stop as soon as the number of users equals zero. No other I/Os can access the file. The WRITELOG, BEGINLOG, ENDLOG, and FLUSHLOG intrinsics suspend the user process and activate the logging process. If needed, the logging process allocates additional extents (one at a time) to the disc destination file up to the maximum the user specified in the :BUILD command. Logging continues until the user-specified maximum is reached (same as EOF). Then the logging process reactivates the user process.

The user needs the record formats to directly access the logging files.

Logging Record Format:

record size = 128 words

user area = 119 words

LOG RECORD AT OPENLOG:

0	2	3	4	6	7	11	12	24	25	127
rec#	cksum	code	time	date	logid	log#	creator	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		

LOG RECORD AT CHANGELOG:

0	2	3	4	6	7	11	12	14	127
rec#	cksum	code	time	date	logid	seq num	c-time	c-date	

CRASH MARK:

0	2	3	4	6	7	127
rec#	cksum	code	time	date		

HEADER RECORD (START/RESTART):

0	2	3	4	6	7	11	127
rec#	cksum	code	time	date	logId		

TRAILER RECORD (STOP):

0	2	3	4	6	7	11	127
rec#	cksum	code	time	date	logId		

NULL RECORD:

0	2	3	4	6	7	127
rec#	cksum	code	time	date		

BEGIN TRANSACTION MARKER:

0	2	3	4	6	7	8	9	127
rec#	cksum	code	time	date	log#	len	user area	

END TRANSACTION MARKER:

0	2	3	4	6	7	8	9	127
rec#	cksum	code	time	date	log#	len	user area	

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

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

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 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 disc 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 disc file, build that file with the correct code (`;CODE=LOG`), and enough disc space to contain one day's output. Be generous with the disc file space. Divide the file into several extents; 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 following information:

- 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.

ACCESSING AND ALTERING FILES

SECTION

IV

This section 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). The following operations can be performed by using MPE file system intrinsics:

- 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 disc file with FLABELINFO.
- Read a user-defined label from a disc file with FREADLABEL.
- Write a user-defined label onto a disc file with FWRITELABEL.
- Update a record on a disc file with FUPDATE.
- Space forward or backward on a disc or tape file with FSPACE.
- Reset the logical record pointer to any logical record in a fixed-record length disc 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 disc 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 disc address if applicable, and accesses it for you. When the user program writes information to a particular file to be output on a device such as a line printer, again the program refers to the file by its formal file designator. MPE then automatically allocates the required device to that file. Throughout its existence, every file remains device-independent; that is, it is always referenced by the same formal file designator regardless of where it currently resides.

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 disc 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 section, 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. Disc 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 disc). For new files on disc, FOPEN specifies the number of user labels to be written.
- Allocating to the file the number of extents initially requested (for new disc files).
- Constructing the control blocks required by MPE for this particular access of the file. The information in these blocks is derived by merging specifications from five sources, listed below in descending order of precedence:
 1. The file label, obtainable only if the file is an old file on disc. This information overrides 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, pre-empting takes place according to the order of precedence shown above. To determine the specifications actually taking effect, the user can call the FGETINFO/FFILEINFO intrinsics, described later in this section. (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 disc 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 section explains what you can accomplish with files using the file system intrinsics. An attempt is made to show practical applications for the intrinsics, instead of merely restating the purpose of each intrinsic as discussed in Section II.

Internal Operations for File Accessing

Before a file can be used, it must be opened with the FOPEN intrinsic. If you are programming in SPL, you must call the FOPEN intrinsic 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 disc file. (A new disc file is a file that has not existed previously in the system.) One of the fundamental things that occurs at FOPEN time is that an access interface is created for the file. This access interface comprises a number of control blocks that are created, and which contain information about the file. In addition, if the new disc 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 disc.

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 disc or private volume disc 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 disc file label is shown below; once their values are established, they cannot be changed by subsequent FOPEN operations:

- File name.
- Sector address.
- Maximum number of logical records.
- Logical record size.
- Block size.
- *Foptions* (exception: disallow file equations 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 disc itself. In the simplified example of Figure 4-1 the file label is shown on the system disc. However, it could be on any disc in the system.

Since this is a new file, there is no information in the file. Therefore, the access mechanism is the only information the system has for this file. Depending on the FOPEN parameters specified, it is possible to write on this file.

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 disc 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 disc. 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 disc.

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 disc 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.

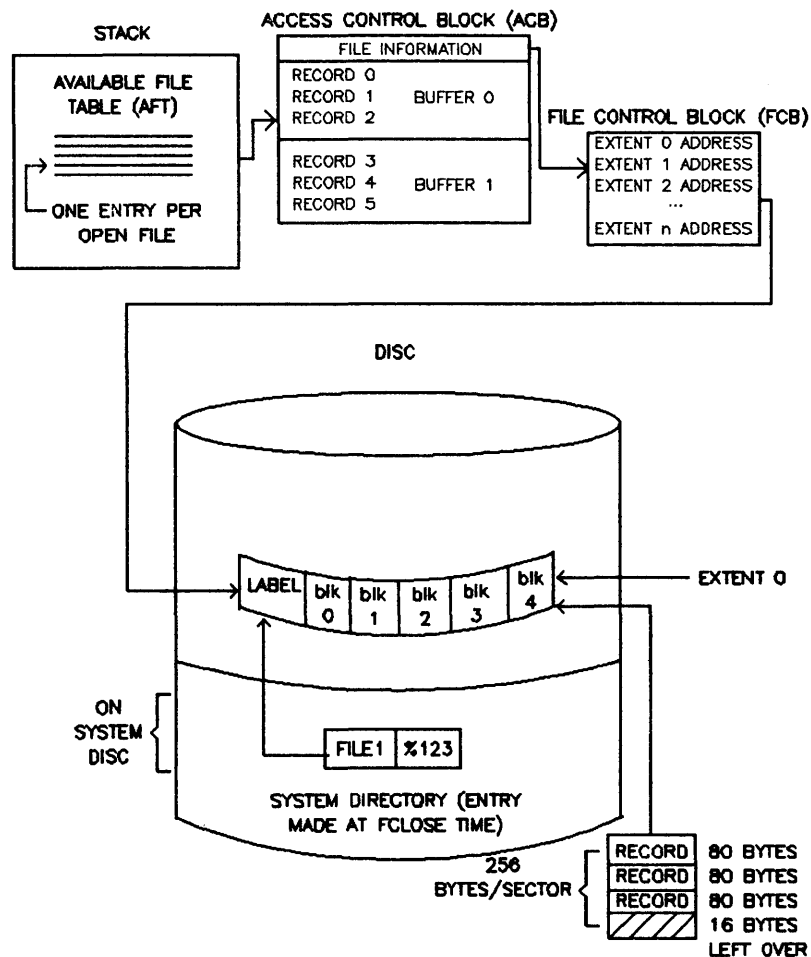


Figure 4-1. File Access Interface for New Disc Files

Accessing and Altering 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 disc file becomes a permanent file in the system. It can now be opened as an old disc 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 disc space for it. It is necessary for the system to establish a mechanism for the user to access the disc 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 disc.

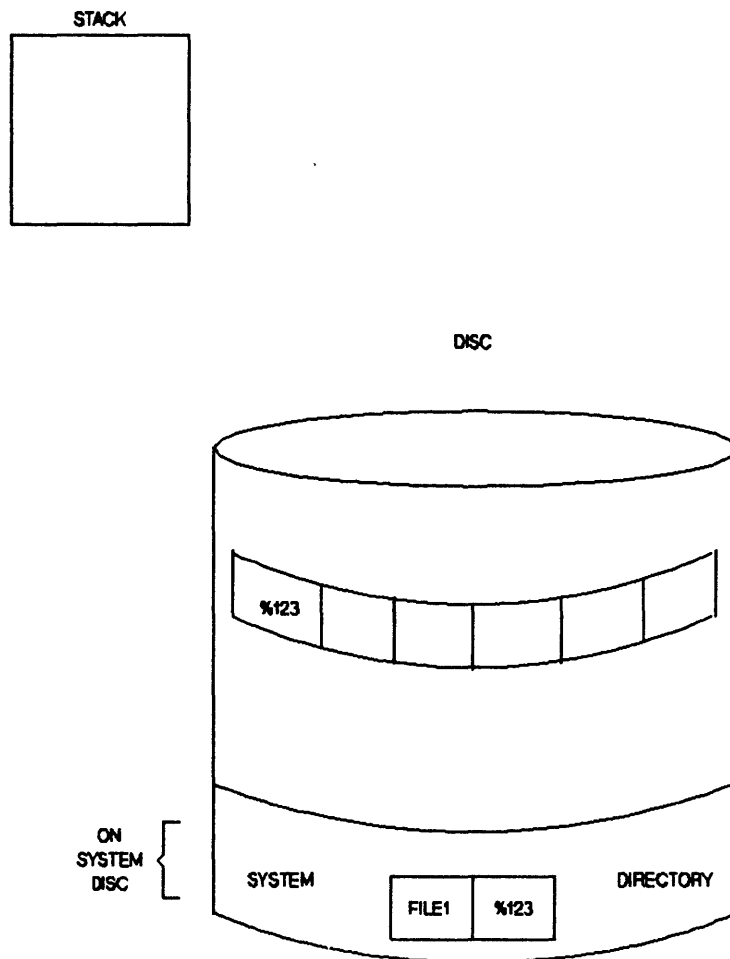


Figure 4-2. Directory File Name and Label Address Pointer

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 disc.

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.

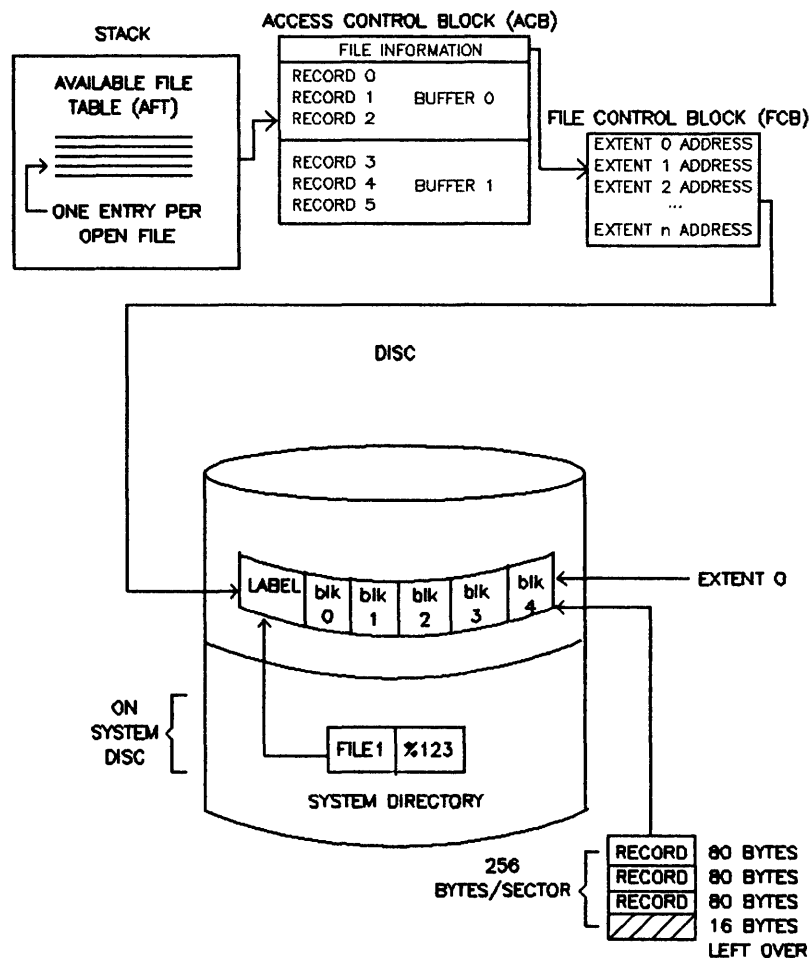


Figure 4-3. File Access Interface for Old Disc Files

File characteristics are obtained from different sources, depending on whether the file is a new disc file, an old disc file, or a file on a device other than disc.

For a new disc file, 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 disc 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.)

Accessing and Altering Files

Old permanent disc files are opened with `FOPEN` *foptions* bits (14:2)=01. Old temporary disc files are opened with `FOPEN` *foptions* bits (14:2)=10. In either case the file characteristics are obtained from the disc file label, `FOPEN` intrinsic parameters and default conditions. Some of these characteristics may be overridden by `:FILE` command parameters. The disc 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 disc, 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 disc file. (Discs 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 devicefiles. For this reason, the file system imposes a number of device-dependent restrictions on devicefiles. 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 disc in transit between the physical devices and the user program. Because data can be temporarily buffered in a disc file, the program assumes that all physical device files which it requires are constantly available to it. Input data typically is read and stored before a program requires it, and output data is delayed until the program's file operations are complete (at `FCLDSE` 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 disc 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.

Table 4-1. Device-Dependent Restrictions

INPUT ONLY DEVICES (SERIAL)

Card Reader/Paper Tape Reader

No carriage control

Undefined-length records

If card reader, ASCII only (can only read ASCII cards without using FCONTROL)

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 Disc 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

Table 4-2. Classification of Devices

DEVICE NAME	DEVICE TYPE NUMBER (OCTAL)	CLASSIFICATION
Moving-Head Disc	00	NEW or OLD
Flexible Disc	02	NEW or OLD
7911, 7912, 7914, 7933, or 7911/ 7912 Integrated Cartridge Tape Unit and 7945 Disc, 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.

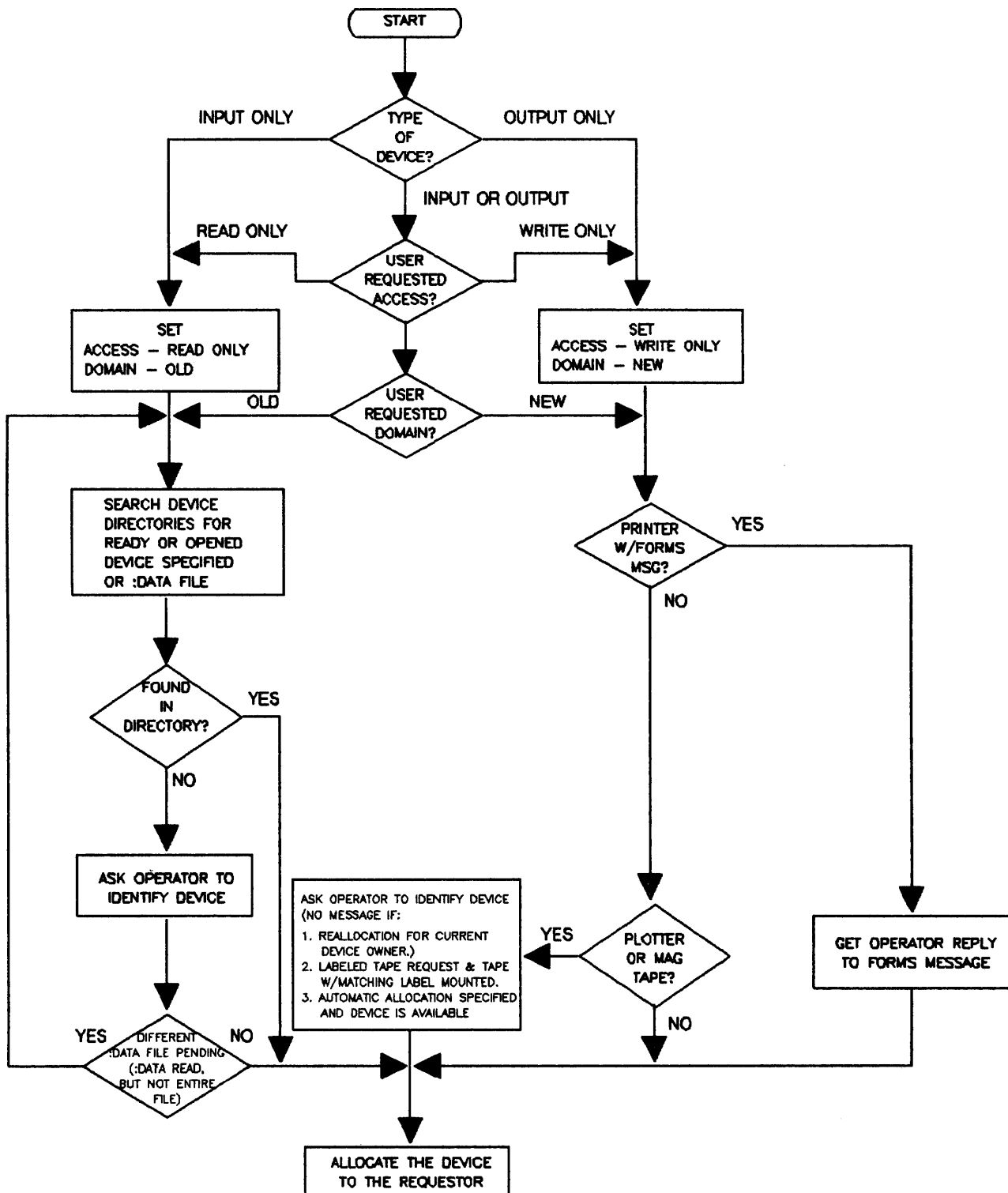


Figure 4-4. Device Allocation Flowchart

Accessing and Altering Files

Non-spooled devices can have only one file (for example, a card deck in the read hopper of a card reader), but spooled devices can have several file entries (for example, card decks which have been read in by the device and are stored as spoolfiles on disc to await access). Such devicefiles 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 disc spoolfile. A permanent :DATA file can be created on disc. This file can be made available to user programs by using the :STREAM command. This also results in a disc 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 discs only:
 1. The call to FOPEN specifies an ANSI or IBM labeled tape or serial disc, 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 disc 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 :DSLIN 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.

Accessing and Altering Files

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		1111111111222222222233	
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 Disc File

Figure 4-5 contains an SPL program which opens two files: a card reader file and a new disc file.

The second FOPEN call in Figure 4-5 (Statement 00019000):

```
OUT:=FOPEN(OUTPUT,%4,%101,128);
```

opens the new disc file. The parameters specified are:

formal designator DATAONE, which is contained in the byte array OUTPUT.

foptions %4, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	BINARY
															4	OCTAL

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 4-5. Opening a New Disc File

Opening an Old Disc File

Figure 4-6 contains an SPL program that opens three files: an old disc file, \$STDIN, and \$STDLIST.

Statement 00016000:

```
DFILE1:=FOPEN(DATA1,%5,%345,128);
```

opens the old disc 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.

recsize 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 RECO
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 ACCES
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 ACCES
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 4-6. Opening an Old Disc File

Accessing and Altering Files

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 FILERROR 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 Disc

Figure 4-7 contains an SPL program that opens a card reader file and a disc file, reads the contents of a card deck and writes the records read from the card deck into the disc file and, finally, closes the disc file as a permanent file.

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,QUI
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 4-7. Opening a File on a Device Other Than Disc

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:

formal designator Omitted.
Default: A temporary nameless file, that can be read but not saved, is assigned.

options %244, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	0	1	0	1	0	0	1	0	0	BINARY
								2		4			4			OCTAL

The above bit pattern specifies the following file options:

Domain: New file. No search of system or job temporary file directory is necessary. Bits (14:2) = 00.

ASCII/Binary: ASCII Bit (13:1) = 1.

Default Designator: \$STDIN. Bits (10:3) = 100.

Record Format: Undefined length. Bits (8:2) = 10.

aoptions Omitted. All bits are set to zero, access defaults to READ only.

All other parameters are omitted in the FOPEN intrinsic call.

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 FILEERROR is called.

The FILEERROR 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.

Accessing and Altering Files

formaldesignator

Omitted.

Default: A temporary nameless file, that can be written on but not saved, is assigned.

foptions

%614, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	1	1	0	0	0	1	1	0	0	BINARY
							6			1			4			OCTAL

The 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 FILERROR is called.

The FILERROR 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 4-8. 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 disc 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 disc file opened as a temporary file can be closed as temporary or saved as a permanent file.

When the FOPEN intrinsic opens a disc 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:

filenum Contained in the identifier DFILE2. The file number was assigned to DFILE2 when FOPEN opened the file.

disposition %2, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	BINARY
														2		OCTAL

The above bit pattern specifies the following:

Domain Disposition: Temporary job file (rewound). The file is retained in the user's temporary (job/session) file domain and can thus be 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 FILERROR(DFILE2,7);
```

checks the condition code. If the condition code is CCL, the error-check procedure FILERROR (statements 00011000 through 00017000 in Figure 4-9) is called. The FILERROR 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 USLINIT
00002000 00000 0 BEGIN
00003000 00000 1   BYTE ARRAY DATA1(0:7):="DATAONE ";
00004000 00005 1   BYTE ARRAY DATA2(0:7):="DATATWO ";
00005000 00005 1   ARRAY LABL(0:8):="EMPLOYEE DATA FILE";
00006000 00011 1   ARRAY BUFFER(0:127);
00007000 00011 1   INTEGER DFILE1,DFILE2,DUMMY;
00008000 00011 1   DOUBLE REC;
00009000 00011 1   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 OPERATIO
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 4-9. 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 disc file specified by OUT. The parameters specified are:

filenum Contained in the identifier OUT. The file number was assigned to OUT when the FOPEN intrinsic opened the file.

disposition %11, for which the bit pattern is as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	BITS
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	BINARY
											1			1		OCTAL

The above bit pattern specifies the following:

Domain Disposition: Permanent file. The file is saved in the system domain. If the file is a new or old temporary file on disc, an entry is created for it in the system file directory. (An error code is returned if a file of the same name exists already in the system directory.) If it is an old permanent file on disc, this disposition value has no effect. If the file is stored on magnetic tape, that tape is rewound and unloaded. Bits (13:3) = 001.

Disc Space Disposition: Unused disc space returned to the system. (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 4-10. 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 disc file, reads the card file, writes these records into the disc file, then closes the disc 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 FILERROR(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 FILERROR(IN,1);             <<CHECK FOR ERROR>>
00019000 00016 1       OUT:=FOPEN(OUTPUT,%4,%101,128);        <<NEW DISC FILE>>
00020000 00027 1       IF < THEN FILERROR(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 FILERROR(IN,3);             <<CHECK FOR ERROR>>
00024000 00045 1       IF > THEN GO END'OF'FILE;             <<CHECK FOR EOF>>
00025000 00046 1       FWRITE(OUT,BUFFER,LGTH,0);             <<COPY CARD TO DISC>>
00026000 00053 1       IF <> THEN FILERROR(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 FILERROR(OUT,5);            <<CHECK FOR ERROR>>
00031000 00072 1 END.

```

Figure 4-11. 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 *options* 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.

options %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.

Accessing and Altering Files

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,GETUSERMOD
00014000 00005 1 IOWAIT,QUIT;
00015000 00005 1 <<END OF DECLARATIONS>>
00016000 00005 1 OUT:=FOPEN(OUTPUT,4,1,,DEV); CCL(1);<<LINEPRINTER 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 TERM>
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);<<OUTPUT ? PROMPT
00025000 00064 2 IOWAIT(FILE); CCNE(6); <<COMPLETE REQUEST>>
00026000 00075 2 FREAD(FILE,BUFR(I*36),-72);CCNE(7);<<INPUT DATA-NOWA
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 N
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);<<OUTPUT ? PROMP
00044000 00207 2 IOWAIT(FILE); CCNE(13); <<COMPLETE REQUEST>>
00045000 00220 2 FREAD(FILE,BUFR(I*36),-72);CCNE(14);<<IN DATA-NOWAIT
00046000 00234 2 END;
00047000 00234 1 GO TO WAIT; <<CONTINUE>>
00048000 00235 1 EXIT;END.

```

Figure 4-12. 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.

OTHER APPLICATIONS OF MPE INTRINSICS

SECTION

V

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 `FMTCLOCK`.
- 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`.

Other Applications Of MPE Intrinsics

- 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:

procname Contained in the byte array `PNAME`. The contents of `PNAME` is the string "PROC1 ". Note that the string is terminated with a blank space.

lib 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,
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 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 CMD>>
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 CMD>>
00042000 00116 2              ACCUM:=ACCUM-OPERAND;                <<SUB CMD>>
00043000 00122 2              ACCUM:=ACCUM*OPERAND;                <<MUL CMD>>
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 5-1. 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).

Other Applications Of MPE Intrinsics

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:

comimage 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:

```
COMMAND(LGTH):=%15;
```

delimiters Omitted. The default delimiter array "COMMA, EQUAL, SEMICOLON, CARRIAGE RETURN" is used.

maxparms 1, specifying that one parameter is expected in *comimage*.

numparms Specified by PARMCNT. It contains the actual number of parameters entered with the command.

parms Specified by PARMINFO. An integer array to which the byte address of the parameter, entered as part of *comimage*, is returned.

NOTE

Although this parameter is listed as a double array in the specifications for the MYCOMMAND intrinsic in Section II, it is declared as a two-word integer array in this program because it is necessary to access each of the words individually. This is more convenient than declaring a one-word double array and a two-word integer array, then setting the two as equivalents.

dict Specified by TABLE, a byte array containing "5,3,"ADD", 5,3,"SUB", 5,3,"MUL", 5,3,"DIV", 5,3,"SET", 0".

The table specified by the *dict* parameter is searched until a match is found between the command name and an entry in the table. 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

Other Applications Of MPE Ininsics

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	:DSLIN	:MPLIN
:ACCEPT	:DSTAT	:MRJECONTROL
:ALLOW	:FILE	:NEWACCT
:ALTACCT	:FOREIGN	:NEWGROUP
:ALTGROUP	:GETLOG	:NEWUSER
:ALTJOB	:GETRIN	:NEWVSET
:ALTLOG	:GIVE	:OUTFENCE
:ALTSEC	:HEADOFF	:PTAPE
:ALTSPoolFILE	: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

Other Applications Of MPE Ininsics

: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 5-2 contains a program which must determine if the user is running the program in an interactive session. The statement:

```
WHO(INTERACTIVE);
```

calls the WHO intrinsic to make this determination. If the logical identifier INTERACTIVE is TRUE (bit (15:1) = 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 CMD>>
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 CMD>>
00042000 00116 2              ACCUM:=ACCUM-OPERAND; <<SUB CMD>>
00043000 00122 2              ACCUM:=ACCUM*OPERAND; <<MUL CMD>>
00044000 00126 2              ACCUM:=ACCUM/OPERAND; <<DIV COMMAND>>
00045000 00133 2              ACCUM:=OPERAND; <<SET COMMAND>>
00046000 00136 2          END;

```

Figure 5-2. 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 5-2. 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 0D (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 0D.
- *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 0D. 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 0D.
- 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 5-3, 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. 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 5-4.

```
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 5-3. 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));        <<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 5-3. 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<3000000D THEN GO LOOP; <<CONTINUOUS LOOP>>
00030000 00027 1 END.
PRIMARY DB STORAGE=%007;  SECONDARY DB STORAGE=%00033
NO. ERRORS=000;          NO. WARNINGS=000
PROCESSOR TIME=0:00:02;  ELAPSED TIME=0:00:26

```

Figure 5-4. 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 5-5, 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 CMD>>
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 CMD>>
00042000 00116 2                  ACCUM:=ACCUM-OPERAND;                    <<SUB CMD>>
00043000 00122 2                  ACCUM:=ACCUM*OPERAND;                    <<MUL CMD>>
00044000 00126 2                  ACCUM:=ACCUM/OPERAND;                    <<DIV COMMAND>>
00045000 00133 2                  ACCUM:=OPERAND;                        <<SET COMMAND>>
00046000 00136 2              END;

```

Figure 5-5. 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 5-5. 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 *table* 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 5-6.

```

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 CMD>>
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 CMD>>
00042000 00116 2                  ACCUM:=ACCUM-OPERAND;          <<SUB CMD>>
00043000 00122 2                  ACCUM:=ACCUM*OPERAND;          <<MUL CMD>>
00044000 00126 2                  ACCUM:=ACCUM/OPERAND;          <<DIV COMMAND>>
00045000 00133 2                  ACCUM:=OPERAND;                <<SET COMMAND>>
00046000 00136 2              END;

```

Figure 5-6. 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 5-6. 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 5-6, 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 10 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>	18 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	:SHOWJCV
: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	:SETJCV	: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.

Other Applications Of MPE Ininsics

All files still open by the process are closed with a disposition of 0. (For more details, refer to the FCLOSE intrinsic discussion in Section II.)

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 Section II 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 5-7 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 QUITPRG 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 Section II.)

```

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 5-7. 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 5-7. 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 QUITPRDG intrinsic call could be as follows:

```
QUITPRDG(1);
```

The parameter (1) can be any user-specified number. When the QUITPRDG intrinsic executes, this number is printed as part of the abort message. In addition, QUITPRDG sets the system Job Control Word, JCW, to the value %100000, with the QUITPRDG 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 disc 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 5-8.)

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 5-9 contains an SPL program that expands and contracts the DL to DB area.

NOTE

All addressing within the DL to DB area is DB-relative *negative* indexing. SPL is the only language which can access this area for you.

The program in Figure 5-9 reads data from \$STDIN and stores it in the DL area at progressively lower (DB-n) addresses. Additional DL space is allocated when the next buffer would lie outside the current DL area. When a null record (0 length) is read, the program outputs the data on a last-in-first-out (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 5-9 sets the DL to DB area to the original area assigned when the process was created (initial DL):

```
TOTALDL:=DLSIZE(0);
```

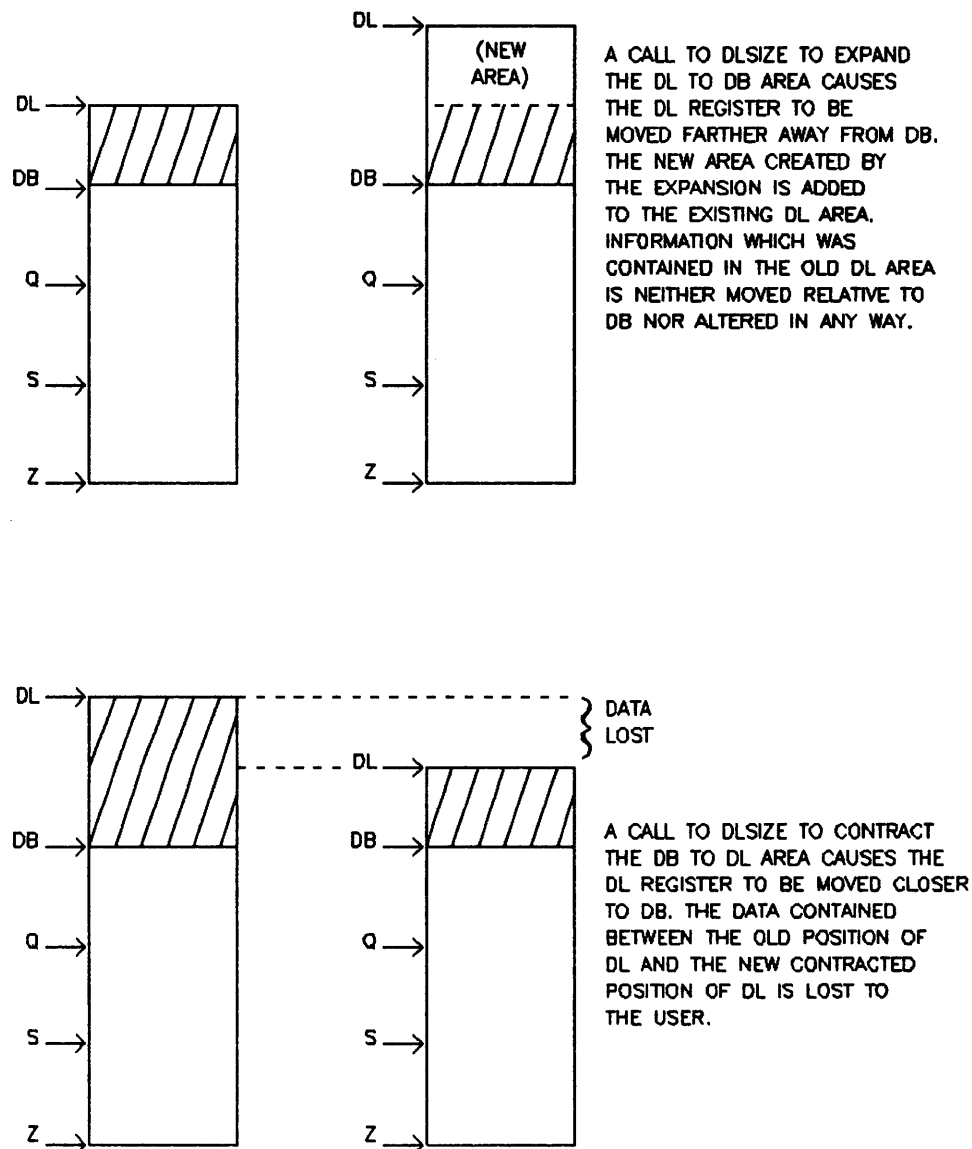


Figure 5-8. Expanding and Contracting the DL to DB Area

```

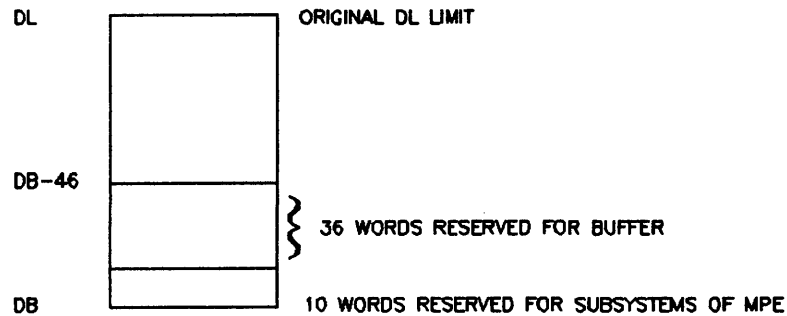
PAGE 0001 HP32100A.08.01 [4W] (C) HEWLETT-PACKARD COMPANY 1980
00001000 00000 0 $CONTROL USLINIT
00002000 00000 0 BEGIN
00003000 00000 1 INTEGER IN,OUT,LGTH,
00004000 00000 1 TOTALDL:=0;
00005000 00000 1 LOGICAL PROMPT:="? ";
00006000 00000 1 LOGICAL POINTER BUFFER:=-46; <<BUFFER:36 ; RESERV:10>>
00007000 00000 1
00008000 00000 1 INTRINSIC FOPEN,DLSIZE,FREAD,FWRITE,QUIT;
00009000 00000 1
00010000 00000 1 <<END OF DECLARATIONS>>
00011000 00000 1
00012000 00000 1 IN:=FOPEN(,%44); <<$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 5-9. Using the DLSIZE Intrinsic (Program DLAREA)

Other Applications Of MPE Ininsics

Consider the following illustration of the DL to DB area:



Statement 00006000 in Figure 5-9:

```
LOGICAL POINTER BUFFER:=-46;
```

sets a pointer to DB-46, which is the DB-relative address of the first word in BUFFER.

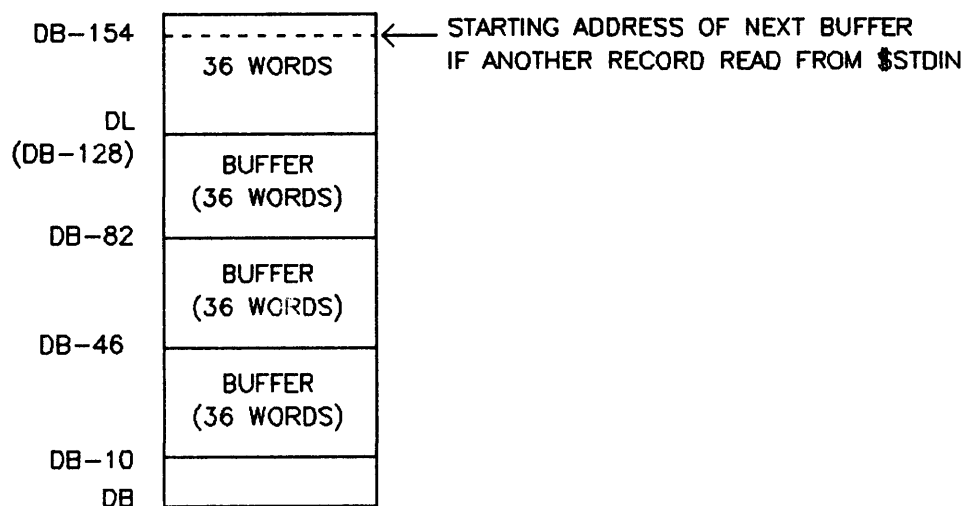
Statement 00021000:

```
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 5-10 shows the results when the program is run.

```
:RUN DLAREA  
? *****  
?      *****  
?      *****  
?      *****  
?      *****  
?      *****  
?      *****  
?      *****  
?      *****  
?      ***  
?      *  
  
      *  
    ***  
   *****  
  *****  
 *****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

(Enter at least one blank space and <CR>)

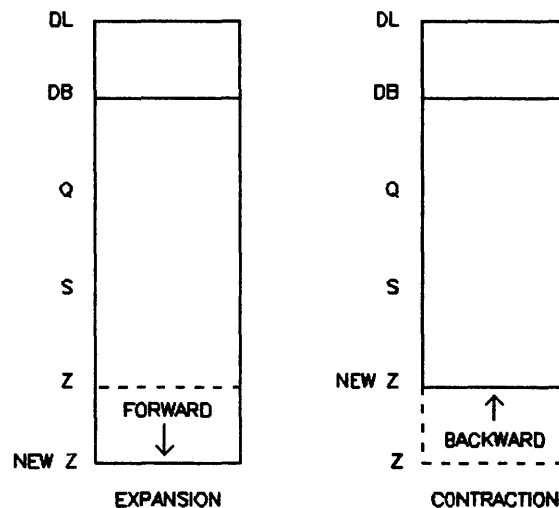
```
? EEEEE  
? HHHHH  
? TTTTT  
?  
TTTTT  
HHHHH  
EEEEE  
  
EEEEEE  
NNNNNN  
DDDDDD  
?:EOD  
  
END OF PROGRAM  
:
```

Figure 5-10. Changing the DL to DB Area Size (Program DLAREA)

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.

Other Applications Of MPE Ininsics

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 non-privileged), and on the code domain of the *plabel* and its mode (privileged or nonprivileged). The code domains are:

PRDG	(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 PRDG, GSL, or PSL, *plabel* must be nonprivileged in PRDG, GSL, or PSL.
- Privileged in PRDG, GSL, or PSL, *plabel* may be privileged or nonprivileged in PRDG, 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);
```

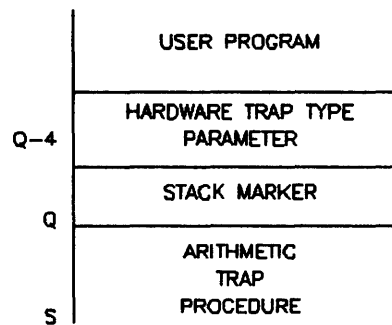
where STATE is true (bit (15:1) = 1) to enable the signals from all hardware traps, or false (bit (15:1) = 0) to disable these signals.

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.

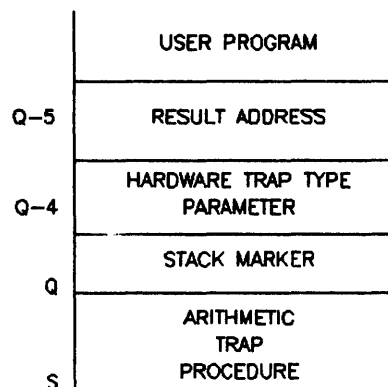
- Bit 15 = Floating Point Divide By 0
 14 = Integer Divide By 0
 13 = Floating Point Underflow
 12 = Floating Point Overflow
 11 = Integer Overflow

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

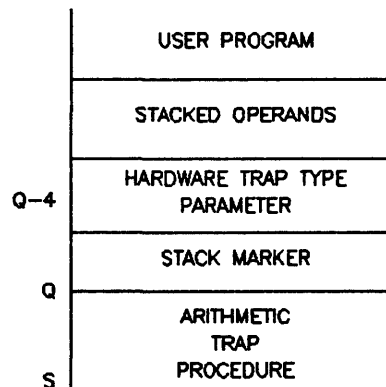
The address of the result operand is left on the stack in Q-5. An EXIT 2 return will resume execution in the user code domain at the instruction following the one which caused the trap. The condition of the stack when the trap procedure is invoked is:



COMMERCIAL INSTRUCTION TRAPS.

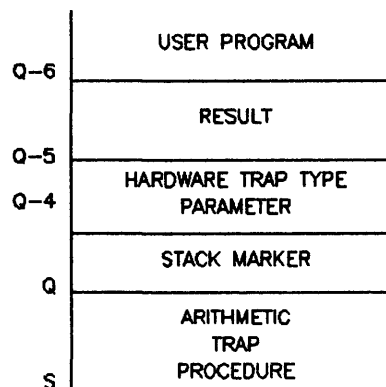
- Bit 7 = Decimal Overflow
- 6 = Invalid ASCII Digit (CVAD)
- 5 = Invalid Decimal Digit
- 4 = Invalid Source Word Count (CVBD)
- 3 = Invalid Decimal Operand Length
- 2 = Decimal Divide By 0

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 5-11. 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 5-11 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 5-11 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 5-11. Using the XARITRAP Intrinsic (Program ATRAP)

Other Applications Of MPE Intrinsics

Statement 00020000 of Figure 5-11 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

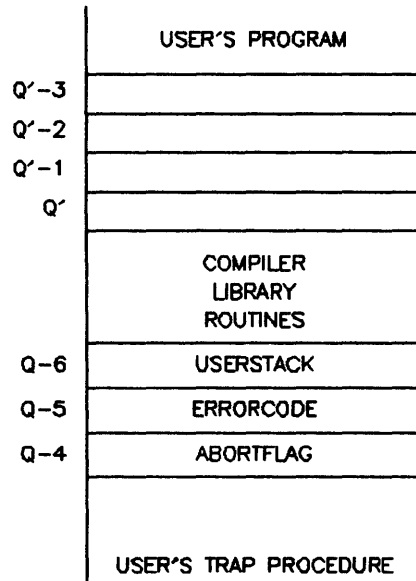
The software library trap reacts to major errors that occur during execution of procedures from the System Library. When a user program begins execution, this trap is disabled automatically. You can enable (or disable) it with the XLIBTRAP intrinsic. When enabled, the library trap passes control to a 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 HP 3000 Compiler Library Reference Manual (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:

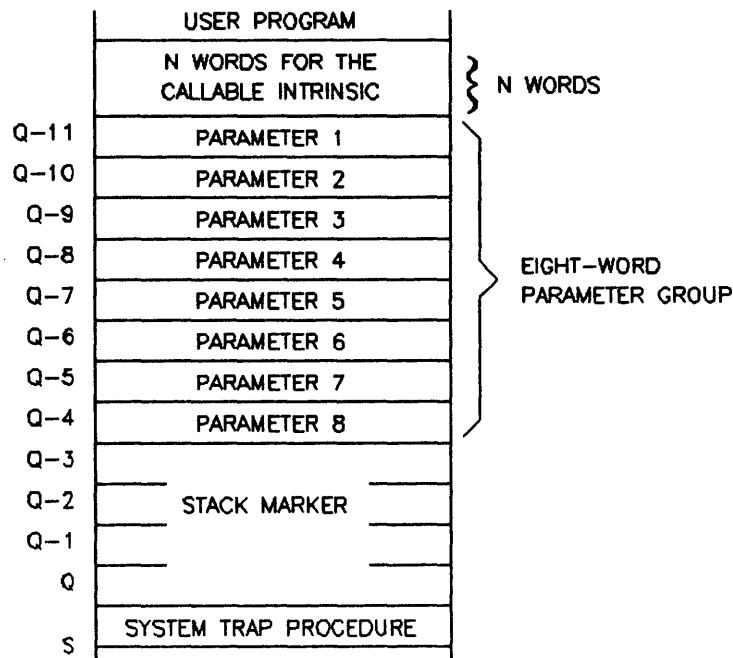
- **Illegal access.** An attempt by a user to access an intrinsic for which he does not have access capability.
- **Illegal parameters.** The passing to an intrinsic of parameters that are not defined for the user's environment.
- **Illegal environment.** The DB register is not currently pointing to 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.

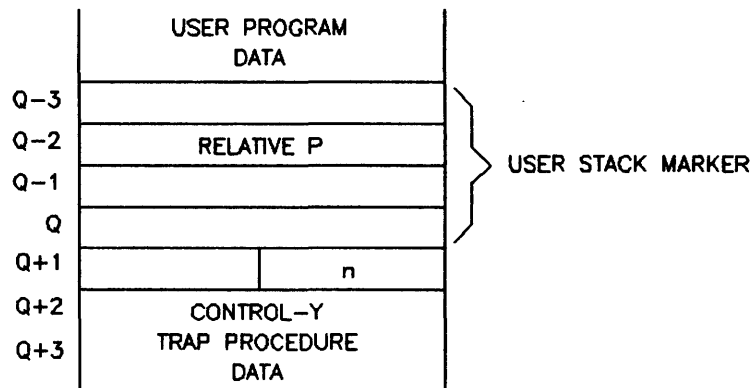
When more than one process is currently running within your process tree structure, the CONTROL-Y signal interrupts the last process to enable the trap.

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:

```
TDS:=%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.

Other Applications Of MPE Intrinsics

Figure 5-12 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:

```
TDS:=%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 5-12. Using the XCONTRAP Intrinsic (Program CONTY)

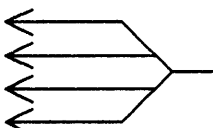
The results of executing the program are shown below:

```

: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.

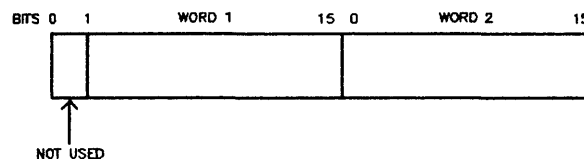
This quantity is reset to zero on 24-day intervals at 12 o'clock midnight. Detection and correction of this case between two calls to `TIMER` (less than 24 days apart) for computing an elapsed time interval can be done by adding 2,073,600,000 (the number of milliseconds in 24 days) to the result when subtracting a current `TIMER` count from a previous count with a negative result.

Figure 5-13 contains a program that uses the system timer bit count to generate a random octal number. This number then is converted to one of the ASCII characters (;, <=>?@, or A through Z.)

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:

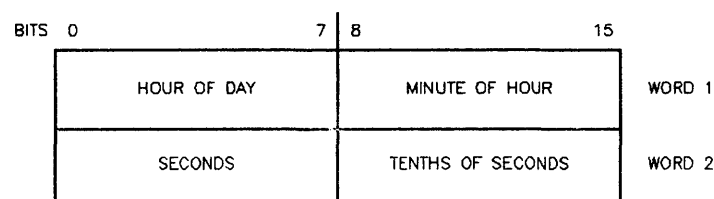


The `INTEGER` function strips Word 1 from this quantity, leaving a 16-bit integer value. The low-order five bits change value most rapidly, and are used to obtain a decimal number from 0 to 32.

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 5-13. Using the TIMER Intrinsic (1 of 2)

```

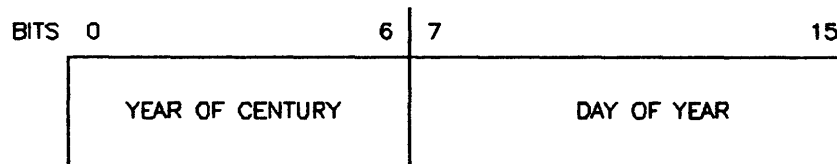
00047000 00202 1  NEXT:
00048000 00202 1      FWRITE(OUT,MSG,8,%320);          <<CONTINUE TEST?>>
00049000 00207 1      IF < THEN QUIT(11);              <<CHECK FOR ERROR>>
00050000 00212 1      FREAD(IN,BUFR(3),-1);            <<GET Y/N ANSWER>>
00051000 00220 1      IF < THEN QUIT(12);              <<CHECK FOR ERROR>>
00052000 00224 1      IF CBUF(6)="Y" THEN GO LOOP;      <<Y-CONTINUE TEST>>
00053000 00232 1  END.

```

Figure 5-13. 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 FMTCLOCK intrinsic. These intrinsics use the information returned by the CALENDAR and CLOCK intrinsics.

The program shown in Figure 5-14 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 5-14. Using the FMTCALENDAR, FMTCLOCK, and FMTDATE Intrinsic

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.

Other Applications Of MPE Intrinsics

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.

Other Applications Of MPE Ininsics

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 CAT <i>nn</i> , 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

Other Applications Of MPE Intrinsic

CATALOG), and installs the message system. Existing catalog is renamed CAT nn 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 disc 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 disc address of CATALOG in the system global area. This entry point may be used when the message system seems to be malfunctioning, but the catalog is intact. (For example, MPE is issuing "MISSING MSG SET= mm . MSG= nn " at terminals and the System Console.) This may be done while the system is running.
HELP	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 5-15 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).

```
:SPLPREP TEST,MSGTEST
```

```
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,PRINTF,GENMESSAGE,PRINT;
```

```
00010000 00000 1
```

```
00011000 00000 1 MOVE BUFF:="CATALOG ";
```

```
00012000 00016 1 FILENUM:=FOPEN(BUFF,5,%420);
```

```
00013000 00026 1 IF <> THEN PRINTF,GENMESSAGE(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 5-15. GENMESSAGE Intrinsic Example

MPE DIAGNOSTIC MESSAGES

APPENDIX

A

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.

MPE Diagnostic Messages

- 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:

p-field

Is the location of the last instruction executed.

s-field

Is output only if the abort occurred when executing code belonging to a library segment referenced by the user program. The field provides the instruction location within the library segment that initiated the abort.

Within the *p-field* and *s-field*, the parameters are:

<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 Tables A-1 through A-9. For a listing of File System Errors refer to the discussion on the FCHECK intrinsic in Section II. Guide (30000-90049).

MPE Diagnostic Messages

<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	} LOGIC ERROR IN THE PROGRAM
2	FLOATING POINT OVERFLOW	
3	FLOATING POINT UNDERFLOW	
4	INTEGER DIVIDE BY ZERO	
5	FLOATING POINT DIVIDE BY ZERO	
6	PRIVILEGED INSTRUCTION	
7	ILLEGAL INSTRUCTION	
8	EXTENDED PRECISION OVERFLOW	
9	EXTENDED PRECISION UNDERFLOW	
10	EXTENDED PRECISION DIVIDE BY ZERO	
11	DECIMAL OVERFLOW	
12	INVALID ASCII DIGIT	
13	INVALID DECIMAL DIGIT	
14	INVALID WORD COUNT	
15	INVALID DECIMAL OPERAND LENGTH	
16	DECIMAL DIVIDE BY ZERO	
17	STT UNCALLABLE	
18	PROCESS QUIT=<number>	<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.)
19	PROGRAM QUIT=<number>	
20	STACK OVERFLOW	Logic error in the program. Probably looping and adding to the stack. May require larger MAXDATA when preparing program.
21	PROGRAM KILLED	Program aborted from an external source.
22	INVALID STACK MARKER	} Possible logic error in program.
23	ADDRESS VIOLATION	
24	BOUNDS VIOLATION	
25	NONRESPONDING MODULE	} Possible hardware problem.
26	DATA PARITY	
27	MEMORY PARITY	
28	SYSTEM PARITY	
29	STACK UNDERFLOW	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.
30	CST VIOLATION	
31	STT VIOLATION	

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

Table A-2. "ERROR:INTRINSIC" Message Numbers (Continued)

MESSAGE NO.	INTRINSIC	MESSAGE NO.	INTRINSIC
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	FWRITE LABEL	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	CREATEPROCESS
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	DMOVEDOUT
56	CAUSEBREAK	133	DMDVIN
60	TERMINATE	134	ALTDSEG
61	CTranslate	135	DLSIZE
62	BINARY	136	ZSIZE
63	ASCII	139	SWITCHDB
64	READ	180	JOBINFO
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 Message

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-9. "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:

FROM/{_S^J}*num,username.acctname/message*

The parameters have the following meanings:

<i>num</i>	The job/session number.
<i>username acctname</i>	The names of the transmitting job/session and user, and 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 preempt 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 preempt 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-10. System Messages

<p>CAN'T INITIATE NEW SESSIONS NOW</p> <p>A new session cannot be initiated due to one of the following problems:</p> <ol style="list-style-type: none"> 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. <div style="border: 1px solid black; padding: 5px; text-align: center; margin: 10px auto; width: 100px;"> NOTE </div> <p>System Operators can bypass rejections due to 2 and 3 by supplying; HIPRI on the :HELLO and :JOB commands.</p>	
<p>*{<i>SESSION</i>} JOB</p>	<p>ABORTED BY SYSTEM MANAGEMENT*</p> <p>The job/session has been aborted by the System Operator through the appropriate command. An immediate logoff takes place.</p>
<p>*{<i>SESSION</i>} JOB</p>	<p>HAS EXCEEDED TIME LIMIT*</p> <p>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.</p>
<p>WARNING:</p>	<p>PRIORITY=XXX</p> <p>The priority passed to the CREATE intrinsic resulted in a conflict with another process, and the priority then assigned was XXX instead of the requested value.</p>
<p>LMAP NOT AVAILABLE</p> <p>LMAP of the process being created, or program file being run, is not available because the code segments are already loaded.</p>	
<p>**POWER FAIL</p> <p>Power failure has occurred and automatic restart is in progress. It is possible that a character has been lost due to a transmission error when the power failure occurred.</p>	

FILE INFORMATION DISPLAY

In addition to Command Interpreter and run-time (abort) error messages, certain file input/output errors result in the output of a file information display 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: <table> <tr> <td>Domain:</td><td>NEW - A new file. SYS - System file domain. JOB - Job temporary file domain.</td></tr> <tr> <td>File Type:</td><td>ASCII BINARY</td></tr> <tr> <td>Default file Designator:</td><td>FORMAL - Actual file designator is the same as the formal file designator. \$STDIN \$STDLIST \$STDINX \$NEWPASS \$OLDPASS \$NULL</td></tr> <tr> <td>Record Format:</td><td>F - Fixed length. V - Variable length. U - Undefined length. ? - Unknown format.</td></tr> <tr> <td>Carriage Control:</td><td>NOCCTL - None. CCTL - Carriage control character expected.</td></tr> <tr> <td>File Equation Option:</td><td>FEQ - :FILE allowed. DEQ - :FILE not allowed.</td></tr> <tr> <td>Labeled Tape Option:</td><td>NOLABEL - Not a labeled tape. LABEL - Labeled tape.</td></tr> </table>	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.
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: <table> <tr> <td>Access Type:</td><td>INPUT - Read access. OUTPUT - Write access. OUTKEEP - Write-only access, without deleting. APPEND - Append access. IN/OUT - Input and output access. UPDATE - Update access.</td></tr> <tr> <td>Multi-record Option:</td><td>NOMR - Single record access. MR - Multirecord access.</td></tr> <tr> <td>Dynamic Locking Option:</td><td>NOLOCK - No locking permitted. LOCK - Locking permitted.</td></tr> </table>	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.								
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.														

MPE Diagnostic Messages

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 *ldev* will be "virtual" rather than a physical device number. (Refer to *ldev* 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.

DEVICE CHARACTERISTICS

APPENDIX

B

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 II).

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 II, 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 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.

Device Characteristics

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-1.

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	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	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-1. 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-1. FCARD Intrinsic Example (3 of 3)

A

Aborting

- Processes, 2-228, 5-28
- Programs, 5-28
- Sessions, 2-5
- User-Process Structure, 2-229

ABORTSESS Intrinsic, 2-5

Accessing

- File Information, 2-85
- Files, 2-111
- Message System, 2-167
- Non-Sharable Devices, 4-2
- Process Information, 2-220
- User Logging Facility, 2-211

Access-Mode Options, Declaring, 4-38

Acquiring

- Global RINs, 3-38
- Local RINs, 2-176, 3-43

ACTIVATE Intrinsic, 2-7

Activating File Access Modes, 2-150

Activating Processes, 2-7, 3-24

Active Process Substates, 3-24

Adjusting Directory Space in USL File, 2-9

ADJUSTUSLF Intrinsic, 2-9

Allocating Devices, 4-12

ALTDSEG Intrinsic, 2-11

Altering, Size of Extra Data Segment, 2-11

Altering, Z to DB Area, 2-268

Aoption Bit Summary, 2-90

Arithmetic Traps, 5-38

ARITRAP Intrinsic, 2-13

Arrays, Searching, 5-4

ASCII Intrinsic, 2-14

ASCII Numeric String

- Converting to Binary Coded Value, 5-20

ASCII Reading Formats, B-4

Assigning \$STDIN/\$STDLIST to Files, 2-32

AS Subqueue, 3-6

Attributes, Determining User, 5-12

Available File Table (AFT), 4-6

Avoiding Deadlocks, 3-34

B

BEGINLOG Intrinsic, 2-16

Binary Code, Converting Numbers to ASCII Strings, 2-38, 5-16

BINARY Intrinsic, 2-18

BS Subqueue, 3-6

C

Calendar Date

- Formatting, 2-108, 2-110, 5-52

- Obtaining, 2-19, 5-52

CALENDAR Intrinsic, 2-19

Calling

- Intrinsics, Calling From Languages, 1-4

- Intrinsics, Calling From SPL, 1-1

- Process, Suspending, 2-213, 5-26

Capabilities

- Create Volumes, 1-9

- Data Segment Management, 1-8, 3-9

- Multiple Resource Identification, 1-8

- Optional, 1-7

- Privileged Mode, 1-8, 3-1

- Process Handling, 1-8, 3-22

- Programmatic Sessions, 1-8

- Required, 2-3

- User Logging, 1-8

- Volume Usage, 1-9

Card Reader, B-1

Carriage-Control Bytes, 2-135

Carriage-Control Codes, B-2

Carriage-Control Directives, 2-161

Catalog Message, 5-56

CAUSEBREAK Intrinsic, 2-20

Changing

- DB Register Pointer, 2-250

- DL to DB Area Size, 5-31

- Length of USL File, 2-51

- Process Priority, 2-178

- Size of Extra Data Segment, 3-22

- Stack Size, 5-30

- Z to DB Area Size, 5-37

CLEANUSL Intrinsic, 2-21

CLOCK Intrinsic, 2-23

CLOSELOG Intrinsic, 2-24

Closing

- Access to Logging Facility, 2-24

- Files, 2-65, 4-28

- New File as Permanent File, 4-31

- New File as Temporary File, 4-28

Code Segments, 3-9

Collecting Mail, 3-33

Column Image Reading Formats, B-4

COMMAND Intrinsic, 2-26

INDEX (Continued)

- Command Parameters, Formatting, 5-5
- Commercial Instruction Traps, 5-40
- Communication, Interprocess, 3-31, 5-54
- Condition Codes, 1-5, 2-3
- Contracting Area Between DL/DB, 2-43
- Control Code Values, 2-68
- Control Operations, 2-68, 2-75
- CONTROL-Y Traps, 5-46
- Converting ASCII Strings
 - To Binary Word, 2-18
 - To Double-Word Binary Value, 2-41
- Converting Binary Numbers
 - To ASCII String, 2-38
 - To Numeric ASCII String, 2-14
- Converting Calendar Date Formats, 2-108
- Converting Numbers From
 - ASCII String To Binary Value, 5-20
 - Binary Code To ASCII String, 5-16
- Converting Strings of Characters, 2-36
 - Between EBCDIC/ASCII, 2-36
 - Between EBCDIK/JIS, 2-36
- Copying
 - Extra Data Segment to Stack, 2-45
 - Stack to Extra Data Segment, 2-47
 - Input From Paper Tapes To
 - Disc File, 2-224
- CREATE Intrinsic, 2-27
- CREATEPROCESS Intrinsic, 2-32
- Create Volumes Capability, 1-9
- Creating
 - Extra Data Segment, 2-171, 3-10
 - Processes, 2-27, 2-32, 3-34
- CS Subqueue, 3-6
- CTranslate Intrinsic, 2-36
 - Translating Characters With, 5-22
- Current Time, Obtaining, 2-40, 5-50

D

- DAScii Intrinsic, 2-38
- Data Segment Mgmt Capability, 1-8, 3-9
- Date, Obtaining, 5-52
- DATELINE Intrinsic, 2-40
- DBINARY Intrinsic, 2-41
- DB Pointer, Changing, 2-250
- DB Pointer, Moving, 3-5
- Deactivating
 - File-Access Modes, 2-150
 - RIO Records, 2-74

- Deadlocks, 3-34
- DEBUG Intrinsic, 2-42
- Declarations
 - Intrinsic, 1-2
 - Procedure, 1-2
- Declaring Access-Mode Options, 4-38
- Define, Parameters, 2-208
- Deleting
 - Extra Data Segment, 3-21
 - Inactive Entries From USL File, 2-21
 - Processes, 2-194, 3-29
- Determining
 - Father Process, 3-35
 - PIN of Process Locking Local RIN, 2-200
 - Son Process, 3-36
 - Source of Activation Call, 2-177
 - Source of Process Activation, 3-35
 - Status of File Pairs, 2-146
 - User Access Mode/Attributes, 5-12
- Device Characteristics
 - Card Reader, B-1
 - Line Printer, B-1
 - Magnetic Tape, B-2
 - Terminals, B-3
- Devices
 - Allocation/Classification Of, 4-12
- Directory Space, Adjusting, 2-9
- Disable
 - Abort Stack Analysis Facility, 2-237
 - Hardware Arithmetic Traps, 2-13
 - Software Interrupts, 2-99
 - Software Library Trap, 2-266
 - System Trap, 2-267
 - User Software Arithmetic Trap, 2-262
- Disc Files,
 - File Label Information, 2-101
 - Opening, 4-20
- Displaying File Information, A-12
- DLSIZE Intrinsic, 2-43
- DL To DB Area Size, Changing, 5-31
- DMOVIN Intrinsic, 2-45
- DMOVOUT Intrinsic, 2-47
- Domains, File, 4-2
- Drive, Optical Mark Reader, 2-54
- DS Subqueue, 3-6
- Dump Stack to File, 2-243
- Duplicative File Pairs, 2-146
- Dynamic
 - Loading, 5-3
 - Load/Unload, Library Procedure, 5-2
 - Unloading, 5-3

INDEX (Continued)

E

Enabling

- Hardware Arithmetic Traps, 2-13
- Software Interrupts, 2-99
- Software Library Trap, 2-266
- Stack Analysis Facility, 2-241
- System Trap, 2-267
- User Software Arithmetic Trap, 2-262

Enabling/Disabling Traps, 5-37

End-of-File Indication, B-2

ENDLOG Intrinsic, 2-49

Entering

- Non-Privileged Mode, 3-5
- Privileged Mode, 2-180, 3-3

Error-Check Procedure Writing, 4-33

Error Messages, Types Of

- ACTIVATE Intrinsic, A-7
- CREATE Intrinsic, A-7
- LOADER Error/Warnings, A-8
- LOCKGLORIN Intrinsic, A-8
- MYCOMMAND Intrinsic, A-8
- Run-Time, A-7
- SUSPEND Intrinsic, A-7

Errors, Intrinsic Call, 1-5

ES Subqueue, 3-6

Executing MPE Commands Programmatically, 2-26, 5-11

Expanding Area Between DL and DB, 2-43

EXPANDUSLF Intrinsic, 2-51

Extended Precision Floating Point Trap, 5-39

Extra Data Segment

- Altering Size Of, 2-11
- Changing Size Of, 3-22
- Creating, 2-171, 3-10
- Deleting, 3-21
- Releasing, 2-144
- Transferring Data From Stack, 2-47, 3-21
- Transferring Data To Stack, 2-45, 3-21

F

FATHER Intrinsic, 2-53

Father Process, Determining, 3-35

FCARD Intrinsic, 2-54

- Using With Optical Mark Reader, B-3

FCHECK Intrinsic, 2-58

FCLOSE Intrinsic, 2-65

FCONTROL Intrinsic, 2-68

FDELETE Intrinsic, 2-74

FDEVICECONTROL Intrinsic, 2-75

FERRMSG Intrinsic, 2-84, 4-33

FFILEINFO Intrinsic, 2-85

FGETINFO Intrinsic, 2-88

File Designators, Parsing/Validating, 2-130, 4-15

File Device Relationships, 4-2

File Domains, 4-2

File Information, Displaying, A-12

File System Error-Check Procedure, 4-33

Files

- Accessing, 2-111
- Closing, 2-65, 4-28
- Closing New File as Permanent File, 4-31
- Closing New File as Temporary File, 4-28
- Device Relationships, 4-2
- Dynamically Lock, 2-104
- How to Use, 4-5
- Non-Sharable Devices, 4-4
- Opening, 4-3
- Opening New Disc Files, 4-17
- Opening On Device Other Than Disc, 4-22
- Renaming, 2-148

FINDJCW Intrinsic, 2-96

FINTEXIT Intrinsic, 2-98

FINTSTATE Intrinsic, 2-99

FLABELINFO Intrinsic, 2-101

FLOCK Intrinsic, 2-104

FLUSHLOG Intrinsic, 2-106

FMTCALENDAR Intrinsic, 2-108

FMTCLOCK Intrinsic, 2-109

FMTDATE Intrinsic, 2-110

FOPEN Intrinsic, 2-111

Foptions Bit Summary, 2-88

Formatting Calendar Dates/Time, 2-108, 5-52

Formatting Command Parameters, 5-5

FPARSE Intrinsic, 2-130

FPOINT Intrinsic, 2-133

FREAD Intrinsic, 2-135

- Using With \$STDIN/\$STDLIST, 4-23

FREADBACKWARD Intrinsic, 2-137

FREADDIR Intrinsic, 2-139

FREADLABEL Intrinsic, 2-141

FREADSEEK Intrinsic, 2-143

FREEDSEG Intrinsic, 2-144

Freeing Local RINs, 2-145, 3-45

FREELOCIN Intrinsic, 2-145

INDEX (Continued)

FRELATE Intrinsic, 2-146
FRENAME Intrinsic, 2-148
FSETMODE Intrinsic, 2-150
FSPACE Intrinsic, 2-153
Functional Return, 2-2
FUNLOCK Intrinsic, 2-155
FUPDATE Intrinsic, 2-156
FWRITE Intrinsic, 2-158
 Using With \$STDIN/\$STDLIST, 4-23
FWRITEDIR Intrinsic, 2-164
FWRITELABEL Intrinsic, 2-166

G

GENMESSAGE Intrinsic, 2-167
GETDSEG Intrinsic, 2-171
GETINFO Intrinsic, 2-173
GETJCW Intrinsic, 2-175
GETLOCRIN Intrinsic, 2-176
GETORIGIN Intrinsic, 2-177
GETPRIORITY Intrinsic, 2-178
GETPRIVMODE Intrinsic, 2-180
GETPROCID Intrinsic, 2-181
GETPROCINFO Intrinsic, 2-182
GETUSERMODE Intrinsic, 2-184
Global RINs
 Acquiring, 3-38
 Inter-Job Level, 3-38
 Locking, 2-196
 Locking/Unlocking, 3-39
 Releasing, 3-39
 Unlocking, 2-254

H

Hand-Shaking Arrangement, 3-37
How to Use Files, 4-5

I

Identifying Job/Session with JOBINFO, 5-14
Identifying Local RIN Owners, 3-44
Implementing Intrinsic Calls, 1-2
Initialize USL File to Empty State, 2-185
Initiate Completion of I/O Request, 2-186
Initiate Session on Specified Terminal, 2-245

INITUSLF Intrinsic, 2-185
Interactive File Pairs, 2-146
Inter-Job Level Global RINs, 3-38
Internal Operations For File Accessing, 4-5
Interprocess Communication, 3-24, 3-31, 5-54
Interprocess Local Level RINs, 3-43
Intrinsics

 Call Errors, 1-5
 Calling, 1-1
 Calling From SPL, 1-1
 Calls, Implementing, 1-2
 Declarations, 1-2
 Functions, 1-7
 Names, 2-1
 Numbers, 2-1
 Procedure Declarations, 1-2
 Time and Date, 5-49

Intrinsics, List Of

ABORTSESS, 2-5
ACTIVATE, 2-7
ADJUSTUSLF, 2-9
ALTDSEG, 2-11
ARITRAP, 2-13
ASCII, 2-14
BEGINLOG, 2-16
BINARY, 2-18
CALENDAR, 2-19
CAUSEBREAK, 2-20
CLEANUSL, 2-21
CLOCK, 2-23
CLOSELOG, 2-24
COMMAND, 2-26
CREATE, 2-27
CREATEPROCESS, 2-32
CTRANSLATE, 2-36
DASCH, 2-38
DATELINE, 2-40
DBINARY, 2-41
DEBUG, 2-42
DLSIZE, 2-43
DMOVIN, 2-45
DMOVOUT, 2-47
ENDLOG, 2-49
EXPANDUSLF, 2-51
FATHER, 2-53
FCARD, 2-54
FCHECK, 2-58
FCLOSE, 2-65
FCONTROL, 2-68
FDELETE, 2-74
FDEVICECONTROL, 2-75
FERRMSG, 2-84, 4-33

INDEX (Continued)

FFILEINFO, 2-85
 FGETINFO, 2-88
 FINDJCW, 2-96
 FINTEXT, 2-98
 FINTSTATE, 2-99
 FLABELINFO, 2-101
 FLOCK, 2-104
 FLUSHLOG, 2-106
 FMTCALENDAR, 2-108
 FMTCLOCK, 2-109
 FMTDATE, 2-110
 FOPEN, 2-111
 FPARSE, 2-130
 FPOINT, 2-133
 FREAD, 2-135
 FREADBACKWARD, 2-137
 FREADDIR, 2-139
 FREADLABEL, 2-141
 FREADSEEK, 2-143
 FREEDSEG, 2-144
 FREELOCRIN, 2-145
 FRELATE, 2-146
 FRENAME, 2-148
 FSETMODE, 2-150
 FSPACE, 2-153
 FUNLOCK, 2-155
 FUPDATE, 2-156
 FWRITE, 2-158
 FWRITEDIR, 2-164
 FWRITELABEL, 2-166
 GENMESSAGE, 2-167
 GETDSEG, 2-171
 GETINFO, 2-173
 GETJCW, 2-175
 GETLOCRIN, 2-176
 GETORIGIN, 2-177
 GETPRIORITY, 2-178
 GETPRIVMODE, 2-180
 GETPROCID, 2-181
 GETPROCINFO, 2-182
 GETUSERMODE, 2-184
 INITUSLF, 2-185
 IODONTWAIT, 2-186
 IOWAIT, 2-188, 4-35
 JOBINFO, 2-190, 5-14
 KILL, 2-194
 LOADPROC, 2-195
 LOCKGLORIN, 2-196
 LOCKLOCRIN, 2-198
 LOCRINOWNER, 2-200
 LOGININFO, 2-201
 LOGSTATUS, 2-204

MAIL, 2-206
 MYCOMMAND, 2-208
 OPENLOG, 2-211
 PAUSE, 2-213
 PRINT, 2-214
 PRINTFILEINFO, 2-216
 PRINTOP, 2-217
 PRINTOPREPLY, 2-218
 PROCINFO, 2-220
 PROCTIME, 2-223
 PTAPE, 2-224
 PUTJCW, 2-226
 QUIT, 2-228
 QUITPROG, 2-229
 READ, 2-230
 READX, 2-232
 RECEIVEMAIL, 2-234
 RESETCONTROL, 2-236
 RESETDUMP, 2-237
 SEARCH, 2-238
 SENDMAIL, 2-239
 SETDUMP, 2-241
 SETJCW, 2-242
 STACKDUMP, 2-243
 STARTSESS, 2-245
 SUSPEND, 2-248
 SWITCHDB, 2-250
 TERMINATE, 2-251
 TIMER, 2-252
 UNLOADPROC, 2-253
 UNLOCKGLORIN, 2-254
 UNLOCKLOCRIN, 2-255
 WHO, 2-256
 WRITELOG, 2-260
 XARITRAP, 2-262
 XCONTRAP, 2-264
 XLIBTRAP, 2-266
 XSYSTRAP, 2-267
 ZSIZE, 2-268

Invoking DEBUG Facility, 2-42
 IODONTWAIT Intrinsic, 2-186
 IOWAIT Intrinsic, 2-188, 4-35

J

JCWs (See Job Control Words)
 Job Control Words (JCWs), 5-53
 System-Defined, 5-53
 User-Defined, 5-53
 Value in JCW Table, 2-226

INDEX (Continued)

Job, Identifying With JOBINFO, 5-14
JOBINFO, 2-190
Job/Session
 Access to Related Information, 2-190
 Input Devices, Reading Input From, 5-23
 I/O Devices, Transmitting Programs, 5-23
 List Device, Writing Output To, 5-25
Job Temporary File Directory, 4-9

K

KILL Intrinsic, 2-194

L

Languages, Calling Intrinsics From, 1-4
Library Trap, 5-42
Library Procedures
 Dynamic Loading, 2-195, 5-3
 Dynamic Loading/Unloading, 5-2
 Dynamic Unloading, 2-253, 5-3
Linear Subqueue, 3-6
Line Printer, B-1
Line Printer Carriage-Control Codes, B-2
LOADER Error/Warning Messages, A-9
Loading, Dynamic, 5-3
Load Library Procedure Dynamically, 2-195
LOADPROC Intrinsic, 2-195
Local RINs
 Acquiring, 2-176
 Freeing, 3-45
 Identifying Owners, 3-44
 Interprocess Level, 3-43
 Locking, 2-198
 Locking/Unlocking, 3-43
 Unlocking, 2-255
LOCKGLORIN Intrinsic, 2-196
Locking
 Files, Dynamically, 2-104
 Global RINs, 2-196, 3-39
 Local RINs, 2-198, 3-43
LOCKLOCRIN Intrinsic, 2-198
LOCRIOWNER Intrinsic, 2-200
Logging Facility
 Close Access To, 2-24
 Provide Access To, 2-211

Logging File,
 Information About, 2-201
 Uses For, 3-51
Logging, User, 3-45
Logging, User Procedures, 3-50
LOGINFO Intrinsic, 2-201
LOGSTATUS Intrinsic, 2-204

M

Magnetic Tape, B-2
Mail
 Receiving, 2-234, 3-33
 Sending, 2-239, 3-32
Mailbox, 3-31
MAIL Intrinsic, 2-206
Mailbox, Testing Status Of, 2-206, 3-32
MAKECAT Program, 5-57
Marking
 End of Logging Transaction, 2-49
 Start of User Logging Transaction, 2-16
Master Queue, 3-5
Message Catalog, 5-56
Message Facility, MPE, 5-56
Messages
 Intrinsic Error, A-5
 LOADER Error/Warning, A-8
 Operator, A-10, A-11
 Run-Time, A-1, A-7
 System, A-11
 System Failure, A-1
 System Operator, A-1, A-11
 User, A-10
Mnemonics, Definitions, 2-1
Moving
 DB Pointer, 3-5
 Physical Record Pointer, 2-153
 Record From Disc File to Buffer, 2-143
 User Logging Memory Buffer to
 Logging File, 2-106
MPE Commands, Executing
 Programmatically, 2-26, 5-11
MPE Message Facility, 5-56
Multi-Access (MULTI), 4-2
Multiple Resource Identification Number, 1-8
MYCOMMAND Intrinsic, 2-208

INDEX (Continued)

N

New Disc Files, Opening, 4-17
 Non-Sharable Devices
 Accessing, 4-2
 Files On, 4-4
 Non-Privileged Mode, Entering, 3-5

O

Obtaining
 Calendar Date, 5-52
 Current Time, 5-50
 Process Run Time, 5-52
 System Timer Information, 5-50
 Octal Values, 2-56
 Opening
 Files, 4-3
 Files On Devices Other Than Disc, 4-22
 New Disc Files, 4-17
 Old Disc Files, 4-20
 \$STDIN, 4-25
 \$STDLIST, 4-25
 OPENLOG Intrinsic, 2-211
 Operator Console, 5-26
 Operator Messages, A-10, A-11
 Optical Mark Reader, 2-54, B-3
 Optional Capabilities, 1-7
 Create Volumes, 1-9
 Data-Segment Management, 1-8
 Multiple Resource Identification No., 1-8
 Privileged Mode, 1-8
 Process Handling, 1-8
 Programmatic Sessions, 1-8
 User Logging, 1-8
 Volume Set Usage, 1-9
 Optional Parameters, 1-3

P

Parameters
 Definition, 2-3
 User-Defined Command, 2-208
 With GENMESSAGE, 5-59
 Parsing File Designators, 2-130, 4-15
 PAUSE Intrinsic, 2-213
 Performing Control Operations, 2-75

Permanently Privileged Programs, 3-1
 PRINT Intrinsic, 2-214
 PRINTFILEINFO Intrinsic, 2-216
 Printing
 Character Strings, 2-214
 File Information, 2-216
 On System Console, 2-217
 Reply, From System Console, 2-218
 PRINTOP Intrinsic, 2-217
 PRINTOPREPLY Intrinsic, 2-218
 Priority Classes, 3-6
 Privileged Mode
 Capability, 1-8, 3-1
 Entering, 2-180, 3-3
 Procedure Declarations, 1-2
 Process Break, Requesting, 5-27
 Processes, 3-23
 Aborting, 2-228, 5-28
 Accumulated CPU Time Of, 2-223
 Activating, 2-7
 Active/Suspended Process Substates, 3-24
 Changing Priority Of, 2-178
 Creating, 2-27
 Creating/Activating, 3-24
 Deleting, 3-29
 Determining Source of Activation, 3-35
 Organization Of User, 3-23
 Rescheduling, 3-34
 Scheduling, 3-5
 Suspending, 2-248, 3-29
 Terminating, 2-251, 5-27
 Process Handling Capability, 1-8, 3-22
 Process Identification Number (PIN), 3-23
 Process Priority
 Changing, 2-178
 Determining, 3-36
 Process Run Time, Obtaining, 5-52
 Process State, Determining, 3-36
 PROCINFO Intrinsic, 2-220
 PROCTIME Intrinsic, 2-223
 Programs
 Aborting, 5-28
 Permanently Privileged, 3-1
 Temporarily Privileged, 3-2
 Programmatic Execution Of Commands, 5-11
 Provide Data on Open User Logging File, 2-204
 Providing Control Operations to Devices, 2-75
 PTAPE Intrinsic, 2-224
 PUTJCW Intrinsic, 2-226

INDEX (Continued)

Q

QUIT Intrinsic, 2-228
QUITPROG Intrinsic, 2-229

R

READ Intrinsic, 2-230
Reading
 Disc File to User Data Stack, 2-139
 Input From Job Input Device, 5-23
 Input From Session Input Device, 5-23
 Logical Record, 2-135
 Logical Record Backward, 2-137
 User File Label, 2-141
Reading ASCII String
 From \$STDIN to Array, 2-230
 From \$STDINX to Array, 2-232
READX Intrinsic, 2-232
Receive Mail From Other Process, 2-234
RECEIVEMAIL Intrinsic, 2-234
Receiving Mail, 3-33
Release Extra Data Segment, 2-144
Releasing Global RINs, 3-39
Renaming File, 2-148
Requesting
 File Information, 2-88
 File Input/Output Error Data, 2-58
 PIN of Father Process, 2-53
 PIN of Son Process, 2-181
 Process Break, 5-27
 Status of Father/Son Process, 2-182
Required Capabilities, 2-3
Required Parameters, 1-3
Rescheduling Processes, 2-178, 3-34
RESETCONTROL Intrinsic, 2-236
RESETDUMP Intrinsic, 2-237
Reset, Terminal to Accept
 CONTROL-Y Signal, 2-236
Resource Identification Numbers
 (See RINs)
Resource Management, 3-37
Resources, 3-37
Retrieve
 Info String From :RUN Command, 2-173
 Parm Value From CREATEPROCESS
 Intrinsic, 2-173

Returns

Actual System-Timer Time, 2-23
Calendar Date, 2-19
Current Date/Time Information, 2-40
FCHECK Error Number Message, 2-84
File Label Data From Disc File, 2-101
From User Interrupt Procedure, 2-98
Process CPU Time, 2-223
System Timer, 2-252
To Non-Privileged Mode, 2-184
To Privileged Mode, 2-180
User Information, 2-256
User Logging File Information, 2-204
Value of System-Defined JCW, 2-175
RINs, 3-37
 Acquiring Local, 3-43
 Freeing Local, 2-145, 3-45
 Identifying Local Owners, 3-44
 Interprocess Local Level, 3-43
 Locking/Unlocking Local, 3-43
Run-Time Messages, A-1

S

Scheduling Processes, 3-5
Searching
 Arrays, 5-4
 Arrays for Specified Entry, 2-238
 Job Control Word Table, 2-96
SEARCH Intrinsic, 2-238
Send Mail to Other Process, 2-239
Sending Mail, 3-32
SENDMAIL Intrinsic, 2-239
Sessions
 Aborting, 2-5
 Identifying with JOBINFO, 5-14
 In BREAK Mode, 2-20
Set
 Bits in System JCW, 2-242
 Logical Record Pointer, 2-133
 System JCW Bits, 2-242
SETDUMP Intrinsic, 2-241
SETJCW Intrinsic, 2-242
Son Process
 Deleting, 2-194
 Determining, 3-36
Special Considerations, 2-3
SPL, Calling Intrinsics From, 1-1
Split-Stack Operations, 2-4
STACKDUMP Intrinsic, 2-243

INDEX (Continued)

Stack Sizes, Changing, 5-30
Standard Traps, 5-39
STARTSESS Intrinsic, 2-245
Subqueue Priority Classes, 3-6
Suspending
 Calling Process, 2-213, 5-26
 Processes, 2-248, 3-29
SUSPEND Intrinsic, 2-248
Suspended Process Substates, 3-24
SWITCHDB Intrinsic, 2-250
Syntax Description, 2-1
System Console
 Messages, A-1
 Requesting a Reply From, 5-26
 Writing Output To, 5-26
System Failure Messages, A-1
System Messages, A-11
System Operator Messages, A-1, A-10
System Timer, Obtaining Data From,
 2-23, 2-252, 5-50
System Trap, 5-44

T

Temporarily Privileged Programs, 3-2
Terminal Carriage-Control Codes, B-2
TERMINATE Intrinsic, 2-251
Terminating a Process, 2-251, 5-27
Testing Mailbox Status, 2-194, 3-32
Time and Date Intrinsics, 5-49
Time Information, Formatting,
 2-109, 2-110, 5-52
Time Of Day, 2-109, 2-110, 5-50
TIMER Intrinsic, 2-252
Transferring Data
 From Extra Data Segment to Stack, 3-21
 From Stack to Extra Data Segment, 3-21
Translating Characters With
 CTRANSLATE Intrinsic, 2-36, 5-22
Transmitting Program Input/Output
 From Job/Session Devices, 5-23
Traps, 1-6
 Arithmetic, 5-38
 Commercial Instruction, 5-40
 CONTROL-Y, 5-46
 Disabling/Enabling, 5-37
 Extended Precision Floating Point, 5-39
 Library, 5-42
 Standard, 5-39
 System, 5-44

U

Unloading, Dynamic, 5-3
Unloading Library Procedure, 2-253
UNLOADPROC Intrinsic, 2-253
Unlocking
 Files Dynamically, 2-155
 Global RINs, 2-254, 3-39
 Local RINs, 2-255, 3-43
UNLOCKGLORIN Intrinsic, 2-254
UNLOCKLOCRIN Intrinsic, 2-255
User Access Mode, Determining, 5-12
User Attributes, Determining, 5-12
User-Defined Job Control Words, 5-55
User Logging, 3-45
 Capability, 1-8
 Flush Memory Buffer Of, 2-106
 How It Works, 3-46
 Mark Beginning of Transaction, 2-16
 Mark End of Transaction, 2-49
 Procedures, 3-50
User Messages, A-10
User Processes, Organization Of, 3-23
Uses For Log File, 3-51
Using Files, 4-5
USL Files
 Directory Space, Adjusting, 2-9
 Inactive Entries, Deleting, 2-21
 Initializing To Empty State, 2-185
 Length Of, Changing, 2-51
Utility Functions, MPE, 5-1

V

Validating File Designators, 2-130, 4-15
Volume Set Usage Capability, 1-9

W

WHO Intrinsic, 2-256
WRITELOG Intrinsic, 2-260

INDEX (Continued)

Writing

- File System Error-Check Procedure, 4-33
- Logical Record In Disc File, 2-156
- Logical Record/User Stack
 - To Disc File, 2-164
- Logical Record to File, 2-158
- Record to Logging File, 2-260
- User File Label, 2-166

Writing Output To

- Job/Session List Device, 5-25
- System Console, 5-26
- System Console/Request A Reply, 5-26

XYZ

XARITRAP Intrinsic, 2-262
XCONTRAP Intrinsic, 2-264

XLIBTRAP Intrinsic, 2-266
XSYSTRAP Intrinsic, 2-267
ZSIZE Intrinsic, 2-268
Z To DB Area Size, Changing, 2-268, 5-37

\$

\$STDIN

- Assigning, 2-32
- Opening, 4-25
- Using FREAD/FWRITE With, 4-23

\$STDLIST

- Assigning, 2-32
- Opening, 4-25
- Using FREAD/FWRITE With, 4-23

READER COMMENT SHEET

MPE V Intrinsic Reference Manual

32033-90007

February 1986

We welcome your evaluation of this manual. It is one of several that serve as a reference source for HP 3000 Computer Systems. Your comments and suggestions help us to improve our publications and will be reviewed by appropriate technical personnel. HP may make any use of the submitted suggestions and comments without obligation.

Is this manual technically accurate? Yes ☐ No ☐ (If no, explain under Comments, below.)

Are the concepts and wording easy to understand? Yes ☐ No ☐ (If no, explain under Comments, below.)

Is the format of this manual convenient in size, arrangement and readability? Yes ☐ No ☐ (If no, explain or suggest improvements under Comments, below.)

Comments:

We appreciate your comments and suggestions. This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

Date: _____

FROM:

Name _____

Company _____

Address _____

FOLD

FOLD



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 781 CUPERTINO, CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Documentation Manager/47LS
Hewlett-Packard Company
Computer Systems Division
19447 Pruneridge Avenue
Cupertino, California 95014

FOLD

FOLD

Part No. 32033-90007
Printed in U.S.A. 02/86
E0286

