# HP 3000

# MULTIPROGRAMMING EXECUTIVE OPERATING SYSTEM

HEWLETT **hp** PACKARD

3000 computer systems

# HP 3000

# MULTIPROGRAMMING EXECUTIVE OPERATING SYSTEM

# List of Effective Pages

| Pages | Effective Date |
|---|---|
| Title . . . . . . . . . . . . . . . | Sep. 1973 |
| Copyright. . . . . . . . . . . . . | Sep. 1973 |
| iii to xvii . . . . . . . . . . . . | Sep. 1973 |
| Part 1 Divider . . . . . . . . . . | Sep. 1973 |
| 1-1 to 1-10 . . . . . . . . . . . | Sep. 1973 |
| 2-1 to 2-28 . . . . . . . . . . . | Sep. 1973 |
| Part 2 Divider . . . . . . . . . . | Sep. 1973 |
| 3-1 to 3-26 . . . . . . . . . . . | Sep. 1973 |
| 4-1 to 4-27 . . . . . . . . . . . | Sep. 1973 |
| 5-1 to 5-53 . . . . . . . . . . . | Sep. 1973 |
| 6-1 to 6-52 . . . . . . . . . . . | Sep. 1973 |
| 7-1 to 7-30 . . . . . . . . . . . | Sep. 1973 |
| 8-1 to 8-73 . . . . . . . . . . . | Sep. 1973 |
| 9-1 to 9-11 . . . . . . . . . . . | Sep. 1973 |
| 10-1 to 10-33 . . . . . . . . . . | Sep. 1973 |
| Part 3 Divider . . . . . . . . . . | Sep. 1973 |
| 11-1 to 11-28 . . . . . . . . . . | Sep. 1973 |
| 12-1 to 12-7 . . . . . . . . . . | Sep. 1973 |
| 13-1 to 13-2 . . . . . . . . . . | Sep. 1973 |
| 14-1 to 14-9 . . . . . . . . . . | Sep. 1973 |
| Part 4 Divider . . . . . . . . . . | Sep. 1973 |
| A-1 to A-5 . . . . . . . . . . . | Sep. 1973 |
| B-1 to B-6 . . . . . . . . . . . | Sep. 1973 |
| C-1 to C-83 . . . . . . . . . . . | Sep. 1973 |
| D-1 to D-2 . . . . . . . . . . . | Sep. 1973 |
| E-1 to E-2 . . . . . . . . . . . | Sep. 1973 |
| F-1 to F-2 . . . . . . . . . . . | Sep. 1973 |
| G-1 . . . . . . . . . . . . . . . | Sep. 1973 |

# *Printing History*

# *Preface*

This manual explains how to compile and execute programs, manipulate files, request utility functions, and perform other standard programming operations under the Multiprogramming Executive Operating System (MPE/3000, Release B) for the HP 3000 Computer. In addition, it shows how to use the various optional capabilities of the system. Subjects are arranged in functional order, beginning with the simplest, most commonly-used operations and proceeding toward more difficult, sophisticated functions. Once the user has familiarized himself with this book, he can use the extensive index in the back to quickly and easily find any information he needs.

This manual is a reference book rather than a tutorial text for new programmers. The reader should understand the fundamental techniques of programming and the operating principles of the HP 3000 Computer. He should also examine the following manual for an overview of the inter-relationships between the main hardware and software features offered:

| Title | Part No. |
|---|---|
| *HP 3000 Software General Information Manual* | *03000-90001* |

The user may also want to read the following manuals for additional information, depending upon the applications he plans:

| Title | Part No. |
|---|---|
| *HP 3000 Systems Programming Language* | *03000-90002* |
| *HP 3000 Systems Programming Language Textbook* | *03000-90003* |
| *HP 3000 Multiprogramming Executive Console Operator's Guide* | *03000-90006* |
| *HP 3000 Multiprogramming Executive System Manager/Supervisor Capabilities* | *03000-90038* |

# *Guide for Readers*

The following guide shows the sections of the manual that should be read to perform particular tasks:

| To: | Read Section: | Prerequisite Sections: |
|---|---|---|
| Initiate and terminate batch jobs or interactive sessions. | III | None |
| Compile, prepare, and execute programs. | IV | III |
| Call MPE/3000 subsystems. | IV | III |
| Create and manage files through MPE/3000 commands. | V | III, IV |
| Read, write, and update files. | VI | III, IV |
| Manipulate files and obtain file-access and error information. | VI | III, IV |
| Alter user subprogram libraries. | VII | III, V |
| Maintain external library procedures. | VII | III, V |
| List system information. | VIII | III |
| Request binary/ASCII code conversion. | VIII | III |
| Return system information to the user's program. | VIII | III |
| Enable or disable traps. | VIII | III |
| Manage interprocess communication through the job control word. | VIII | III |
| Reserve and allocate user resources. | IX | III |

| To: | Read Section: | Prerequisite Sections: |
|---|---|---|
| Diagnose errors. | X | None |
| Handle processes. (Optional Capability.) | XI | II, III, IV |
| Manage Data Segments. (Optional Capability.) | XII | II, III, IV |
| Manage multiple RIN's. (Optional Capability.) | XIII | II, III, IV, IX |
| Execute programs in privileged mode. (Optional Capability.) | XIV | II, III, IV |

# *CONTENTS*

## FIGURES

# PART 1
## System Overview

# SECTION I
# Introduction to MPE/3000

The Multiprogramming Executive Operating System (MPE/3000) is a general-purpose, disc-based software system that supervises the processing of user programs submitted to the HP 3000 Computer. MPE/3000 relieves the user from many program control, input/output, and other housekeeping responsibilities by monitoring and controlling the input, compilation, run preparation, loading, execution and output of user programs. MPE/3000 also controls the order in which programs are executed, and allocates the hardware and software resources they require.

## FEATURES

MPE/3000 offers the user many important features; some of these are found elsewhere only in medium to large-scale computers.

## Multiprogramming

Through multiprogramming (interleaved processing), MPE/3000 allows numerous users to execute many different programs concurrently. The number of programs that can be processed concurrently depends on such factors as the hardware configuration, program operating modes (batch or interactive), and applications involved. Each programmer, however, uses the computer as if it were his own private machine — in other words, he need not depend on, nor even be aware of, others using the machine.

## General-Purpose Versatility

MPE/3000 allows users to run batch and interactive programs concurrently.

**BATCH PROCESSING.** Batch processing lets programmers submit to the computer, as a single unit, commands that request various MPE/3000 operations such as program compilation and execution, file manipulation, or utility functions. Such a unit is called a *job*. Jobs contain all instructions to MPE/3000 and references to programs and data required for their execution; once a job is running, no further information is needed from the programmer. (Frequently, new programs and data are submitted as part of the job.)

Jobs are input through batch input devices such as card readers. In fact, several jobs can be submitted from multiple batch input devices concurrently. MPE/3000 schedules each job according to its priority. When a job enters execution, the commands within it are sequentially executed on a multiprogramming basis. MPE/3000 generates the job output on a local device such as a line printer, tape unit, or disc unit, or on a local or remote terminal. When one job is temporarily suspended, perhaps to await input of data, another (if available) immediately enters execution. Thus, when many jobs are active in the system, continuous processing and high throughput can be maintained.

INTERACTIVE PROCESSING. In interactive processing, the programmer interacts conversationally with the computer, receiving immediate responses to his input. Many users at different remote or local terminals can program on-line in this fashion. This type of interaction, called a *session*, can be used for program development, information retrieval, computer-assisted education, and many more applications where the user at a remote terminal must access the system directly.

MPE/3000 can continue to execute batch jobs at the same time it handles sessions. The basic difference between a session and a batch job is that a session is interactive, but a job is not. Thus, during a session, the user maintains a dialogue with the system to control input and monitor output; in a batch job, however, the command stream is entered into the system without a user/system dialogue.

## Choice of Programming Languages

Under MPE/3000, the user can submit programs written in the following languages:

- FORTRAN/3000
- BASIC/3000
- COBOL/3000
- SPL/3000 (Systems Programming Language)

In addition, MPE/3000 supports subsystems for editing program files, performing statistical operations, aiding in debugging programs, running on-line diagnostic programs, and various other applications.

Each language translator and subsystem is accessed by a unique MPE/3000 command. The programmer need learn only one set of conventions for using these programs, because they all use the same general command formats, special characters, and error-diagnostic methods. Command coding is based upon the American Standard Code for Information Interchange (ASCII) Character Set, shown in Appendix A.

## Operating Simplicity

MPE/3000 is easy to initialize, operate, monitor, and shut down. Its operation is overseen by a user with the MPE/3000 system manager capability, who assigns programming capabilities of various levels to each user. *Standard capabilities* allow the typical "general applications"

programmer to interact with the computer through a batch input device or a terminal. If this programmer is planning only to compile and execute a batch job, or to run an on-line interactive program, he may need to know only a few simple MPE/3000 commands. As his needs become more extensive, however, so must his knowledge of MPE/3000. A user planning more complex operations can be assigned various *optional capabilities*. These allow him to access more sophisticated system resources for such tasks as executing privileged instructions. Sets of capabilities are also used to protect the system and its users by limiting access to special system capabilities only to those who understand their correct use. Capability sets greatly simplify use of the system from the stand-point of each individual user — they define the extent to which he must understand and interrelate with MPE/3000, and permit a user to ignore aspects of MPE/3000 that do not apply to him.

## Input/Output Conveniences

Because MPE/3000 treats all input/output devices as files (or groups of files in the case of mass storage devices), the programmer may access these devices by file names rather than by device types or logical unit numbers. The file names used in programs are independent of the devices used for file input and output, and need only be associated with these devices at the time the programs are run. This means that the user can write programs without immediate concern for the physical source of input or destination of output. This independence also means that programs can be run in either batch or interactive mode without changing the names of the files they reference.

Files on disc can be structured for either direct or sequential access, on an exclusive or shared basis. Direct-access files contain fixed-length records; sequential files, however, can contain either fixed- or variable-length records. For files on disc, storage space is automatically allocated as it is needed. MPE/3000 permits simultaneous access of sharable disc files by many programmers.

## Processing Efficiency

MPE/3000 offers the user increased throughput by taking best advantage of the HP 3000 Computer architecture and the rapid speed of the central processor. This, in turn, permits the rapid changes between user environments that make multiprogramming in batch and interactive modes possible.

RE-ENTRANT CODE AND PRIVATE DATA. Within MPE/3000, many user and system functions can be active simultaneously without mutual interference. This is because the hardware provides protection of programs and guarantees the privacy of user data areas. MPE/3000 keeps code and data logically separate by organizing them into re-entrant code segments (which can be shared among users but not altered) and data segments (which cannot be shared but which can be altered by the creating user). *Code-sharing* means that only one copy of each program, accessible to many users concurrently, need exist in memory at any time. *Code segmentation* allows code to be moved from disc into main memory only when needed and dynamically relocated in main memory to accommodate other programs and routines. *Code re-entrancy* means that when a program is interrupted during execution of a code segment, and another program uses that same segment, the segment is completely protected from modification and will be returned, intact, to the previous program.

**STACK ARCHITECTURE.** Many powerful operating system features are made possible by the computer's use of stacks (linear storage areas for data) where the last item stored in is always the first item taken out. Some of these features are

- Ease of compilation and parameter passing

- Fast execution

- Highly-efficient subroutine linkage

- Minimum overhead

- Dynamic allocation of temporary storage

- Rapid interruption and restoration of user environments

- Code compression

- Recursion (where a procedure calls itself)

**MICROPROGRAMMING AND THE CENTRAL PROCESSOR.** Additional economy has been provided by microprogramming many system operations normally provided by software. These operations are requested by machine instructions that each, in turn, execute many micro-instructions built into the central processor hardware. Microprogramming eliminates the repetitive coding and main memory requirements otherwise needed for recurring operations (such as moving character strings from one location to another, or scanning strings for a particular character), thereby placing the operating burden on the highly-efficient micro-processor rather than on MPE/3000 software.

**VIRTUAL MEMORY.** The MPE/3000 virtual memory offers users a total memory space that far exceeds the maximum main-memory size of 131,072 bytes. Virtual memory consists of both main memory and an extensive, flexible storage area on disc. User programs and information in the disc area are subdivided into units (segments) of code or data that are dynamically moved, through overlays, into main-memory (core) for execution.

**HIGH DATA THROUGHPUT.** High throughput of data is facilitated by high-speed channels, double-buffering, many input/output devices, and input/output program execution (through an input/output processor) in parallel with regular program execution (by the central processor).

**AUTOMATIC SCHEDULING.** MPE/3000 automatically schedules all jobs and sessions according to their priorities. When execution of a running program is interrupted for any reason (such as input/output or an internal interrupt), MPE/3000 passes control to the program of highest priority awaiting execution.

## Accounting Facility

MPE/3000 keeps track of various system resources used by each job/session, group, and account; these resources include permanent file space, central-processor time, and (for sessions) terminal connect time. Limits can be set for the maximum use of these resources at the group or account level. As each job/session is logged off, the resource-use counters are updated. When another job/session attempts to log-on, and the central processor or (for sessions) terminal connect time limits have previously been exceeded, access is refused. When a request is made to save a file, and this action would result in exceeding the permanent file space limit at either the account or group level, the request is denied.

The accounting information for each group and account can be extracted and displayed, showing counts and limits for permanent file space (in disc sectors), central processor time (in seconds), and connect-time (in minutes).

## Logging Facility

MPE/3000 enables a user with System Supervisor Capability to control the recording (on disc) of information about overall system activity. This log can be displayed through user-created analysis routines. The user can specify the information to be recorded on the log file, making his selection from the following areas: job/session and program initiation and termination, file closing, and system start-up and shut-down.

## System-to-System Compatibility

All HP 3000 Computers operate under a single operating system — MPE/3000. This means that programs prepared on one HP 3000 can be run on any other without modification (provided that all devices required by those programs are connected on-line). It also means that users moving from one installation to another need not undergo additional training or read additional documentation to prepare themselves for the new environment.

## Program and File Security

Each user operates in an environment protected from interference by other users. Program protection is provided by the hardware; file security is provided by a software facility based upon a series of lockwords and hierarchical access restrictions that allow the programmer to specify the degree of security desired.

## System Manager and System Supervisor Capabilities

MPE/3000 provides many tools for overall management and control of the system to persons with the *System Manager* and *System Supervisor Capabilities*. These features are discussed in detail in *HP 3000 Multiprogramming Executive System Manager/Supervisor Capabilities (03000-00038)*.

The System Manager Capability allows a user to have final control of overall use of the system by defining the accounts under which users access MPE/3000, and the resource-use limits (if any) that apply to these accounts.

The System Supervisor Capability allows a user to supervise and control the general operation of the system by:

- Creating tapes for backing-up and modifying the system.
- Displaying certain system information.
- Permanently allocating/deallocating programs in virtual memory.
- Exercising greater scheduling control over processes than that allowed other users.
- Managing the system log file.

It is important to bear in mind that the above capabilities are granted to users through the assignment of special *attributes*, and in no way imply formal responsibilities or duties. Thus, a regular programmer may have the System Manager Capability or the System Supervisor Capability, or both capabilities — in addition to several others discussed later. For this reason, it is best to speak of "a user with System Manager Capability" rather than a formal "System Manager."

## SOFTWARE

MPE/3000 is furnished to the customer on a reel of magnetic tape. As part of the operating system software, these major components are provided:

- *System Configurator*, for configuring and making a back-up copy of MPE/3000.

- *System Initiator*, for installing and starting MPE/3000.

- *Command Interpreter*, for handling the commands that allow the user to interact with MPE/3000 through a terminal or batch input device.

- *File Management System*, for providing uniform access to disc files and standard input/output devices, and maintaining file security.

- *Memory Management System*, for dynamically allocating main-memory among contending users.

- *Dispatcher*, for allocating central-processor time among programs in execution.

- *Input/Output System and Drivers*, for scheduling, initializing, monitoring, and completing input/output requests for standard devices (accessed through the File Management System).

- *System Library*, for storage of frequently-used routines sharable among many users.

- *Segmenter Subsystem*, for segmenting and loading programs, and resolving references to external code.

- *Accounting System*, for maintaining and displaying resource usage counts.

- *Logging System*, for maintaining the system log file.

No additional or auxiliary software is required to install, operate, or maintain MPE/3000.

## HARDWARE

The minimum hardware configuration required by MPE/3000 is

- Mainframe and Accessory Equipment Supplied, including HP 3000 Central Processor and 65,536 Bytes of Main Memory (Core), SIO Multiplexer Channel, System Control Desk (with System Console), and Asynchronous Terminal Controller

- Disc

- Magnetic Tape Unit

The disc is used to contain portions of MPE/3000 and user programs and data. The magnetic tape unit is used for cold-loading MPE/3000, loading subsystems (such as compilers), and storing user programs and data. (MPE/3000 is initialized and maintained through the console, which can also be used for input/output of user programs.)

The following optional hardware can be added to the system:

- Main Memory (for a total of up to 131,072 bytes)
- Discs (Fixed-Head or Moving-Head)
- Magnetic Tape Units
- Card Readers
- Line Printers
- Card Punches
- On-Line Terminals
- Paper Tape Readers
- Paper Tape Punches
- Selector Channels

With this optional equipment, many hardware configurations are possible. For example, a small system used for batch processing might appear as shown in Figure 1-1. In this system, the card reader is used for input and the line printer for output, while the disc is used for storing system and user programs and data. The magnetic tape unit is used for cold-loading MPE/3000 and for storing user programs and data. The console is used to initialize and maintain MPE/3000.
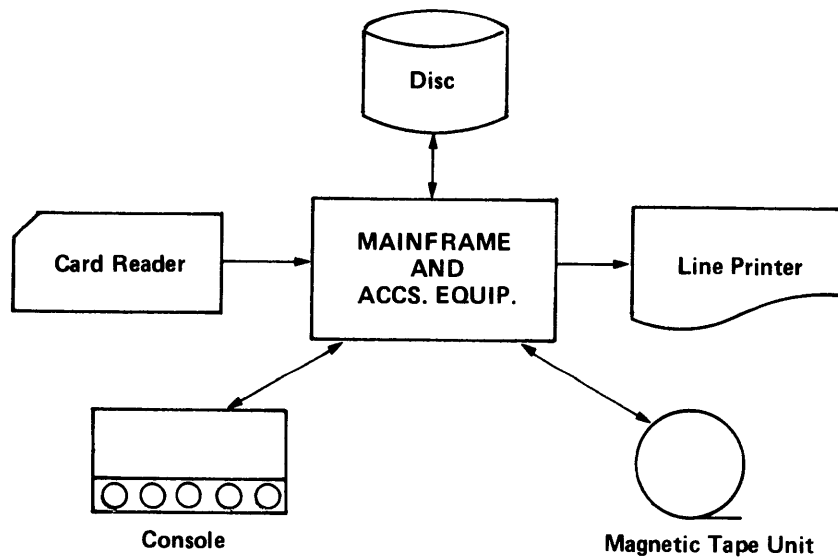


Figure 1-1. Small Batch System

A small interactive system might be configured as shown in Figure 1-2. In this system, up to seven terminals are used for input and output.
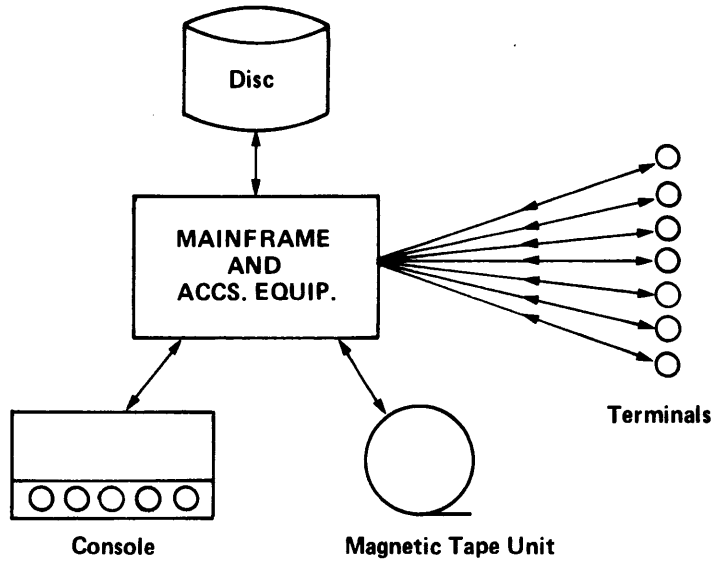


Figure 1-2. Small Interactive System

A combined batch and interactive system is illustrated in Figure 1-3. Here, batch and terminal operations can occur separately or concurrently.
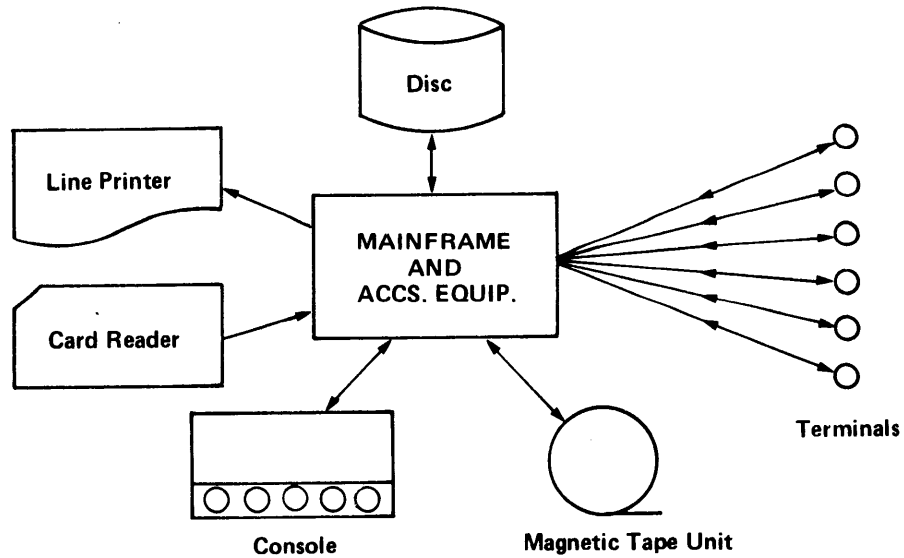


Figure 1-3. Combined Batch and Interactive System

A large processing system is shown in Figure 1-4. This system incorporates a large central memory, fixed-head and moving-head discs, many input/output devices for multiple-job batching, and eight terminals for remote processing.
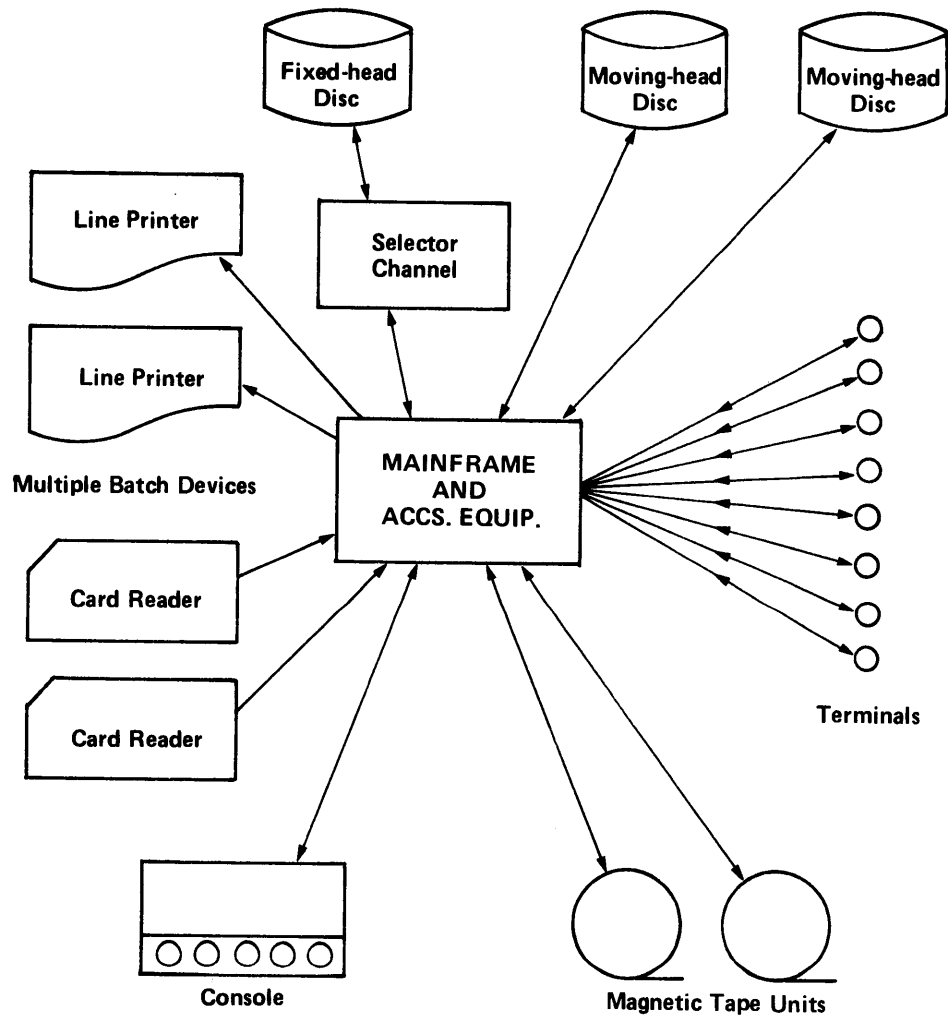


Figure 1-4. Large Processing System

# SECTION II
# How MPE/3000 Operates

In supervising the compilation and execution of user programs, MPE/3000 performs the following tasks:

- Controls all input/output.

- Allocates memory to user programs and the system routines required for their support.

- Schedules batch programs for access to the central processor.

- Allocates central processor time to programs.

- Swaps portions of programs to and from main memory as required during execution.

NOTE: *For programmers using the standard capabilities of MPE/3000 to compile and run general application programs, MPE/3000 automatically performs these tasks. Such users normally are not concerned with the internal aspects of MPE/3000 discussed in this section, and can proceed directly to Part 2 for directions on how to log-on to MPE/3000, and how to compile and run programs. But as a programmer's needs become more complex, perhaps involving calls to higher-level MPE/3000 system procedures (intrinsics), he may want to scan this section for a general system orientation, or refer to parts of it to clarify terms and concepts. Those programmers using MPE/3000 on very sophisticated levels (involving the optional capabilities discussed in Part 3), will find the information in this section essential; they should read it thoroughly.*

## MPE/3000 RESIDENCE

MPE/3000 resides in three general areas:

1. Those routines most frequently used reside in *main memory*, all of which is part of virtual memory. (Routines not residing in main memory are moved there only when required. This ensures that as much main memory as possible remains available for the execution of user programs.)

2. Frequently-used routines reside on *portions of disc within virtual memory;* these disc areas are treated as logical extensions of main memory.

3. All remaining routines (including a permanent copy of each compiler) are stored in *areas of disc outside of virtual memory;* these areas are treated as conventional secondary storage.


## PROCESSES

In MPE/3000, programs are run on the basis of *processes* created and handled by the system. A process is not a program itself, but the unique execution of a program by a particular user at a particular time. Therefore, if the same program is run by several users, or more than once by the same user, it is used in several distinct processes.

The process is the basic executable entity in MPE/3000. A process consists of a process control block that defines and monitors the state of the process, a dynamically-changing set of code segments, and a data area (stack) upon which these segments operate. (Thus, while a *program* consists of instructions (not yet executable) and data in a file, a *process* is an executing program with a data stack assigned.) The code segments used by a process can be shared with other processes, but its data stack is private (Figure 2-1). For example, each user working on-line through the BASIC language is running his program under a separate process; all use the same code (the only copy of the BASIC interpreter in the system), but each has his own stack. (The interrelationship of the process control block, code segments, and data stack is discussed in detail under "PCB/Code Segment/Stack Interaction.")

Processes, and the elements that comprise them, are invisible to the programmer accessing MPE/3000 through standard capabilities; in other words, this programmer has no control over processes or their structure. For this user, MPE/3000 automatically creates, handles and deletes all processes. The user with certain optional capabilities, however, can create and manipulate processes directly.

The entire MPE/3000 system is a collection of processes that are either active or dormant. These processes are logically organized in a tree structure (see Figure 2-2); each process has only one immediate ancestor (father) but could have several immediate descendants (sons). The root process that controls all other processes in MPE/3000 (Figure 2-2) is the only process that has no ancestor. It is the first process established, and is created during system initialization. During system configuration, the root process uses the data specified to create various system processes, including the input/output and user controller processes; the user controller process creates its own descendent processes. (A *system process* is a process that executes on behalf of MPE/3000.)
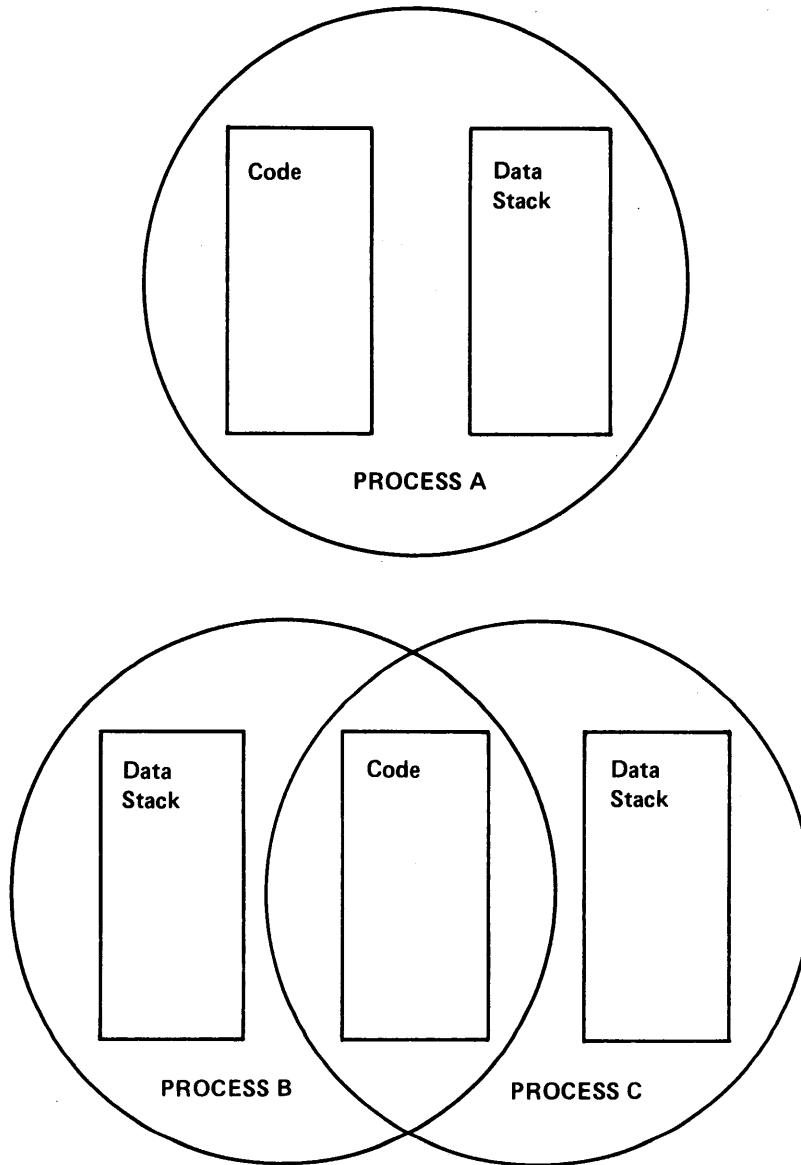
Figure 2-1. Code-Sharing and Data Privacy

Figure 2-2. Process Organization

The user controller process (UCOP) is the ancestor of all existing user processes. (A *user process* is a process that runs on behalf of the user.) When the user first accesses MPE/3000, the UCOP creates a *main process* that oversees the execution of his program. If the user is running a batch program (called a *job*), a job main process (JMP) is created. If he is executing an interactive program (*session*), a session main process (SMP) is created.

All JMP's and SMP's use system code, the MPE/3000 Command Interpreter, interfacing directly with the user. They may create descendants that use system code, such as the BASIC/3000 Interpreter or FORTRAN/3000 or SPL/3000 compilers when translating source programs; when executing user programs, of course, the descendant processes use the user's code.

## PROCEDURES

In MPE/3000 most individual programming operations are handled by unique sets of code known as *procedures*. (Others are handled by special programs or interrupt routines.) In SPL/3000, the language in which MPE/3000 is coded, a procedure is defined by a procedure declaration consisting of

- A procedure head, containing the procedure name and type, parameter definitions, and other information about the procedure

- A procedure body, containing executable statements and data declarations local to this procedure

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

In SPL/3000, each procedure is executed by a corresponding *procedure call*. When a procedure call is encountered, control is transferred to the procedure, which runs until an exit is encountered. Control is then returned to the statement following the call for that procedure.

While a procedure in SPL/3000 is similar to a subroutine in many respects, there are major differences between the two:

- When referencing a logical set of code, a subroutine call is significantly faster and more efficient than a procedure call, while a procedure call has more powerful (and general) applications.

- A subroutine declaration may be used within a procedure declaration, but a procedure declaration may not appear in a subroutine.

- A procedure allows dynamic storage allocation, permitting local variables.

- A subroutine can contain non-local references to variables declared within the scope of the procedure in which it occurs.

- A subroutine can be invoked only within the procedure of which it is a part.

Unlike procedures, subroutines are not suitable for solving complex problems that require explicit stack manipulation or many temporary storage locations. They *are* useful, however, for various utility and arithmetic-function routines.

In addition to the procedures provided by MPE/3000, SPL/3000 allows the user to write procedures to suit his own purposes. To distinguish MPE/3000 system procedures (which are always available to the user, directly or indirectly) from any other procedures, the term *intrinsics* is applied to MPE/3000 procedures. Similarly, the term *intrinsic calls* is used to denote the procedure calls that reference MPE/3000 procedures. From this point on, these terms will carry these respective meanings throughout this manual.

Each large functional unit of MPE/3000, such as the Command Interpreter, File Management System, or Input/Output System, consists of many intrinsics. The creation, handling, and deletion of processes is performed exclusively through intrinsics not directly accessible by users with the standard MPE/3000 capabilities. (However, these intrinsics can be accessed directly by users having certain optional capabilities described in Part 3.)

## SYSTEM LIBRARY

The MPE/3000 system library is a collection of frequently-used routines that can be readily shared among many users. A library code segment consists of one or more procedures. Some of these segments reside permanently in virtual memory, while others are stored elsewhere on disc and called into virtual memory as needed.

## USER INTERFACE

In requesting system functions, the programmer never interacts directly with the computer hardware. Instead, he interacts with a dual-level software interface that accepts *commands* (for general functions external to his program) and *intrinsic calls* (for specific functions within his program). In general, the commands and intrinsic calls themselves access uncallable intrinsics that reference machine instructions. These, in turn, execute micro-instructions from the microcode hardwired into read-only memory (Figure 2-3). It is this microcode only that operates directly on the computer hardware.

### Commands

Commands are used to initiate and terminate jobs, re-specify file characteristics, compile and execute programs, call various utility subsystems, and perform other broad functions external to user programs. Commands are entered through any standard input device, typically the card reader (for jobs) or the terminal (for sessions). When the input device is connected on-line to MPE/3000, a main process is automatically initiated that uses the MPE/3000 Command Interpreter program to read and translate the commands. The first command entered must always be one that initiates a job or session.
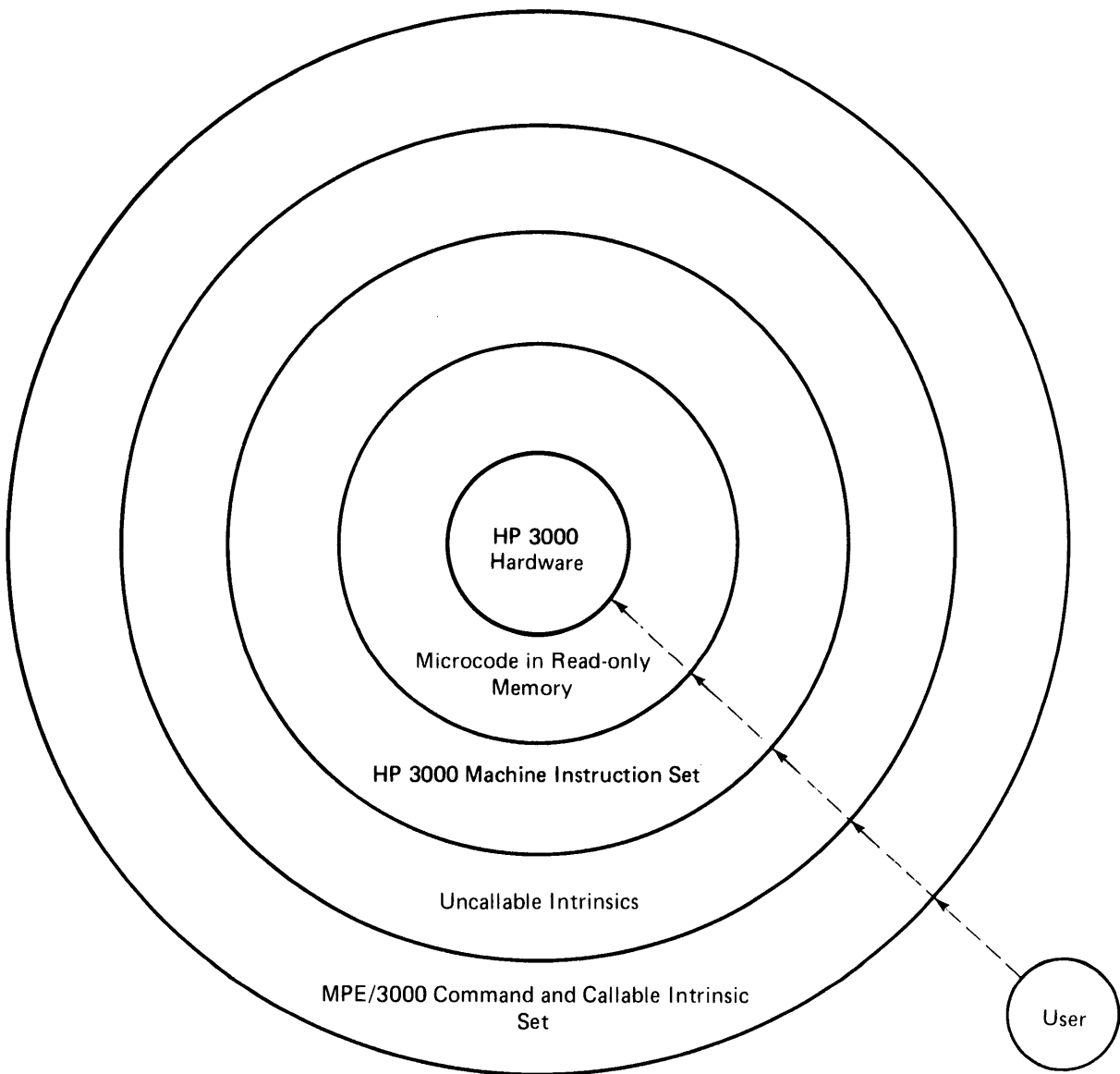
Figure 2-3. User/Software/Hardware Interface

Each command invokes an intrinsic that implements the function desired. (Such intrinsics are called *command executors.*) When a command is entered, the Command Interpreter checks it against the names in the MPE/3000 command dictionary. If the command is valid, the Command Interpreter passes the command parameter list image to the appropriate executor for execution. When the action requested by the command is satisfactorily completed, control returns to the Command Interpreter.

## Intrinsic Calls

Intrinsic calls implement MPE/3000 functions requested *programmatically* (within a user's program), such as reading, writing on, and updating files; skipping forward and backward on files, or returning system table information to the user's program. In an SPL/3000 program, the user writes the intrinsic calls explicitly, but in FORTRAN, COBOL, or BASIC programs, in most general applications, the compiler generates any necessary intrinsic calls automatically — they are invisible to the user. (Provisions are also available in FORTRAN, COBOL, and BASIC that allow the user himself to call intrinsics, at his option.) All MPE/3000 intrinsics are treated as external procedures by user programs. External linkages from user programs to intrinsics are satisfied when the user programs are segmented and allocated residence in virtual memory.

The set of intrinsics available for a user depend upon the capabilities assigned to him by the user who is managing the account under which he accesses MPE/3000. For example, only the user with the *Process-Handling Optional Capability* can directly access intrinsics that create and manipulate processes. The set of intrinsics available for a user defines the computer for that user. The user calls *callable* intrinsics appropriate for a particular application, and these intrinsics check the validity of the requests. Callable intrinsics, in general, invoke *uncallable* intrinsics to actually perform the necessary operations. (Uncallable intrinsics are those not available to standard users because their mis-use can be hazardous to the system; they can be accessed, however, by users with a special MPE/3000 optional capability.)

Certain programs that may be executed by all users sometimes require special capabilities. In this case, the program (not the user) is assigned these capabilities by the user who creates the program, as discussed later.

## INPUT/OUTPUT

The MPE/3000 Input/Output System schedules, initiates, and completes all input/output requests for all HP-supported devices connected to the system. Normally, the input/output system remains invisible to the user, since he accesses it indirectly through the MPE/3000 File Management System.

The Input/Output System includes an independent controller for each type of hardware device connected to the computer. Each controller is associated with an input/output monitor process. This process services the requests for the controller and calls input/output initiator and completion driver routines as required. Requests to the Input/Output System are issued by the File Management and Memory Management Systems.

## File Management System

The File Management System provides a uniform method of directing input and output of information. It handles various input/output applications, such as the transfer of information to and from user processes, compilers, and data management subsystems.

MPE/3000 treats each set of input or output information as a set of logical records called a *file*. When a file is created, MPE/3000 allocates (or requests the operator to allocate) a device for its storage. Input read from a hardware device such as a card reader is accepted directly from that device. Similarly, output from an executing process is transmitted directly to the required device (such as a line printer).

Information is moved between a file and main memory in *physical records*; a physical record is the basic unit that can be transferred to or from the device on which the file resides. In files on disc or magnetic tape, physical records are organized as *blocks* of logical records; the block size is specified by the user, but input/output and blocking are performed by MPE/3000, freeing the user from the actual record-handling details. On unit record devices, however, the physical record size is determined by the device itself, and logical records are not blocked.

The user references a file by the *file name* assigned to the file when it is created. When he does this for an existing file, MPE/3000 determines the device or disc address where the file is stored and accesses the file for him. Throughout its existence, the file remains device-independent.

The MPE/3000 File Management System allocates devices for the storage of files on the basis of specifications from the user. For example, the user can request a device by generic type name (such as any magnetic tape unit or line printer), or by the logical device number that refers to a specific device. Logical device numbers are related to all devices when MPE/3000 is configured.

## Scheduling

The MPE/3000 Scheduling System ensures that all processes competing for the central processor access it in an orderly manner. All processes are placed in a master scheduling queue in order of their priority. (This queue is actually a linked list of PCB's — process control blocks — for the processes scheduled.) When a process in execution is completed, terminated, or interrupted, the MPE/3000 Dispatcher program searches the master queue for the process of highest priority awaiting execution and transfers control to that process.

Within the master queue, five standard subqueues are used for scheduling processes created by user programs (Figure 2-4). Since MPE/3000 schedules each process independently, not all processes for a job are necessarily entered in the same subqueue.
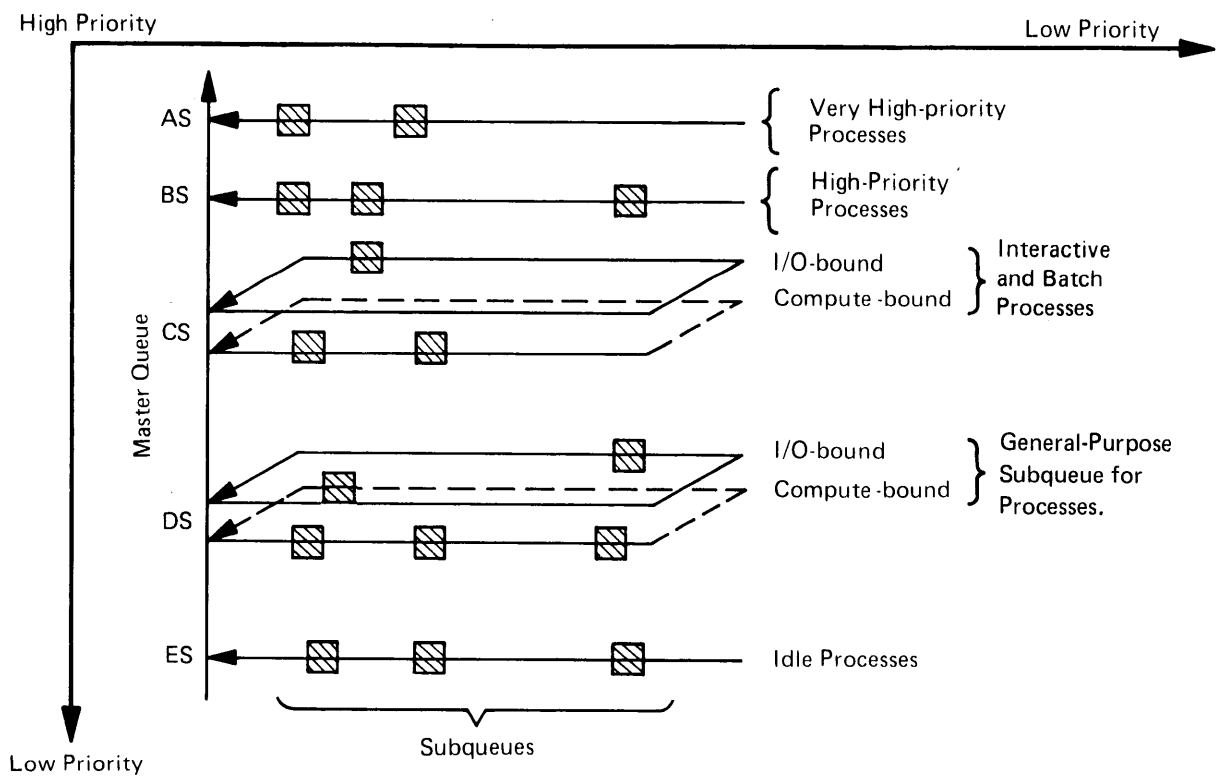
**Figure 2-4. Scheduling Queues**

Three of the standard subqueues are linear in structure; two of these (AS and BS) are high-priority subqueues that are presently available for general purposes, but in subsequent releases of MPE/3000 will be used for processes with new MPE/3000 optional capabilities; the third linear subqueue (ES) is available for idle processes with low priority. In a linear subqueue, the process with highest priority accesses the central processor first and maintains this access until the process either is completed or suspended to await input/output. (The master queue itself is a linear queue.)

The other two standard subqueues are circular subqueues. One (CS) is used for interactive sessions and multiprogramming batch jobs. The other (DS) is available for general use at a lower priority than the CS subqueue. In these subqueues, each process accesses the central processor for an interval (time-slice) of limited duration. At the end of each time-slice, or when the currently-running process enters a waiting state, control is transferred to the next process in the subqueue, continuing in a round-robin fashion. This time-slicing is controlled by a system timer. Each of the two circular subqueues is composed of two sub-subqueues. In both subqueues, the lower-priority sub-subqueue services compute-bound processes while the higher-priority one services input/output-bound processes (those that require excessive time for input/output). When a compute-bound process in one of these subqueues is repeatedly suspended for input/output rather than exceeding its time-slice, the scheduling system moves it into the slightly higher-priority input/output-bound sub-subqueue. This ensures that the process obtains adequate central processor time. When the process is no longer input/output-bound, it is returned to the compute-bound sub-subqueue.


## COMPILATION

User programs are entered into the computer in a source-language, translated into binary form by a process executing a compiler, and stored on disc (Figure 2-5A). Because the code making up a compiler is re-entrant, it can be shared by many users. Therefore, only one copy of a compiler is required in memory no matter how many programmers need it at one time to compile their programs.

MPE/3000 regards user source-language programs as consisting of program units. A program unit is the smallest divisible part of any program or subprogram. For example, in SPL/3000, this is an outer block program unit or a procedure; in FORTRAN/3000, this is a main program, subroutine, function, or block data unit. In COBOL/3000, this is a main program or section. Thus, a *program* consists of one or more program units, one of which must be an outer block (SPL/3000), or main program (FORTRAN/3000 or COBOL/3000) program unit. A *subprogram*, which is the smallest individually-compilable entry, in turn, consists of one or more program units.

When a source-language program is compiled, each program unit is transformed into a relocatable binary module (RBM) that contains user code plus header information that labels and describes this code.

COMPILATION | PREPARATION | ALLOCATION | EXECUTION

Source Program Card Images → Compiler → RBM's → Segmenter Subsystem → Code and Data (Stack) Segments → Loader → Code and Stack Segments Linked to External Segments → User Process

Source File

USL (object) File on Disc

Program File

A | B | C | D
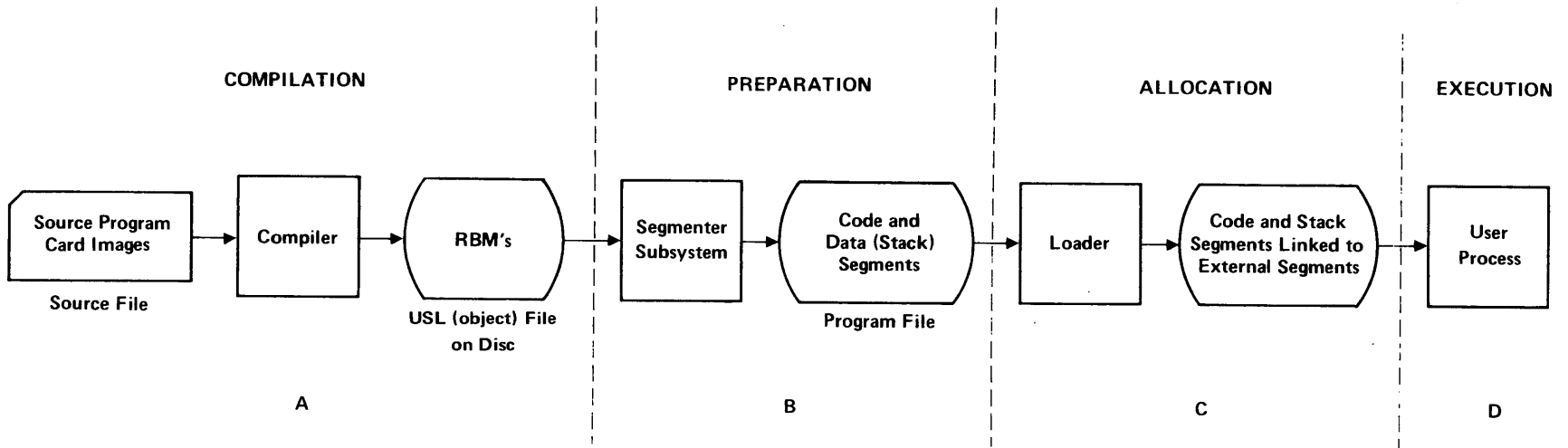
Figure 2-5.  Program Management

## PROGRAM PREPARATION

The RBM's that result from compilation of a user's program onto disc make up a user sub-program library (USL) file. For each program unit compiled, there is one RBM in the USL.

The USL is not executable, however. Instead, it must be *prepared* for running by MPE/3000. During preparation, a process running the MPE/3000 Segmenter binds the RBM's from the USL (object file) into linked code segments arranged in a *program file* (Figure 2-5B). Each segment contains machine instructions produced from the user's program, plus linkages to other segments. The code in a segment cannot be altered because it is treated as read-only information; thus, it remains re-entrant and can be executed repeatedly by many users. When a program is segmented, a special segment for the input of user data is also initially defined; this contains the stack.

## ALLOCATION/EXECUTION

When a program is run, it is allocated and executed. In *allocation*, a process running the MPE/3000 loader binds the segments from the program file to referenced external segments from a library (Figure 2-C). MPE/3000 then creates a process to run the program (Figure 2-D).

*Execution* now begins. As it progresses, many new processes may be created, run, and deleted. For each process in execution, one or more code segments and one stack, operating under control of a process control block (PCB), typically exist in main memory. Not all code segments belonging to a process in execution need exist in main memory simultaneously. Typically, a process operates on a set of segments that are dynamically swapped between main memory and disc. MPE/3000 maintains records of the frequency each segment is used, so that those used least frequently in main memory become the most eligible for swapping out. The user never needs to determine whether a segment is in main memory or on disc at any given time — this is always done for him automatically by MPE/3000.

During allocation/execution, MPE/3000 keeps track of each code segment by maintaining information about its nature and current location in the code segment table (CST). Similarly, information about the stack is dynamically recorded in the data segment table (DST).

A particular program can be run by many user processes simultaneously, with all processes accessing the same copy of code. But unlike the program code segments, the data segment containing the stack is private to each user's process and cannot be shared among others. As execution progresses, data enters and leaves the stack dynamically. Within the stack, data is arranged as a linear group of items accessed from one end (called the *top-of-stack*). When the last instruction in a process is executed, MPE/3000 releases those segments associated only with that process, including the stack segment, and deletes their related entries in the CST and DST. (However, shared code segments are not released until the last process using them is deleted.) All descendants of this process are deleted, and all files opened by it are closed. (CST and DST entries assigned to the released segments can be re-assigned.)

## PCB/CODE SEGMENT/STACK INTERACTION

Each PCB contains all information needed to control a process. This information includes the priority of the process, and pointers to queues and ancestor and descendent processes.

Each code segment executed by a process can contain one or more program units that include calls to procedures elsewhere in this segment or in another segment. Within a program unit, there are generally many executable machine instructions. Each procedure call also references machine instructions that handle the procedure requested.

When a code segment is executing in main memory, it is defined by pointers in three hardware registers: the Program Base (PB), Program Counter (P), and Program Limit (PL) registers (Figure 2-6). (There is only *one* set of these registers; at any particular instant, their contents refer to the code segment currently in execution.) The PB register contains the absolute address of the starting location of the segment in main memory. The P register holds the absolute address of the instruction currently being executed. The PL register indicates the absolute address of the last location of the code segment. Execution can only be transferred from this segment by an interrupt or by a call or exit instruction referencing a procedure in another segment, in which case the PB, P, and PL registers are re-set to reflect the characteristics of the new segment. Whenever an instruction is executed, the PL and PB registers are checked to ensure that referenced addresses fall within the proper segment boundaries. This *bounds check* guarantees that other programs and the system itself are protected against improper access. Since all addresses within a code segment are relative to the contents of the three program registers, a segment can be relocated anywhere in main memory and only the register contents need be changed to reflect this transfer.
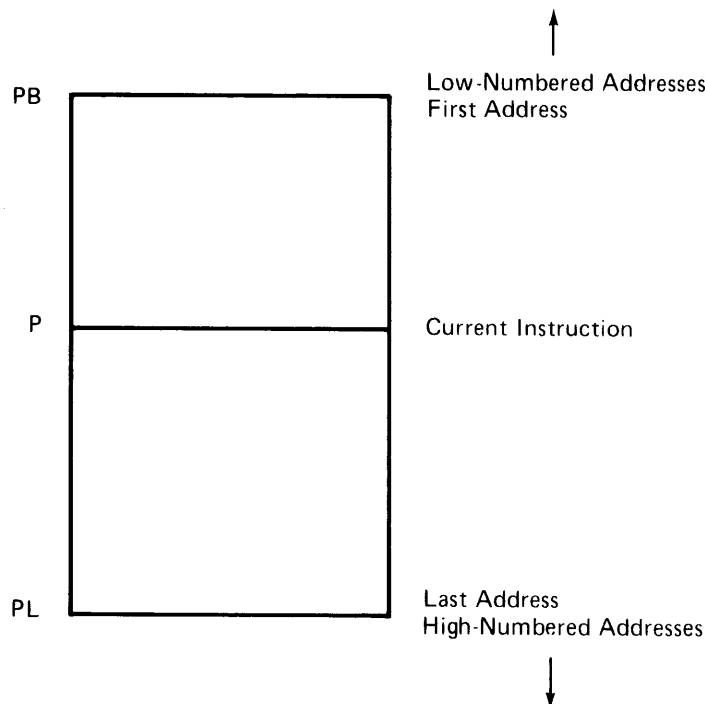


Figure 2-6. Code Segment and Associated Registers

As with code segments, there are normally several stacks resident in memory. (One stack exists for each process.) But since the execution of any process is interleaved with that of others, only one stack is active at any particular instant. Most dynamic computational operations take place on the stack. The last element added to the stack is placed in the word at the *top-of-the-stack*. In this position, it will be the first element removed when the process associated with the stack requests data from the stack. (In other words, the last element in is the first one out.) Each time data is added to the stack, the previous top element becomes the second element from the top; each time the topmost element is removed, the second element returns to the top of the stack.

Programmers operate *directly* on elements in the stack only when using the SPL/3000 language. However, any program in any other language references the stack implicitly when it manipulates data, although the user need not be aware of this.

The stack data segment consists of two general areas: a process control block extension (PCBX) area and a stack area. The *PCBX* contains certain frequently-needed information for process control that need not reside in main memory, such as register settings and file pointers; it is always contiguous with the stack segment. (In general, the PCB is used when the process is *not* running in main memory; the PCBX is used when it is.) The *stack area* is the most significant part of the stack data segment; this is the area where the user's data is stored and manipulated. The boundaries of this area, and its logical subdivisions, are delimited by pointers in registers (Figure 2-7). (At any instant, these registers always apply to the stack belonging to the currently-executing process.)

The Data Limit (DL) register contains the absolute address of the first word of the stack area in main memory. The area between this address and that stored in the Data Base (DB) register is an SPL/3000 *user-managed area* that can be accessed and used only through SPL/3000 programs, such as compilers. These programs sometimes require this space for SPL/3000 own-arrays. (A part of this user-managed area, that between the addresses DB-10 and DB-1, is the subsystem data area reserved for data used during subsystem operation.)

The DB register points to the beginning of the *global area* within the stack area. This area is used for global variables (those declared within the data group of an SPL/3000 main program, and thus usable by any procedure within that program). This area also contains global arrays and pointers to those arrays. The end of the global area is initially indicated by the Stack Marker (Q) register, which also denotes the beginning of the *local area*. (As will be illustrated later, the Q-register pointer changes whenever the running process calls or exits from a procedure.) At any given time, the local area contains data local (relating only) to the procedure currently in execution. The end of the local area is delimited by the Top-of-Stack (S) register, which points to the last item in the stack. All storage between the addresses in the S register and the Stack Limit (Z) register, is unused space that remains available for additional data. The Z register indicates the last main memory location that can be used by user data in the stack area. Beyond the address in the Z register, a special zone is available for stack overflow — a condition that occurs when the S-register pointer must be moved beyond the Z-register pointer and space must be provided for certain end-of-stack information.
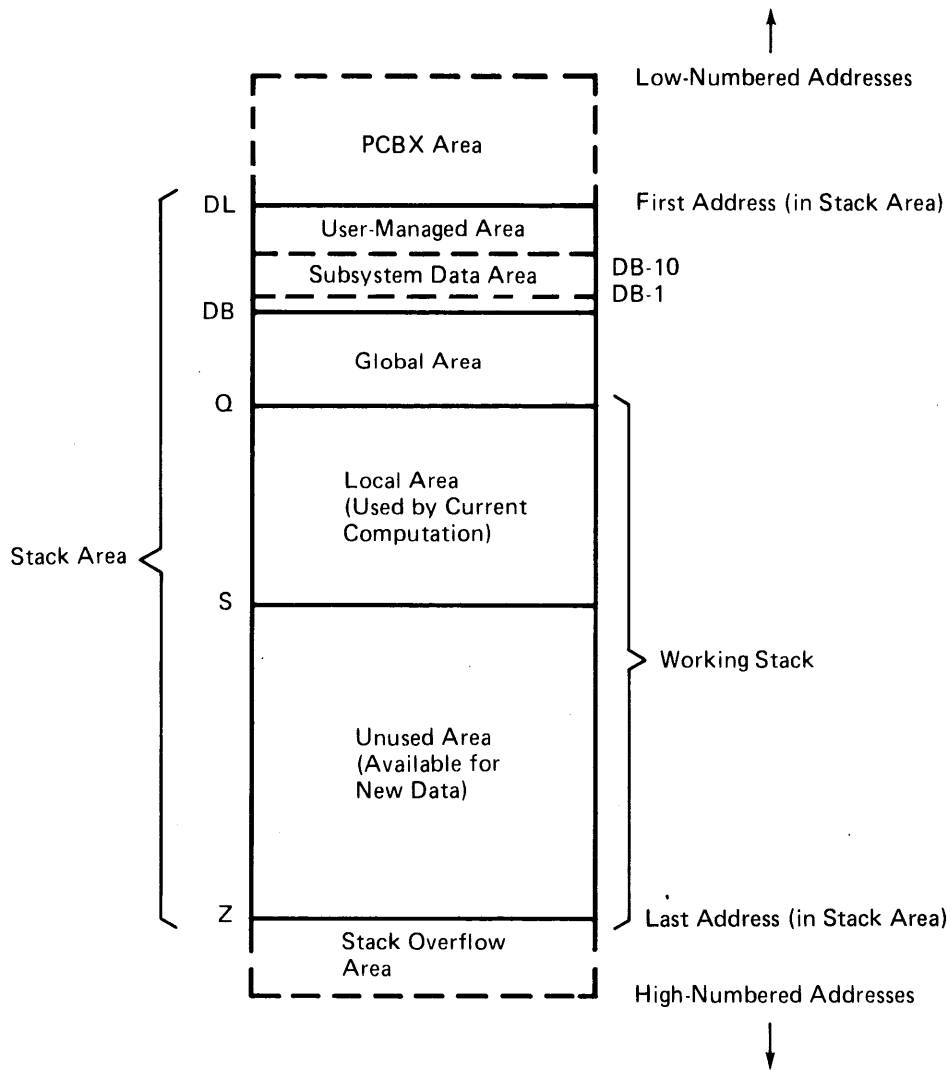
**Figure 2-7. Data (Stack) Segment and Associated Registers**

The contents of the DL and Z registers delimit the boundaries of the stack area. Through bounds checks that reference these registers, MPE/3000 (and certain hardware provisions) ensure that the user's data remains within these limits, and that no other user accesses this area. Thus, the privacy of the user's stack is guaranteed. All locations in the stack are addressed relative to the DB, Q, or S addresses.

Although the top-of-stack is logically indicated by the S register, up to four of the topmost elements of the stack can actually exist in central processor registers (the TR registers) rather than in main memory — in effect, the stack "spills over" from memory into these registers. These conditions greatly enhance processing speed. The number of TR registers currently in use is indicated by the SR register. The absolute address of the last item of the stack actually residing in main memory is stored in the SM register. The contents of the S register are actually denoted by

$$S = SM + SR$$

Thus, the S register is said to contain the *logical* top-of-stack rather than the actual top-of-stack in memory.

Before a process begins execution, stack space is first reserved for global data, beginning at the DB-address and terminating at the Q-address, which denotes the beginning of the dynamic *working stack* (Figure 2-8A). At this time, no data is stored beyond the Q-address, and so the the S register also points to the same address as the Q register — that is, the bottom and top of the working stack coincide. But, as the process begins execution, data is added to the stack, and the S-pointer (top-of-stack) moves away from the Q-pointer (Figure 2-8B). If, at some point, the process encounters a procedure call, a new area for data local to that procedure must be defined. To do this, the system hardware places a group of four words called the *stack marker* on top of the stack to save information necessary to re-create the currently-defined local area later. The Q and S registers are then pointed to the top word of the stack marker, which also delimits the beginning of a new, fresh and unique local area for the procedure just called (Figure 2-8C).

The words in the stack marker preserve the state of the machine at the time of the procedure call. These words contain the following information, (shown in order ascending toward the Q-address):

| Word | Contents |
|------|----------|
| Q-3 | Current contents of the Index (X) register. |
| Q-2 | The return address for the *code* segment, denoted by P+1 (relative to the PB register). |
| Q-1 | The current contents of the Status register (which includes the number of the code segment containing the calling procedure). |
| Q-0 | The delta-Q value, which is the number of words between the new and previous Q-locations, enabling the later re-setting of Q to its old value. |

## A

DL

User-Managed Area

DB

Global Area

Q,S

Z

A

## B

DL

User-Managed Area

DB

Global Area

Q

Main Program Local Area

S

↓

Z

B

## C

DL

User-Managed Area

DB

Global Area

Q(old)

Main Program Local Area

S(old)

Stack Marker

Q(new),S →

Z

C

## D

DL

User-Managed Area

DB

Global Area

Q(old)

Main Program Local Area

S(old)

Stack Marker

Q(new)

Procedure's Local Data

S(new)

↓

Z

D

## E

DL

User-Managed Area

DB

Global Area

Q

Main Program Local Area

New Data

S

Z

E

Figure 2-8. Stack Operation

2-18

As data is added to the stack during execution of the new procedure, the S-pointer moves away from the new Q-pointer, reflecting the latest data added (Figure 2-8D). When the procedure exits back to the main program, the new local data area is deleted from the stack, the stack marker is used to restore the Q-pointer to its previous setting, any value returned by the procedure is left at the new top of the stack, and the S-pointer is set to indicate the new top-of-stack (Figure 2-8E). This results in a "clean" stack from which temporary data local to the called procedure is eliminated because it is no longer needed.

Whenever a procedure is called, the Q and S registers are manipulated in this manner. The Q register changes with each procedure call and exit; the S register may change when an instruction references data. Thus, when a process executes a main program (outer block) that calls three procedures, there will be a maximum of four local areas (one for the main program and three for the procedures called by it) on the stack. Each procedure's local area will be delimited at its base by its own stack marker. Within these stack markers, the delta-Q words will form a logical chain that links the present Q register setting back to its initial value.

## MEMORY MANAGEMENT

The MPE/3000 virtual memory, which frees the user from the restrictions of a physical memory of constant size, consists of main memory and portions of disc. Both the dynamic swapping of segments from disc to main memory, and their dynamic relocation within main memory, are controlled by the Memory Management System through segment descriptors in the CST and DST. When additional memory is required during execution of a process, executed code segments are overlayed by incoming segments rather than actually being swapped out; this saves considerable memory overhead. (If needed later, re-entrant copies of these code segments can be brought in again from disc.) Because the content of a data segment changes continually, however, it is sometimes necessary to swap such a segment both to and from disc to preserve and maintain it until the process completes execution.

Since code can be shared between programs, both the need for multiple copies of programs or routines, and time devoted to swapping between disc and main memory, are reduced.

The user's stack is not the only data area handled as a segment in MPE/3000. In addition, extra data segments (auxiliary to user stacks) and input/output buffers are also managed in this way by the Memory Management System.

### Main-Memory Linkages

Main memory is organized into a sequence of variable-length, linked areas that facilitate the insertion or deletion of segments. The memory linkages contain the following information about each area:

- Availability (in use or available)

- Size

- Type (code or data)

- Table pointer (to the CST or DST entry relating to the area)

Those areas that are available are linked together through an Available-Space List; as areas in use are released, they are added to this list. Adjacent available areas are combined to form one contiguous area. Requests for memory can be implicit (through a CST presence-trap) or explicit (through calls from the system or user). To allocate memory, a space-allocation routine interrogates the Available-Space List to find the first area large enough to satisfy the request. If the search succeeds, all or a portion of that area is allocated. If the search fails, one or more overlays is performed to free an area of adequate length.

Assigned storage is linked according to the priority and frequency of its access. Frequently-accessed segments with high priority are located at or near the end of the Assigned-Storage List; infrequently-accessed segments with low priority are located near the head of this list. When an overlay is required, the list is searched, beginning with the head entry. Segments declared main-memory resident, of course, are not overlayed. If a data segment is selected for overlay, and it has been modified in memory, it is copied to a reserved overlay-storage area on disc. (Code segments are overlayed without being copied, since copies already exist on disc.) Each process is assigned such a reserved area when it is initialized; it may contain either code or data segments.

## Main-Memory Use

When MPE/3000 is initialized, part of main-memory (usually in the low-address area) is pre-allocated to MPE/3000 or to the hardware for resident routines and tables; this area is not available for swapping. The remaining portion of main memory is established as the linked areas described above. During MPE/3000 initialization, some of this space is allocated for the CST, DST, and PCB tables, and the remainder is available for overlays (and so described in the Available-Space List).

## USER JOB PROCESSING

The following paragraphs discuss how MPE/3000 concepts interrelate, from the standpoint of the user's program.

### Batch Programs (Jobs)

To illustrate how a typical batch program (a job) is processed, consider a small FORTRAN source program punched on cards. The user arranges the deck so that the MPE/3000 commands, the FORTRAN program, and any input data are presented in the proper order, and submits this deck to the computer operator to run.

The operator enters the job through a card reader. MPE/3000 assigns the job a unique system job number, and schedules it for access to the central processor. When the central processor is available, a special system process determines whether the attempted access is valid (by checking the user's identity and passwords) and requests the user controller process (UCOP) to create a job main process (JMP) that uses Command Interpreter code to run the job (Figure 2-9A, B). The JMP completes the job initialization and becomes the process responsible for handling interactions (commands) between the user and MPE/3000 (Figure 2-9B).
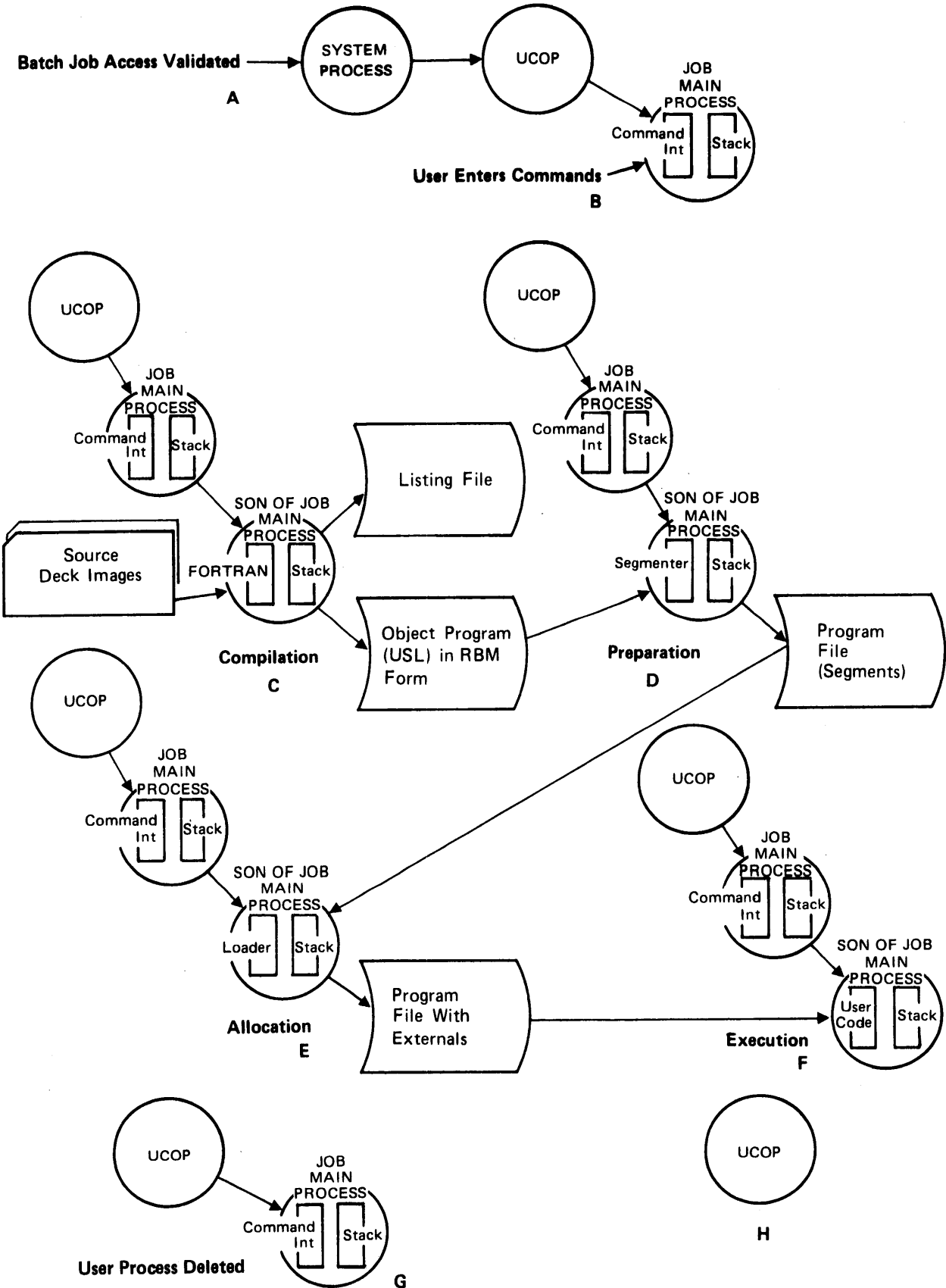
Figure 2-9. Batch Job History

When this process encounters a user request to compile, prepare, and execute the user's program, it creates another process that calls the FORTRAN/3000 compiler into virtual memory and executes the compiler code. (Figure 2-9C). This process translates the FORTRAN program from source to object code and organizes this code into RBM's stored as a USL on disc. In this process, the compiler interacts with the Input/Output and File Management Systems to read input and write output. Next, the main process creates a process that runs the segmenter subsystem, which reads the program from the USL and prepares it for execution by arranging it as a set of linked segments on a program file (Figure 2-9D). Then, the loader allocates the segments in virtual memory on disc, and satisfies external references to library routines (Figure 2-9E). The initial stack area is also defined at this time. Then, the code segment containing the entry point of the user's program, and the stack segment, are moved into main memory and execution of the user's process begins. (Figure 2-9F). Files needed by the job, and their associated devices are assigned as requested.

As processes are created by and for the job, they are entered into queues and scheduled for access to the central processor. Each process may create many descendent processes as the job progresses. Various processes share the central processor through multiprogramming, and in this way are run concurrently. The code segments belonging to each process obtain and operate on data from the stack associated with that process. As each process proceeds, segments are swapped into main memory, as they are needed, by the Memory Management System, which calls the Input/Output System to handle this function. Data enters and leaves the stack area dynamically, handled by the File Management System. Program output is transmitted to a line printer and printed by an input/output process as the job is run.

As the job progresses, the operator receives any pertinent messages. He can terminate the job at any time. When any process (including the main process) is completed, it and its descendants are deleted from the system; all files opened by this process are closed (Figure 2-9G, H). As the job is completed, system information related to the job is printed, including the time and date, and central processor time used.

## Interactive Programs (Sessions)

As an example of how an on-line, interactive program is handled, consider a BASIC-language program run by a user at a remote terminal.

The user contacts the system by turning the terminal on (if the terminal is connected directly) or by dialing the system (if the terminal is connected by switched telephone lines). In response, a special system process determines the validity of the access and requests the UCOP to create a session main process (SMP) that uses the Command Interpreter code. The SMP completes the session initialization and becomes the process responsible for MPE/3000 interactions (commands) between the user and the system. The user next enters a command to access the BASIC/3000 Interpreter. A process is now created that handles input in the BASIC language. (Only one copy of the BASIC/3000 Interpreter is needed in main memory no matter how many users are programming simultaneously in BASIC; however, each user accesses the interpreter through a separate process.)

Many independent user processes share the central processor through time-slicing. Some commands create processes that may, in turn, create descendant processes. When required by each process, data segments are swapped to and from main memory. The Memory Management System interacts with the Input/Output System to accomplish this. The File Management System is invoked to allocate devices for files whenever the files are opened. The user can direct output to the terminal, or he can have it transmitted to a printer.

The user can terminate the session at any time. When he does this, the SMP prints the time and date, central processor time, and terminal connect time.

## ACCOUNT/GROUP/USER ORGANIZATION

When a user logs on to MPE/3000, two basic elements must be defined: an identifiable unit to which system resources (such as disc file space and central processor time) are allocated and charged, and a local set (domain) of disc files accessible by the user. The basic unit to which resources are assigned is the *account*; this is the major "billable unit" in MPE/3000. Associated with each account is a unique file domain, a set of *users* who can access MPE/3000 through this account, and a set of *groups* which partitions the account's accumulated resources and divides its file domain into private sub-domains.

Each account is defined, modified, and deleted by commands issued by a user with the MPE/3000 System Manager Capability, who has ultimate control over access to the system and allocation of its resources. Each account is identified by a name. Optionally, a password can be associated with the account to validate a user's ability to access MPE/3000 under this account at log-on time. A maximum priority also is associated with the account; this designates the highest priority at which any process run under this account can be scheduled.

Limits are assigned for maximum disc file space, central processor time, and on-line connect-time permitted each account; running counts of the use of these resources are maintained for billing purposes. To maintain an account, the user acting as System Manager grants a user the Account-Manager Capability. This account-managing user may in turn, assign the same capability to other users in his account.

The users and groups associated with each account are defined by commands issued by the account-managing user. Each user is identified by a name (unique to this account) and optional log-on password. He is assigned a maximum allowable priority for his processes, which cannot exceed the priority allowed to his account. Each account possesses a public group (to which all of its users have read and program-execution access) in addition to other groups that may be covered by various security provisions. Each group is identified by a name unique within its account, and optionally, by a password used to validate access to the group and its files at log-on time.

As with an account, limits are assigned for the maximum disc file space, central processor time and on-line connect time usable by a group; and running counts of resources used by the group

are maintained. (File space is always charged to the group containing the file, rather than the group to which the user who created the file was logged on.)

Any MPE/3000 installation can contain several accounts; each account can have several users and groups associated with it; each group can possess several files (see Figure 2-10) which constitute a subset of the file domain. When the user logs on, he specifies the account, user, and group names (and, if required, the account, user, and group passwords). Furthermore, any file in a group may also be protected by a *lockword* required at any time the user accesses the file during the course of his job or session (in addition to standard file security mechanisms described later.)

Each user can be associated with a *home group* by the user managing his account. If the user does not specify a group when he logs on, he is given the home group by default.

Once the standard user has established communication with MPE/3000, if the normal (default) system security provisions are in force, the user has unlimited access to all files in his log-on group and home group. Furthermore, he can read, and execute programs residing in, files in the public group of his account or in the public group of the system account. He cannot, however, access other files in the system in any way.

The normal MPE/3000 security provisions can be overridden at the account, group, or file level, (by System Manager, Account Manager, or standard users, respectively) to provide more or less restriction to users. Furthermore, users with special capabilities (discussed next in this section) are generally subject to fewer restrictions.


## CAPABILITY SETS

The HP 3000 Computer is used by a large variety of programmers, ranging from those who want to run simple applications programs in BASIC to system programmers who are actually modifying MPE/3000. To protect the system and its users in general, users with System and Account Manager Capabilities can limit access to special system capabilities only to those who fully understand their correct use. This is done through capability sets. Specifically, when a System-Manager User creates an account, he defines for it a capability set that determines whether or not users communicating with MPE/3000 through this account can be allowed certain functions. When an Account-Manager User defines the users of his account, he associates with each user an individual capability set that may allow the user some or all of the general account capabilities. Each capability set contains three types of attributes: user, file-access, and capability-class. A fourth attribute, the local attribute, may also be defined. The combination of these attributes determines the set of commands and intrinsics available to the user. This division of commands and intrinsics greatly simplifies use of the system from the standpoint of each individual user — it defines the extent to which he must understand and interrelate with MPE/3000, and permits a user to ignore aspects of MPE/3000 that do not apply to him.

Capability sets are also defined for groups by the Account-Manager User. Group capability sets contain only one type of attribute — the capability-class attributes. The capability set for a group may allow that group some or all of the capability-class attributes defined for the account to which the group belongs. The group capabilities relate to the user's capabilities as noted at the end of this section under *Program Capability Sets.*
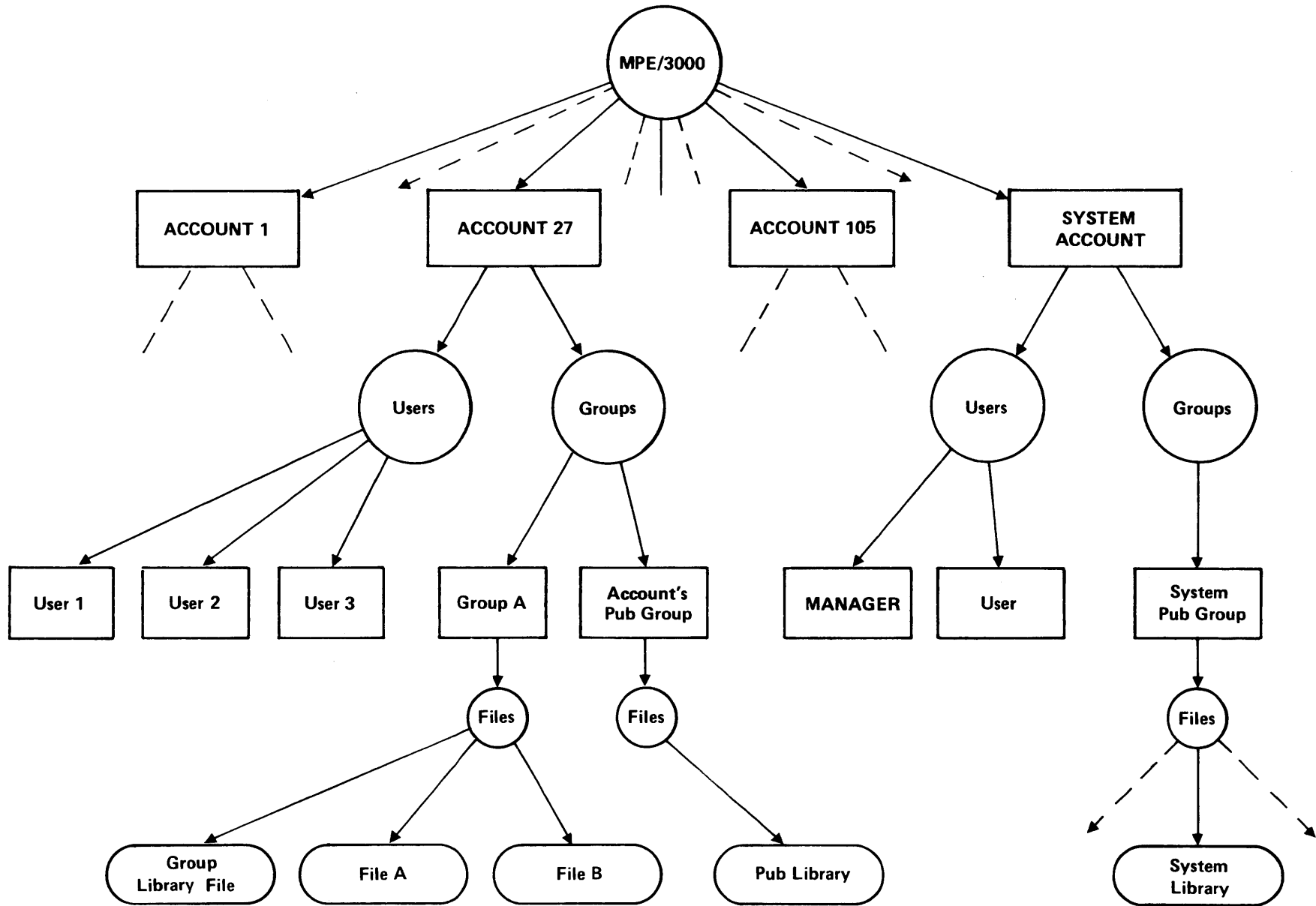
Figure 2-10. Account/User/Group Organization

As noted at the end of this section, capability-class attributes are also associated with each program on a program file, passed as parameters (in the command that prepares the program) to the MPE/3000 Segmenter.

## User Attributes

The user attributes designate the general level at which the user interfaces with MPE/3000. These attributes can be assigned in any combination, and define capabilities *in addition* to those of a standard user.

SYSTEM MANAGER ATTRIBUTE. Grants the user the capability to manage the overall system and create the accounts within it. The first user with the System Manager Attribute is designated on the system tape furnished the customer. He, in turn, can designate other users having the same or different capabilities.

ACCOUNT MANAGER ATTRIBUTE. Allows the user to manage all users and groups within an account. The first manager for each account is designated by a user with the System Manager Attribute when the account is created. A user with the Account-Manager Attribute, in turn, can assign this attribute to other users in his account.

SYSTEM SUPERVISOR ATTRIBUTE. Allows the user to have day-to-day external control of this system. It allows him to manage scheduling subqueues, alter the system configuration, maintain the system library, and display various items of system information. This attribute may be assigned by a user with the System Manager Attribute.

ACCOUNT LIBRARIAN ATTRIBUTE. Can be assigned to grant a user special file-access modes for maintenance of files within his account. For example, an Account Librarian Attribute may be used to designate users who can purge (but not create or alter) files within the account. (File-access modes such as read-access or write-access, are discussed in Section V.) This attribute is assigned by users with the Account Manager Attribute.

GROUP LIBRARIAN ATTRIBUTE. Similar to the Account Librarian Attribute, but limits the special file-access modes allowed the user to his home group. This attribute is assigned by users with the Account Manager Attribute. It could be used, for example, where it is desired that only one user can have the capability to alter files within a particular group. This user could be assigned the Group Librarian Attribute and his access modes (as defined in Section V) could be made greater than those of other users.

DIAGNOSTICIAN ATTRIBUTE. Permits the user to run diagnostic programs for on-line check-out of hardware under the System Diagnostic Monitor (SDM/3000). This attribute is assigned by users with the Account Manager Attribute.

## File Access Attributes

The file-access attributes determine whether the user has the capability to

- Save user files permanently

- Access card-readers, line printers, and other non-sharable input/output devices (other than the standard job/session input and listing devices which are available to all users)

These attributes can be assigned in any combination.

## Capability-Class Attributes

These attributes define the general MPE/3000 resources available to a user. They allow the user to

- Access MPE/3000 in an inter-active on-line mode

- Access MPE/3000 in local, batch processing mode

Most users have *only* these MPE/3000 Standard Capabilities; all users except console operators must have one of these capabilities *at least.*

- Create, handle, and delete processes directly

- Manage data segments (by creating extra data segments)

- Have exclusive use of more than one system resource simultaneously

- Operate in privileged mode (which permits a user to access *all* MPE/3000 resources, including uncallable intrinsics)

MPE/3000 Optional Capabilities

Either the Interactive or Batch-Access Attribute is required to communicate with MPE/3000; most users are assigned both as *standard capabilities*. The remaining capability-class attributes are *optional capabilities*. They are independent and can be assigned in any combination. In Part 3 of this manual, the optional capabilities are discussed in the order of the increasing power they give the user. Programmers should be aware, however, that the more powerful the optional capability, the more hazardous its misuse is to the system. Thus, they should use optional capabilities with caution.


## Local Attributes

The local attribute is a two-word quantity used only for special applications that require further unique classification of users. MPE/3000 does not reference or make use of this attribute in any way; rather, it is defined arbitrarily by System or Account Managers, and used by accounts or groups for whatever purpose they require.


## Program Capability Sets

Capability sets are always associated with prepared programs as well as users.

Each time someone runs a program, MPE/3000 automatically assigns that program the user and file-access attributes of that user. But the capability-class attributes assigned to the program are designated by the user who originally prepares the program; they are passed to the MPE/3000 Segmenter as parameters of the command that prepares the program. If the preparing user does not designate capability-class attributes for his program, MPE/3000 assigns, by default, the standard capabilities possessed by that user — interactive access, batch access, or both. (When programs prepared from passed files or job temporary files (explained in Section V) are run, they are assigned the standard capabilities (interactive and/or batch access) possessed by the user who runs them.)

If the program resides on a permanent file, the program's capability-class attributes should not exceed those defined for the *group* to which the program file belongs. If they do, the user will not be able to run the program when he attempts to do so.

Because the capability set is associated with the entire set of code segments being run (and hence with the process running them), all procedures, subprograms, and subroutines on those code segments have the same capability. For the same reason, a *user* need not have the same capabilities as the programs he runs.

# PART 2
## Standard Capabilities

# SECTION III
# Communicating with MPE/3000

To communicate with MPE/3000, the user issues commands and intrinsic calls.

*Commands* are requests issued to MPE/3000 to perform various broad functions external to the user's program. For example, they are used to initiate and terminate jobs and sessions, create and maintain files, compile and execute programs, call various utility subsystems, and obtain job status information. Commands can be entered through any standard input device, typically the card reader (for jobs) or the terminal (for sessions). Each command is accepted by the MPE/3000 Command Interpreter, which passes it to the appropriate procedure for execution. Following this execution, control returns to the Command Interpreter.

*Intrinsic calls* are used to invoke MPE/3000 functions requested within a user's program, such as reading, writing on, and updating files, skipping forward and backward on files, or returning system table information to the user's program. In an SPL/3000 program, the user writes the intrinsic calls explicitly. In FORTRAN/3000, BASIC/3000, or COBOL/3000 programs, for most applications, the compiler generates any necessary intrinsic calls automatically—they are invisible to the user. At their option, however, FORTRAN/3000, COBOL/3000, and BASIC/3000 users can also directly call intrinsics as their needs require, providing added power and flexibility to these standard programming languages.

The programmer can use intrinsic calls to invoke the Command Interpreter from within his program, and pass to it command images that will be interpreted and executed as the corresponding system commands.

The commands and intrinsic calls discussed in this part of the manual (Part 2) allow the user to initiate batch jobs and interactive sessions, access the file system, compile and execute programs, and use other *standard capabilities* of MPE/3000. They do not, however, permit him to handle processes, directly access the computer hardware, or use other *optional capabilities*; the commands and intrinsic calls for these capabilities are explained in Part 3.

## COMMANDS

Each command entered by the user, whether in a batch job or interactive session, consists of

- A colon (used as a command identifier)
- A command name
- A parameter list (in most cases)

3-1

The end of each command is delimited by the end of the record on which it appears—for example, a *carriage return* for terminal input or the end of the card on which it is punched for card input. But, if the last non-blank character of the record is a *continuation character*, as defined later in this section, the end-of-record does not terminate the command. (Users running programs in batch mode should bear in mind that all 80 columns on each card image are scanned by MPE/3000, and thus no characters are ignored.)

The *colon* identifies a statement as an MPE/3000 command. In a batch job, the user begins each MPE/3000 command with this colon in column 1 of the source card (or card image). In an interactive session, however, MPE/3000 prints the colon on the terminal whenever it is ready to accept a command; the user responds by entering the command after the colon. (Interactive subsystems of MPE/3000 also use unique prompt characters; in a session, the prompt character output tells the user that a subsystem is ready.)

The *command name* requests a specific operation, and appears immediately after the colon. Imbedded blanks are not permitted within the command name. The end of the command name is delimited by any non-alphabetic character, normally a blank.

The *parameter list* contains one or more parameters that denote operands for the command. It is required in some commands but optional or prohibited in others. Parameter lists can include positional parameters and/or keyword parameter groups. Within the parameter list, delimiters (commas, semicolons, equal signs, or other punctuation marks) are used to separate parameters or parameter groups from each other, as described below.

Normally, the parameter list itself is separated from the command name by one or more blanks. However, when the first optional parameter in a positional list (as defined below) is omitted, the command name can be followed immediately by any other delimiter. (At the user's option, he can include one or more blanks between the command name and this delimiter.) Any delimiter can be optionally surrounded by any number of blanks, permitting a free and flexible command format.

*EXAMPLE:*

*The following command (RUN) is preceded by a colon and includes a parameter list with two parameters:*

**:RUN PROG, ENTRYX**

Both decimal and octal numbers are permitted as command parameters. Octal numbers, however, are always preceded by a percent sign (%).

## Positional Parameters

With positional parameters, the meaning of the parameter is designated by its position in the list. For example, in the MPE/3000 command to compile a FORTRAN/3000 program, the parameter specifying an input file always precedes the one that specifies an output file. Positional parameters are mutually-separated by commas or semi-colons. The omission of an optional positional parameter from within a list is indicated by two adjacent delimiters; the

omission of a positional parameter that would otherwise immediately follow a command name is indicated by a comma or semi-colon as the first character in the parameter list. When parameters are omitted from the end of a list, however, no adjacent delimiters need be included to signify this — the terminating *return* or end-of-card is sufficient.

*EXAMPLE:*

*The first command below has its first parameter omitted; the second command has its second (embedded) parameter omitted; the third command has its last two (trailing) parameters omitted; the fourth command has all parameters omitted. (In the second and third commands, the asterisk (\*) is not a delimiter, but a special character used to denote a back-reference to a previously-defined file, as described in Section V. In each case, the asterisk is considered part of the following parameter.)*

```
:FORTRAN  ,USFL,LISTFL,MFL,NFL
:FORTRAN  *SOURCEFL,,LISTFL,MFL,NFL
:FORTRAN  SOURCEFL,USFL,*LISTFL
:FORTRAN
```

A further example illustrates the relationship between the positions of parameters in a list and their meanings:

*EXAMPLE:*

*In the following command (:FORTRAN), three positional parameters appear: INP refers to an input source file, OUT indicates an object (USL) output file, and LST indicates the listing output file. For the :FORTRAN command, these three fields always have the same meanings. (Note that the second delimiting comma in the parameter list is followed by an optional blank. In future examples, for clarity, delimiters will always be followed by blanks.)*

```
:FORTRAN INP,OUT, *LST
```

## Keyword Parameters

When a parameter list is so long that use of positional parameters becomes difficult, keyword parameter groups are often used. The meaning of such a group is independent of its position in the list. A keyword parameter group is designated by a keyword that denotes its meaning, and *optionally* an equal sign and one or more sub-parameters. (Each keyword group is preceded by a semi-colon. When more than one sub-parameter appears in a group, they are usually separated from each other by commas. Like other delimiters, semi-colons and commas can be optionally preceded or followed by blanks.) With respect to each other, keyword groups can appear in any order. When keyword groups and positional parameters both appear in a list, however, the positional parameters always precede the keyword groups; when this occurs, and

*trailing* parameters are omitted from the positional group, their omission need *not* be noted by adjacent delimiters; instead, the occurrence of the first keyword indicates this omission.

*EXAMPLE:*

*In this example, DL and CAP designate keyword parameter groups. PH, DS, and MR are sub-parameters of the keyword CAP.*

```
:PREP INPT, OUTP; DL=500; CAP=PH, DS, MR
```

## Continuation Characters

When the length of a command exceeds one record (source card or entry-line), the user enters an ampersand (&) as the last non-blank character of this record and continues the command on the next record. In this case, the next record must begin with a colon (entered by the user in batch processing, but prompted by the terminal in interactive processing). Optionally, blanks can be embedded between the colon that begins a continuation record, and the rest of the information on that record. Commands can be continued up to 255 characters, including prompting colons and continuation ampersands.

*EXAMPLE:*

*The following command image contains a continuation character at the end of the first line:*

```
:RUN PROGB; NOPRIV; LMAP; STACK=500; PARM=5; &
: DL=600; LIB=G
```

In continuing a command onto another line, the user cannot divide a primary command element (a command name, keyword, positional parameter, or keyword sub-parameter) — no primary element is allowed to span more than one line.

MPE/3000 does not begin interpretation of a command until the last record of the command is read. For interpretation, all records within the command are concatenated, and all prompt characters and continuation ampersands are replaced by one or two blanks.

## Command Description Format

To help clarify the command descriptions that appear throughout this manual, system output is *underlined*. Input information is not underlined. (In cases where differences in output occur between batch and interactive processing, *interactive* output is assumed unles otherwise noted.) For both input and output, literal information that always appears exactly as shown is designated by *CAPITAL LETTERS AND SPECIAL CHARACTERS IN ITALICS;* on the other hand, symbols that represent variable information are indicated by *lower-case italics.*

Optional information is indicated by surrounding [*brackets*]. (However, the user does not enter these brackets as part of the command.) {*Braces*} indicate that one of the items included is required and must be entered by the user.

*EXAMPLE:*

*The colon is output by the terminal during interactive sessions (as shown by the underline); BASIC is a literal command name entered by the user, (as indicated by capital letters); and commandfile, inputfile, and listfile are variable parameters input by the user (as shown by lower-case letters). All three parameters are optional (as indicated by brackets).*

   _:BASIC   *[commandfile] [,[inputfile] [,[listfile] ] ]*

When more than one item is enclosed vertically in brackets, this indicates that exactly one of these items *may* be specified.

*EXAMPLE:*

   _:SHOWJOB   $\begin{bmatrix} \#jsnumber \\ jsname \end{bmatrix}$

When more than one item is enclosed vertically in braces, this means one item *must* be specified.

*EXAMPLE:*

   _:SAVE   $\begin{Bmatrix} \$OLDPASS,newfilereference \\ tempfilereference \end{Bmatrix}$

If items are shown within vertical and disjoint brackets, this signifies that they are all optional and that those specified by the user can appear in any order.

*EXAMPLE:*

   _:RUN  progfile [,entrypoint]
          [;NOPRIV]
          [;LMAP]
          [;MAXDATA = segsize]
            •
            •
            •

## Command Errors

All MPE/3000 commands issued, and any system messages for the user, are copied to the job/session listing device.

When MPE/3000 detects an error in a command, it suppresses execution of that command and prints an error message. If this occurs during a batch job, (and no :CONTINUE command precedes the erroneous command, as discussed later) all information between the erroneous command and the end of the job is ignored, and the job is aborted. If an error occurs during an interactive session, an indication is printed and control returns to the user; he may then re-enter this command correctly, or enter any other command he desires simply by pressing the carriage-return key and entering the command. Alternatively, if he desires a fuller explanation of the error, he can request it by entering any character directly after the error-number shown in the error message. In response, the explanation appears on the next line. Error messages and on-line error recovery are discussed in Section X.

## INTRINSIC CALLS

A major advantage of MPE/3000 intrinsics is that they can be called directly from programs written in standard, higher-level programming languages (FORTRAN/3000, COBOL/3000, and BASIC/3000) as well as from SPL/3000 programs. The rules for invoking intrinsics from programs written in standard languages are presented in the manuals covering those languages. However, because intrinsic calls are issued primarily by users programming in SPL/3000, the rules for issuing these calls from SPL/3000 programs are summarized below.

Before an intrinsic can be called from an SPL/3000 program, it must be declared within the declaration portion of the program, following all data declarations, like any other SPL/3000 procedure. This can be done by writing the standard PROCEDURE declaration. Normally, such a declaration would contain both the procedure head and the procedure body (code). But, since an intrinsic is an *external* procedure, the user follows the normal SPL/3000 conventions for such procedures by writing only the intrinsic *head* and including within it the OPTION EXTERNAL notation. (EXTERNAL signifies that the body code will be supplied from a library and linked to the user's program during program allocation.) In this manual, the complete head format for each intrinsic is presented with the discussion of that intrinsic. (Note that in SPL/3000, each statement is delimited by semi-colons.)

*EXAMPLE:*

*The format for the head of the FREAD (file read) intrinsic is presented in Section VI. Using this format, the user could write a declaration that will enable him to call the FREAD intrinsic later in his program. The declaration could appear as follows. Note that the user must include the OPTION EXTERNAL notation.*

```
INTEGER PROCEDURE FREAD (FN,TAR,TC);
VALUE FN,TC;
INTEGER FN,TC;
ARRAY TAR;
OPTION EXTERNAL;
```

Because some intrinsic heads are rather long, the SPL/3000 programmer can save time and effort by using an alternative method of declaring intrinsics; the INTRINSIC declaration. This is written in the following format

INTRINSIC *intrinsicname, intrinsicname,* . . . , *intrinsicname;*

In the *intrinsicname* list, the user names all intrinsics he intends to call within his program. (When more than one is named, the names are separated by commas.) He then need not write the heads for these intrinsics.

*EXAMPLE:*

*To use the INTRINSIC declaration to declare the FOPEN, FREAD, FWRITE, and FCLOSE intrinsics, the user could write:*

**INTRINSIC FOPEN,FREAD,FWRITE,FCLOSE;**

Regardless of whether the programmer declares an intrinsic with a PROCEDURE declaration or an INTRINSIC declaration, he must know the formal head format for each intrinsic, since this tells him the number and type of parameters that are used in calling that intrinsic.

The user calls an intrinsic in his program exactly as he calls any SPL/3000 procedure: he writes the intrinsic name and a parameter list, enclosed in parentheses. These elements follow the positional format shown in the intrinsic head; parameters are separated from each other by commas.

*EXAMPLE:*

*A call to the FREAD intrinsic could be written as*

**FREAD (INFILE,BUFFER,-80);**

If the OPTION VARIABLE notation appears in the intrinsic head, some of the intrinsic parameters are optional. In this manual, these optional parameters are indicated as **bold face** in the intrinsic head formats.

Since all intrinsic parameters are positional, the user indicates a missing parameter within a list by following the previous delimiting comma with another comma by itself.

*EXAMPLE:*

*The second parameter is missing:*

**FOPEN (FILENAME,,3);**

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

NOTE:    In some intrinsic calls, input parameters are passed to the intrinsic
as words whose individual bits or fields of bits signify certain functions
or options. In cases where some of the bits within a word are described
in this manual as "not used by MPE/3000," the user is advised to set
such bits to zero. This will help ensure the compatibility of his current
programs with future releases of MPE/3000.

In cases where output parameters are passed by an intrinsic to words
referenced by a users' program, bits within such words that are
described as "not used by MPE/3000" are set as noted in the dis-
cussion of the particular parameter.

## Intrinsic Description Format

In this manual, the complete intrinsic declaration head is shown with the discussion of each
intrinsic; included within this is the intrinsic call format, distinguished from the remainder of
the head by a box. This format appears as follows:

procedure type    PROCEDURE    | intrinsic name (parameter list) |
declaration elements

The *procedure type* applies only to *type intrinsics*—those that return a value to the user's
program in response to the intrinsic call. This describes the type of data returned, such as
integer, real, or byte values. The data is returned through the intrinsic name, as demonstrated
in later examples. Additionally, type intrinsics can be directly related to functions called in
FORTRAN/3000 programs, as noted in the manual *HP 3000 FORTRAN (03000-90007)*.

The *declaration elements* describe each parameter (variable, pointer, label, and array) in the
parameter list, and its characteristics. If the VALUE notation is present in the declaration,
either a numeric value (expression) or a symbolic identifier may be specified for the parameters
indicated by this notation. If VALUE is not present, only symbolic identifiers may be used
for parameters.

*EXAMPLE:*

*This is the intrinsic description format for the FREAD intrinsic:*

                                                        *(intrinsic name)*

*(Procedure Type)* ——► INTEGER PROCEDURE    | FREAD (filenum,target,tcount); |

                                                        *(intrinsic parameters)*

*(Declaration* ————————► VALUE filenum, tcount
*Elements)* ————————► INTEGER filenum, tcount;
                              ARRAY target;
                              OPTION EXTERNAL;

3-8
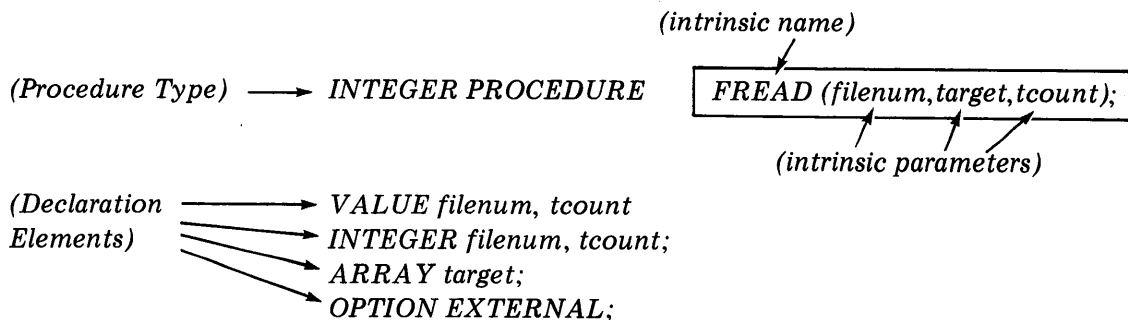
# Intrinsic Call Errors

Some intrinsics alter the *condition code*, made available to the user's FORTRAN/3000 and SPL/3000 programs through the Status register. Since the contents of this register change continually, the user should check this register for condition codes immediately upon return from an intrinsic. A condition code is always one of the following, and has the general meaning indicated. The specific meaning, of course, depends upon the intrinsic called.

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

Two types of errors may be encountered when an intrinsic is executed. The first, denoted by the CCG or CCL condition codes, is generally recoverable 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 class demanded by the intrinsic. The user's program may recover from this error or take some other appropriate action if an appropriate error-trap procedure has been defined (as discussed in Section VIII). Otherwise, his program terminates, and an abort-error message appears on his output device. If the program was entered in a batch job, MPE/3000 removes the job from the system (unless a :CONTINUE command, defined later in this section, precedes the error); if it was entered in an interactive session, MPE/3000 returns control to the user at the terminal. Abort-error messages are described in Section X.

> *NOTE:* *Whenever a callable intrinsic is invoked by a process running in either the non-privileged mode, or the privileged mode with the DB register pointing to the DB area in the user's stack, a bounds check takes place to ensure that all parameters in the intrinsic call reference addresses that lie between the DL and S addresses in the stack (prior to the intrinsic call); if an address outside of these boundaries is referenced, an abort-error occurs.*
>
> *When a callable intrinsic is invoked by a process running in the privileged mode, and the DB register points to a data segment other than the user's stack segment, the results depend on the particular intrinsic. Most intrinsics immediately abort in this case. Others (indicated in Appendix C) 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. Any additional special actions taken by a particular intrinsic are described in the discussion of that intrinsic.*

## BATCH JOBS

The user can initiate and terminate batch jobs through any device that accepts serial input (such as a card reader, tape unit, or terminal used non-interactively) located at the computer site, or through a terminal used non-interactively and located remotely, as discussed below.

### Initiating Batch Jobs

A batch job consists of a set of cards (or card images) that begins with a :JOB command, contains additional MPE/3000 commands, user programs, and optional data, and terminates with an :EOJ command. The :JOB command initiates the job by establishing contact with MPE/3000. If it accepts this command, MPE/3000 begins job processing. When processing begins, commands are sequentially accepted from the job until an :EOJ command is encountered or until the job is halted by one of the abnormal events described under "Premature Job or Session Termination."

A job can be entered through any device configured to accept jobs. The user (or computer operator) first requests an interrupt on the job input device as follows:

1.   For card readers or magnetic tape units, the user turns the device on-line.

2.   For terminals, the user turns the terminal on, dials MPE/3000 (if the terminal is connected over switched telephone lines), and presses the *return* key (to generate a carriage return).

> *NOTE:    The device through which a job/session is entered is called the job/ session input device, and is recognized as the standard input file for the job/session. Each potential job/session input device is related, during system configuration, to a corresponding job/session listing device that is recognized as the standard output file for the job/ session (unless another device is designated by the user for this purpose). There is one job/session input device and one corresponding job/session listing device for each job/session.*

The user writes the :JOB command in the following format. (The keyword parameter groups, of course, can be specified in any order.)

```
:JOB    [jobname,] username [/upass] .acctname [/apass] [,groupname [/gpass] ]
        [;TERM=termtype]
        [;TIME=cpulimit]
        [;PRI=executionpriority]
        [;INPRI=selectionpriority]
        [;OUTCLASS=outputclass]
```

The parameters have the meanings noted below.

> NOTE:   In MPE/3000, names (such as the jobname, username, upass, acctname, apass, groupname, and gpass parameters noted below) consist of from one to eight alphanumeric characters, beginning with a letter, unless otherwise specifically indicated. Embedded blanks are not permitted within username.accountname specifications.

jobname
: An arbitrary name used in conjunction with the *username* and *acctname* parameters to reference this job in other commands. If omitted, a *null jobname* is assigned. (Optional parameter.)

> NOTE:   A fully-qualified jobname consists of [jobname,] username.acctname.

username
: The user's name, as established in MPE/3000 by the user with Account Manager Capability. This name is unique within the account. (Required parameter.)

upass
: The user's password. (Required if the user has been assigned a password by the user with Account Manager Capability.)

acctname
: The name of the user's account, as established by the user with System Manager Capability. Note that this parameter is preceded by a *period* as a delimiter. (Required parameter.)

apass
: The account password. (Required if the account has been assigned a password by the user with System Manager Capability.)

groupname
: The name of the group (established by the user with Account Manager Capability) that the user wishes to use, during his job, for his local file domain and for central processor time accumulation charges. If omitted, MPE/3000 assigns the user's home group. (Optional parameter for users with a home group; Required parameters for users without a home group.)

*gpass*      The group password. When a user logs on under his home group, no password is needed. (Required parameter for some groups.)

*termtype*   The type of terminal used for input.

(MPE/3000 uses the *termtype* parameter to determine device-dependent characteristics such as delay factors for carriage returns.) This parameter is indicated by one of the numbers 0 through 8, with the meanings shown below:

0 = ASR 33 EIA-compatible HP 2749B (10-characters per second (cps)).

0 = ASR 35 EIA-compatible (10 cps).

1 = ASR 37 EIA-compatible (15 cps).

3 = Execuport 300 Data Communication Transceiver Terminal (10, 15, 30 cps).

4 = HP 2600A or DATAPOINT 3300 (10-240 cps).

5 = Memorex 1240 (10, 30, 60 cps).

6 = GE Terminet 300 Data Communication Terminal Model B (10, 15, 30 cps).

7 = Selectric (IBM 2741) CALL 360 Correspondence Code (14.8 cps).

8 = Selectric (IBM 2741) (PTTC/EBCD Code) (14.8 cps).

*NOTE:   Users at IBM Selectric terminals are directed to the NOTE under the discussion "Initiating Sessions," later in this section.*

(Additional information on terminals appears in Section VIII.)

If the *termtype* parameter is omitted, and a terminal is used, a default of 0 is assigned. (Optional parameter for ASR 33 or ASR 35 terminals; Required parameter for all others, to ensure correct listings.)

| | |
|---|---|
| *cpulimit* | Maximum central-processor time permitted for the job, entered in seconds. If this limit is reached during the job, the job is aborted. If a question mark is entered, no limit applies. The maximum limit entry is 32767. If omitted, an installation-defined limit applies. (Optional Parameter.) |

> *NOTE:*    *The* cpulimit *parameter should not be confused with the central-processor time limit defined for groups and accounts. The* cpulimit *parameter (or its default value) applies to each individual job, and aborts the job as soon as it is exceeded. The central-processor time limit for groups and accounts, however, limits the total central-processor time used by all jobs that have run under the particular group and account--it is checked only at log-on time, and can only terminate access at that time; it will never cause a job in process to abort.*

| | |
|---|---|
| *executionpriority* | The priority (subqueue) desired for MPE/3000 command interpretation, and also the default priority for all programs within the job. For users with *standard capabilities*, this may be CS, DS, or ES, indicating the priorities (subqueues) described in Section II. For users with optional capabilities, this may be any of the priorities described in Part 3 (depending on the maximum subqueue priority permitted this user). If the priority specified exceeds that permitted for this user, the highest priority possible below BS applies. If this parameter is omitted, CS is assigned by default. (Optional Parameter.) |
| *selectionpriority* | The relative priority to be used for selecting this job when several jobs of equal *executionpriority* are ready to begin processing. This parameter is one or two digits, ranging from 0 (lowest priority) to 15 (highest priority). The default value is 8. Note that this parameter merely supplements other criteria used to determine final selection. Additionally, other factors being equal, candidates of equal priority are serviced on a first-in, first-out basis. (Optional Parameter.) |
| *outputclass* | A particular device (such as a specific line printer) to be used for listing output. This parameter is a device class name or a logical device number (as defined in Section V), and may only be entered by users who have the *non-sharable input/output device* file-access attribute. If this parameter is omitted, the device assigned is the one defined (during system configuration) as the default job/session listing device that always corresponds to the user's current input device. (Optional Parameter.) |

The *upass*, *apass*, and *gpass* parameters, when included, must be separated from their preceding parameters with a slash. (The slash should not be preceded nor followed by blanks.) When the job list device is not the job input device, passwords are not printed on the job listing.

3-13

*EXAMPLE:*

*This job is named ALPH22, and is entered under the user name RJOHNSON, account name ACCT1003, and the group GPA2. The group password MG03 is used. A central processor time limit of 300 seconds is entered. An execution priority of DS is requested. For all other parameters, default values apply. Notice the continuation character (&) at the end of the first line.*

```
:JOB  ALPH22,RJOHNSON.ACCT1003,  GPA2/MG03;  TIME=300;  &
:PRI=DS
```

If the central-processor time already accumulated by previous jobs exceeds the total time allotted for the user's account or group, the job is rejected upon submission. (Note that these account/group limits are checked only at the start of a job.) Otherwise, the job runs until completed or until aborted because of an error or because the limit designated by the *cpulimit* parameter in the :JOB command is reached.

All MPE/3000 commands encountered during job processing are listed on the job listing device (typically, a *printer*). Acceptance of the job is indicated by information output in the following format, immediately after the listing of the :JOB command.

*JOB NUMBER = #Jnnnnn*
*date, time*
*HP32000v.uu.ff*

 *nnnnn*  Job number assigned by MPE/3000 to uniquely identify the job to the system. This may range from one to five digits long. The user can reference the job in subsequent commands by this number or by the job identification specified through the :JOB command:

 *[jobname,] username.acctname*

 *date*  Current date (day-of-week, month and day, year)

 *time*  Current time (hours:minutes am/pm)

 *v*  MPE/3000 version level.

 *uu*  MPE/3000 update level.

 *ff*  MPE/3000 fix level.

*EXAMPLE:*

*If the job in the previous example had been assigned the job number 7, and had been submitted at 3:23 p.m. on January 31, 1972, the job output listing would appear as follows (with the beginning of the acceptance message shown here by an arrow):*

```
:JOB  ALPH22,RJOHNSON.ACCT1003,  GPA2/MG03;  TIME=300;  &
:PRI=DS
JOB  NUMBER  =  #J7
MON,JAN  31,  1972,  3:23  PM
HP32000B.02.08
```

## Terminating Batch Jobs

The :EOJ command terminates the batch job. This command has no parameters and is written simply as

*:EOJ*

When this command is encountered, MPE/3000 acknowledges job termination by printing the following information on the job listing

*CPU(SEC) = cputime*
*ELAPSED (MIN) = elapsedtime*
*date, time*
*END OF JOB*

| | |
|---|---|
| *cputime* | Total central processor time used by job, in seconds, charged to account and group. |
| *elapsedtime* | Total wall-clock time between the beginning and end-of-job, in minutes. This value is *not* charged to the account and group. |
| *date* | Current date (day-of-week, month and day, year) |
| *time* | Current time (hours:minutes am/pm) |

If no :EOJ command is included to terminate the current job, the next :JOB command will have one of the following effects:

1.  If read by the MPE/3000 Command Interpreter, it will terminate the current job and start a new one.

2.  If read by a user's program, it will signal an end-of-file to the program but will be ignored by MPE/3000. (Because this can occur, the user should be sure that he terminates his job with an :EOJ command.)

## Typical Job Structures

All batch jobs must begin with a :JOB command and terminate with an :EOJ command. Additionally, the end of each program input within a job should be indicated with an :EOD command, (written simply as :EOD). Beyond this, job structures can vary considerably from application to application.

*EXAMPLE:*

*A job submitted through the card reader to compile and execute a program might appear as follows. (Note that an :EOD card denoting the end of the program must precede the :EOJ card.)*
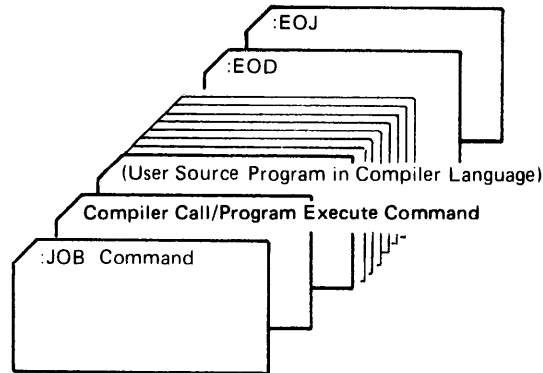


When data is included within a job, its end must also be indicated with an :EOD command. (Users writing language processors and other subsystems should be aware that MPE/3000, when reading input from the standard input stream, interprets *any* record beginning with a colon as the end-of-data, whether or not this record contains an :EOD command. The :EOD command is used as a data delimiter in general applications because it executes no *additional* functions.)

*EXAMPLE:*

*The job shown below includes data to be processed during program execution:*
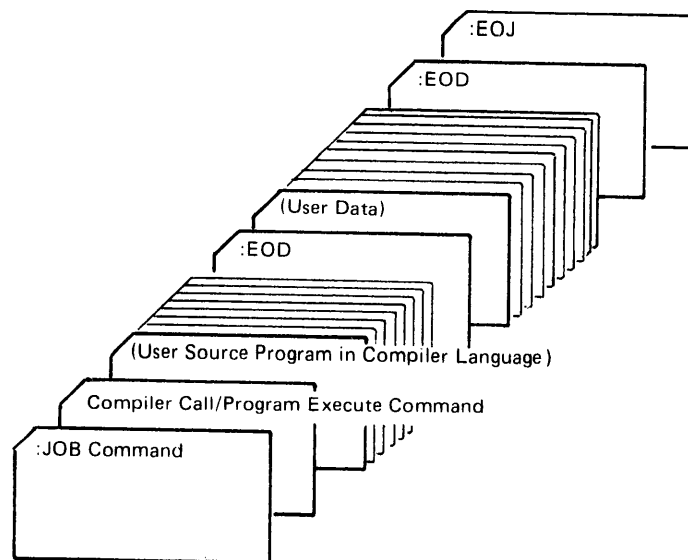
Of course, jobs with much more elaborate structures are possible. Such jobs may contain several user programs, input to different compilers, obtaining input data from various sources, and transmitting output data to several devices:

*EXAMPLE:*

*The following job contains: a FORTRAN/3000 program to be compiled and executed; data for that program; and an SPL/3000 program to be compiled and executed, with data input from a disc file.*
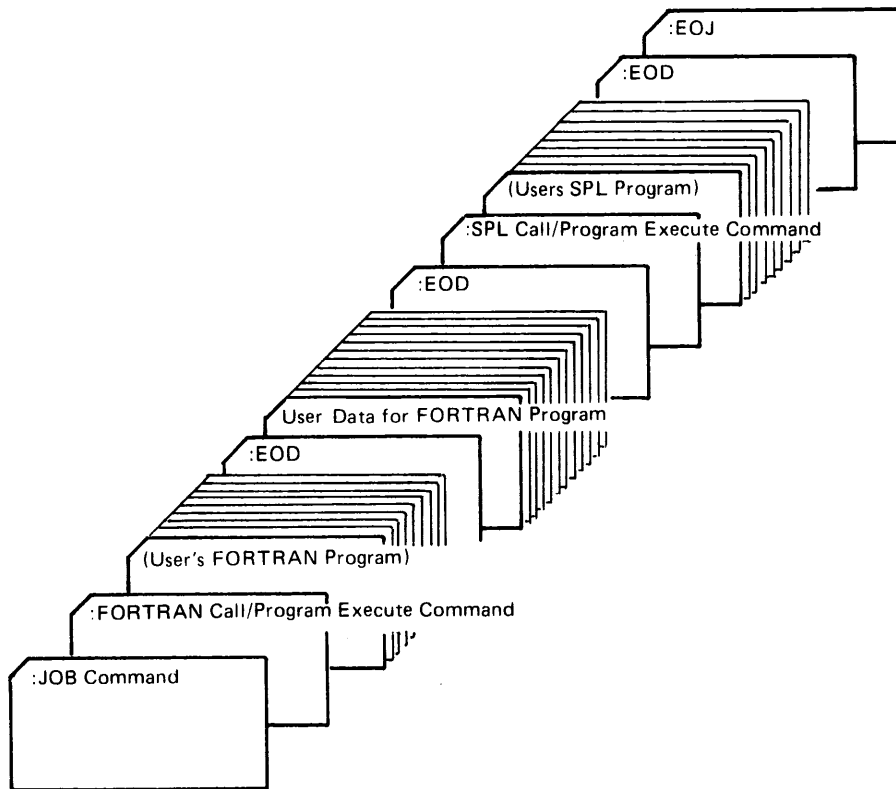
## INTERACTIVE SESSIONS

The user initiates and terminates interactive sessions from a terminal. (Non-interactive devices cannot be used for this purpose.)

### Initiating Sessions

The user initiates a session by turning the terminal on, dialing MPE/3000 (if the terminal is connected over switched telephone lines), pressing the *return* key to generate a carriage return and produce the first prompt character (colon), and entering the :HELLO command described below. This logging-on establishes contact with MPE/3000 and allows the user to enter commands to compile and run programs or request other MPE/3000 standard capabilities. From this point on, the user is automatically prompted for each command (and thus does not type the colon). The session exists until the user issues a command to terminate it, or until one of the conditions described under "Premature Job or Session Termination" occurs.

> *NOTE:* *Users at IBM 2741 Selectric Terminals (PTTC/EBCD Code*
> *or Call 360 Correspondence Code) must always type H or*
> *h as the first character of their input following the initial*
> *prompt character; this enables MPE/3000 to determine*
> *(from this character's bit pattern) the type of Selectric*
> *being used.*
>
> *Users at IBM 2741 Selectric Terminals (CALL/360 or*
> *PTTC/EBCD code) should note that any CALL/360 or*
> *PTTC/EBCD character that does not have an equivalent*
> *ASCII character is ignored on input.*
>
> *Users at IBM 2741 Selectric Terminals (Call 360 Corre-*
> *spondence Code) will receive a not-equals sign (≠) as*
> *the first prompt character.*

The user enters the :HELLO command in the following format

```
:HELLO [sessionname,] username[/upass] .acctname[/apass] [,groupname[,gpass] ]
      [;TERM = termtype]
      [;TIME = cpulimit]
      [;PRI = executionpriority]
      [;INPRI = selectionpriority]
```

The parameters have the meanings noted below.

> *NOTE:* *The sessionname, username, upass, acctname, apass,*
> *groupname and gpass parameters are all names that*
> *can contain up to eight alphanumeric characters,*
> *beginning with a letter.*

| | |
|---|---|
| *sessionname* | An arbitrary name used in conjunction with the *username* and *acctname* parameters to reference this session in other commands. If omitted, a null *sessionname* is assigned. (Optional parameter.) |

*username*
*upass*
*acctname*
*apass*
*groupname*
*gpass*
*termtype*
*executionpriority*
*selectionpriority*

The same definitions noted under the corresponding parameters in the :JOB command, applying to *sessions* rather than jobs.

| | |
|---|---|
| *cpulimit* | The same definition noted for *cpulimit* in the :JOB command, applying to *sessions*, and with this exception: if the *cpulimit* parameter is omitted, *no* limit applies. |

The *upass*, *apass*, and *gpass* parameters, when included, must always be separated from their preceding parameters by a slash. (This slash should not be bounded by blanks.)


*EXAMPLE:*

*In the following command, the user establishes a session named INTER3, under the user name JONES, account name ACT20, and GROUP 3. The account password of PASS is also entered.*

    :HELLO INTER3,JONES.ACT20/PASS, GROUP3


When a user is beginning a session through a terminal connected over switched telephone lines (a dial-up terminal), he must specifically log on within a period defined during system configuration. Otherwise, his terminal is disconnected and he must dial MPE/3000 again.

When entering a session through certain full duplex terminals, the user can suppress the printing (echoing) of passwords at his terminal by temporarily requesting half-duplex mode (by pressing the *ESC* and ; keys). (The user returns to full-duplex mode by pressing the ESC and : keys.) Where the terminal does not permit dynamic half-duplex operation, the user can conceal the passwords simply by omitting them from the command; in this case, after the :HELLO command is echoed, MPE/3000 outputs a corresponding request for each required password, directs the carriage to skip one line, prints an eight-character mask, and returns the carriage to the beginning of the line, as shown:

{ *USER*
  *ACCOUNT*    *PASSWORD?*
  *GROUP* }

**HHHHHHHH** ◄——— *(mask)*
↑——————————— *(carriage return)*

The user then enters the password. The password is echoed on the input/listing device, but is printed on top of the mask to ensure privacy.

*EXAMPLE:*

*Suppose that a group password of ORG was required, but not entered, in the command shown in the previous example. The user would be prompted for the password, and would enter it (over the mask) as follows:*

```
:HELLO INTER3,JONES.ACT20/PASS, GROUP3
GROUP PASSWORD?
▶  ⋈⋈⋈⋈⋈⋈⋈⋈
```

For sessions in which input and listing output are generated on separate devices, MPE/3000 omits the password entered.

Upon initiation of the session, the following information is transmitted to the terminal:

*SESSION NUMBER = #Snnnnn*
*date, time*
*HP32000v.uu.ff*

| | |
|---|---|
| *nnnnn* | Session number assigned by MPE/3000 to uniquely identify the session. This may range from one to five digits long. |
| *date* | Current date (day-of-week, month and day, year) |
| *time* | Current time (hours:minutes am/pm) |
| *v* | MPE/3000 version level. |
| *uu* | MPE/3000 update level. |
| *ff* | MPE/3000 fix level. |

*EXAMPLE:*

*Suppose that the session in the previous example is accepted by MPE/3000 under the session number 512 at 12:37 p.m. on May 5, 1972. The following information would be output. (Remember, the mask is printed before the password may be entered.)*

```
:HELLO  INTER3,JONES.ACT20/PASS,  GROUP3
GROUP  PASSWORD?
MMMMMMM
SESSION  NUMBER  =  #S512
FRI,MAY  5,  1972,   12:37  PM
HP32000B.02.08
```

If the connect time or the central processor time limit allotted to the account or group specified by the :HELLO command has been exceeded by previous sessions or jobs, the current session is rejected at log-on. If these limits expire during the session, however, the session is allowed to continue until terminated by the user; the next session under this account or group, however, will be rejected at log-on.

If the user enters an illegitimate :HELLO command, the following error message appears:

*:ERR 8*

At this point, a request to re-enter the :HELLO command appears. (The erroneous element in the previous :HELLO command is not identified.)

> *NOTE:* *A system power or line failure automatically results in the permanent termination of sessions; the sessions must be reinitiated from the beginning.*

**Interrupting Program Execution Within Sessions**

The user can interrupt execution of a program or subsystem at any point in a session by striking the BREAK key. (However, the BREAK key does not interrupt execution of MPE/3000 commands.)

This returns control to the MPE/3000 Command Interpreter, allowing the user to issue commands to abort the program (without disrupting the session), perform various file or utility operations, or resume the program.

To abort the program, the user enters this command:

*:ABORT*

To request an operation (such as creating or deleting a file), the user enters the command for that operation. File and utility commands are executed immediately. Some commands, however, require aborting of the user's program before they can be executed. When the user enters such a command during a program break, MPE/3000 prints:

*ABORT?*

The user responds by entering *YES* (to abort the current program and execute the command) or *NO* (to return control to the Command Interpreter).

To resume a program at the point where it was interrupted, the user enters:

*:RESUME*

If a read operation was pending for the program on the terminal when the break occurred, the following message is output:

*READ PENDING*

The user must satisfy the pending read before program execution can resume. (All characters entered before the break are retained.)

> NOTE:   The *:ABORT* and *:RESUME* commands are legitimate only during a break.


## Terminating Sessions

To terminate a session, the user enters the following command:

*:BYE*

MPE/3000 acknowledges termination by printing the following information:

*CPU (SEC) = cputime*
*CONNECT (MIN) = connecttime*
*date, time*
*END OF SESSION*

| | | |
|---|---|---|
| *cputime* | = | Total central-processor time used by the session, in seconds, logged against group and account. |
| *connecttime* | | Total wall clock time between log-on and log-off, in minutes, logged against group and account. |
| *date* | | Current date (day-of-week, month and day, year) |
| *time* | | Current time (hours:minutes am/pm) |

*EXAMPLE:*

*Typical terminal output resulting from logging off:*

```
:BYE

CPU (SEC)=4
CONNECT (MIN)=2
THU,MAY 7, 1973,  12:56 PM
END OF SESSION
```

For accounting purposes, the connect time is rounded upward to the nearest minute, and the central processor time is shown in seconds. Both sums are added to the usage counters for the session's account and group.

If the user hangs up the receiver at a dial-up terminal, an implicit :BYE command is issued and the session terminates.


## Typical Session Structure

All sessions must begin with a :HELLO command. They typically terminate with a :BYE command. The end of data within a session must be indicated with the :EOD command.

If a user issues a second :HELLO command within his current session, this results in an implicit :BYE and :HELLO sequence - that is, it terminates the current session and starts another one.

The structure of sessions varies with the application. One example follows.


*EXAMPLE:*

:HELLO *(Session, user, and account identification.)*

SESSION NUMBER = #Snnn
*(date) (time)*
HP3200v.uu.ff

:*(MPE/3000 Command to call BASIC Interpreter.)*
.
.
.
*(User Program in BASIC.)*

*(User Command to exit from BASIC Interpreter.)*

:BYE

CPU (SEC) = cputime
CONNECT(MIN) = connecttime
date, time
END OF SESSION

# READING DATA FROM OUTSIDE STANDARD INPUT STREAM

Users can designate, for a job or session, input consisting only of data to be read from a non-sharable, auto-recognizing device that is *not* the standard input device for that job or session. (An auto-recognizing device is one that automatically accepts the :JOB, :HELLO, or :DATA commands.) This designation is typically made to read a deck from a card reader while in session mode. Indeed, it is required for every data deck not imbedded in the standard input stream ($STDIN).

To designate the data for the job or session, the user must precede the data with the :DATA command and terminate it with the :EOD command. The :DATA command implicitly initiates communication with MPE/3000 and thus is the only command that is not entered within a formally-initiated job or session. (Note that the :DATA command does not apply to data from non-auto recognizing devices, or data imbedded in the job or session input stream.)

The :DATA command format is

> *:DATA [jobname,] username[/upass] .acctname[/apass] [;filename]*

| | | |
|---|---|---|
| | *jobname* | The name of the job or session that is to read the data. (Optional parameter.) |
| (A complete job or session identi-fication.) | *username* | The user's name, as established in MPE/3000 by the user with Account Manager Capability. (Required parameter.) |
| | *upass* | The user password.(Required if user has a password.) |
| | *acctname* | The name of the account, as established by the user with System Manager Capability. (Required parameter.) |
| | *apass* | The account password.(Required if account has a password.) |

*filename*   An additional qualifying name for the data that can be used by the job or session to access the data. It may be used, for example, to distinguish two separate data decks from different card readers read by the same program. If *filename* is omitted, no such distinguishing name is assigned. (Optional parameters.)

The data can *only* be read by a job or session that has the same identity:

> *[jobname,] username.acctname*

The data exists in the system until read by the job/session.

If the *filename* parameter is omitted from the :DATA command, then the data can be read by *any* access from the job with the corresponding identity.

If the job attempts to read data but omits the file *formaldesignator* when opening the data file, any file preceded by a :DATA command referencing that job's identity will satisfy the request.

The *jobname, username, acctname,* and *filename* parameters are all names that can contain up to eight alphanumeric characters, beginning with a letter.

*EXAMPLE:*

*To designate a card data file for a session identified as JOBSP, BLACK.ACTSP, the user submits the following :DATA command on a card preceding the data deck:*

```
:DATA JOBSP,BLACK.ACTSP; FILESP
      •
      •
      •
(USER DATA)
      •
      •
      •
:EOD
```

*The session can access the data in the card file by specifying the card reader (either by device class name or logical device number, as shown in Section V), and the filename FILESP.*


## PREMATURE JOB OR SESSION TERMINATION

A *job* is terminated before an :EOJ command is encountered, if any of the following events occur:

1.   The Console Operator issues a command to terminate the job.

2.   MPE/3000 aborts the job because an error occurred in the interpretation or execution of an MPE/3000 command.

3.   MPE/3000 aborts the job because a subsystem error was encountered.

4.   A program within the job is aborted.

5.   A second :JOB command or a :DATA command is encountered. It will be executed, resulting in job termination (unless it is read as part of a program's data input from a file/device, in which case it signals an end-of-file to the program but is ignored by MPE/3000).

6.   The central processor time limit specified in the *cpulimit* parameter of the :JOB command (or its default value) was exceeded.

Events 2 through 4 (above) place the job in an *error state* (by setting the error state flag). Normally, this results in job termination. But, if the user anticipates an error as a result of a specific command, he can override premature termination by using the :CONTINUE command before that command. The :CONTINUE command format is

   *:CONTINUE*

The :CONTINUE command permits the job to continue even though the following command results in an error (in which case the error state indication is re-set).

*EXAMPLE:*

*Suppose the user submits a job to execute three programs (with the :RUN command), but anticipates a probable terminating error in the first program. To continue the job and execute the second and third programs, he uses the :CONTINUE command as follows:*

```
:JOB  JNAM,UNAM.ACCTNAM
:CONTINUE
:RUN  PROG1
:RUN  PROG2
:RUN  PROG3
:EOJ
```

A session may be terminated before a :BYE command is encountered, if any of the following events occur:

1.   The Console Operator issues a command to terminate the session.

2.   A second :HELLO command, or a :JOB or :DATA command, is encountered in the session input stream.  This will implicitly result in a :BYE command, terminating the current session.

3.   The central processor time limit specified in the *cpulimit* parameter of the :HELLO command was exceeded.

# SECTION IV
# Compiling, Interpreting,
# Preparing, and Executing Programs

After a user initiates a batch job or session, he can enter commands to compile, prepare, and execute user programs or to access various MPE/3000 subsystems. Most of these commands require references to files used for input and output. For instance, a command to compile a program references a file that contains source program input, another used for program listing output, and a third used for object program output. Because of the importance of file references as command parameters, some of the rules for specifying files are introduced before the compilation, preparation, and execution commands themselves. The complete rules concerning files are discussed in Sections V and VI.

## REFERENCING FILES

When compiling, preparing, or executing programs, the user can designate the files to be used for input and output in either of two ways:

1.  By naming the files as positional parameters (called *actual file designators*) in the compilation, preparation, or execution command.

2.  By omitting optional parameters from the compilation, preparation, or execution command, allowing MPE/3000 to assign standard default files.

### Specifying Files as Command Parameters

The user can name the following types of files as parameters in a compilation, preparation, or execution command.

- System-Defined Files

- User Pre-Defined Files

- New Files

- Old Files

**SYSTEM-DEFINED FILES.** System-defined file designators indicate those files that MPE/3000 uniquely identifies as standard input/output devices for a job/session. They are referenced by the following actual file designators:

| Actual File Designator | Device/File Referenced |
|---|---|
| $STDIN | A filename indicating the standard job or session input device (that from which the job or session is initiated). For a job, this is typically a card reader. For a session, this is typically a terminal. Input data images in the $STDIN file should not contain a colon in Column 1, since this indicates the end-of-data. (When data is to be delimited, this is properly done through the :EOD command, which executes no other functions.) |
| $STDINX | Equivalent to $STDIN, except that MPE/3000 command images (those with a colon in Column 1) encountered in a data file, are read without indicating the end of data. (However, the commands :JOB, :DATA, :EOJ, and :EOD are exceptions that always indicate the end-of-data and are *never* read as data.) Furthermore, any requests with a byte count less than 5 indicate the end-of-data in this file, if there is a colon in Column 1. |
| $STDLIST | A filename indicating the standard job or session listing device. The job or session listing device is customarily a printer for a batch job and a terminal for a session. |
| $NULL | The name of a non-existent "ghost" file that is always treated as an empty file. When referenced as an input file by a program, that program receives an end-of-data indication upon each access. When referenced as an output file, the associated write request is accepted by MPE/3000 but no physical output is actually performed. Thus, $NULL can be used to discard unneeded output from a running program. |

**USER PRE-DEFINED FILES.** A user pre-defined file is any file that was previously defined or redefined in a :FILE command, as discussed in Section V. In other words, it is a back-reference to that :FILE command. In compilation, preparation, or execution commands, the actual file designator of this file is written as

*formaldesignator*

> *formaldesignator* = The name used in the *formaldesignator* parameter of the :FILE command (Section V).

**NEW FILES.** New files are files that have not yet been created, and are being created/opened for the first time by the current batch job or interactive session. New files can have the following actual file designators:

| Actual File Designator | File Referenced |
| --- | --- |
| $NEWPASS | A temporary disc file that can be automatically passed to any succeeding MPE/3000 command within the same job/session, which references it by the filename $OLDPASS. (Passing is explained in later examples.) Only one such file with this designation can exist in the job/session at any one time. (When $NEWPASS is closed, its name is automatically changed to $OLDPASS, and any previous file named $OLDPASS in the job/session is deleted.) |
| *filereference* | Any other new file to which the user has access. Unless the user specifies otherwise, this is a temporary file, residing on disc, that is destroyed upon termination of the program. If closed as a *job/session temporary file*, as shown in Section V, it is saved until the end of the job/session, when it is purged. If closed as a permanent file, it is saved until purged by the user. Typically, this format consists of a file name containing up to eight alphanumeric characters, beginning with a letter. In addition, other elements (such as a group name, account name or lockword) can be specified. The complete rules governing the *filereference* format are presented in Section V. |

**OLD FILES.** Old files are existing files currently-resident in the system. They may be named by the following designators:

| Actual File Designator | File Referenced |
| --- | --- |
| $OLDPASS | The name of the temporary file last closed as $NEWPASS. |
| *filereference* | Any other old file to which the user has access. It may be a job/session temporary file created in this or a previous program in the current job/session, a permanent file saved by any program in any job/session, or a permanent file built (with the :BUILD command) in any job/session. The format is the same as *filereference*, noted above and defined in Section V. |

**INPUT/OUTPUT SETS.** All of the preceding actual file designators can be classified as those used as input parameters *(Input Set)* and those used as output parameters *(Output Set)*. These sets, referred to frequently throughout this manual, are defined as follows:

**Input Set**

| | |
|---|---|
| $STDIN | The job/session input device. |
| $STDINX | The job/session input device with commands allowed. |
| $OLDPASS | The last $NEWPASS file closed. |
| $NULL | A constantly-empty file that will produce an end-of-file indication whenever it is read. |
| *formaldesignator* | A back-reference to a previously-defined file. |
| *filereference* | A file name, and perhaps account and group names and a lockword. |

**Output Set**

| | |
|---|---|
| $STDLIST | The job/session listing device. |
| $OLDPASS | The last file passed. |
| $NEWPASS | A new temporary file to be passed. |
| $NULL | A constantly-empty file. |
| *formaldesignator* | A back-reference to a previously-defined file. |
| *filereference* | A file name, and perhaps account and group names and a lockword. |

Further information on how these files are specified is presented under "Implicit :FILE Commands" in Section V.

**Specifying Files By Default**

When a user omits an optional file parameter from a compilation, preparation, or execution command, the subsystem invoked uses one of the members of the input or output set by default. The file designator assigned depends on the specific command, parameter, and operating mode, as noted later in this section.

4-4

## USING THE BASIC/3000 INTERPRETER

The BASIC/3000 Interpreter is generally used for on-line programming during sessions. However, it can also be used to interpret BASIC/3000 programs submitted in the batch-job mode. In either case, the user calls the BASIC/3000 Interpreter with a command of the following format.

   :BASIC [commandfile] [,[inputfile] [,listfile] ]

| | |
|---|---|
| *commandfile* | Source file (device) from which BASIC commands and statements are input. This can be any ASCII file from the input set. (The designator *BASCOM*, however, must *not* be used.) If omitted, the default file $STDINX is assigned; (in sessions, this is typically the terminal; in batch jobs, it is usually the card reader). (Optional parameter.) |
| *inputfile* | Source file containing data input to BASIC program. This can be any ASCII file from the input set. (The designator *BASIN*, however, must *not* be used.) If omitted, the default file $STDINX is assigned; (for sessions, this is typically the terminal; in jobs, it is typically the card reader). (Optional parameter.) |
| *listfile* | Destination file for BASIC program listing and output. Can be any ASCII file from output set. (The designator *BASLIST*, however, must *not* be used.) If omitted, the default value $STDLIST (typically, the terminal for sessions or the printer for jobs) is assigned. (Optional parameter.) |

*EXAMPLES:*

*The following :BASIC command, entered during a session, specifies (by default) a terminal as the source for BASIC commands and input data, and the destination of output data generated by the user's program. (This is the application of the :BASIC command used most commonly.)*

### :BASIC

*In special cases, the programmer may not use the terminal for input/output. The next :BASIC command, encountered during a batch job input through the card reader specifies the card reader (default input device) as the source of BASIC commands, the user-created disc file DATABANK as the source of input data, and the line printer (default list device) as the destination of output data. (Notice that the omission of the first parameter is indicated by beginning the parameter list with a comma.)*

### :BASIC , DATABANK

When the BASIC/3000 Interpreter has been entered, a greater-than sign (>) is used as the prompt character. This is returned by the terminal during sessions or entered by the user during jobs.

## COMPILING/PREPARING/EXECUTING PROGRAMS

User source programs written in compiler languages undergo the operational steps outlined below. In most cases, the details of these steps will be invisible to the user during normal compilation and execution. When necessary, however, the user can advance through each of these steps independently, completely controlling the specifics of each event along the way.

1. The source language program is *compiled* (translated by a compiler) into binary form and stored as one or more relocatable binary modules (RBM's) in a specially-formatted disc file called a user subprogram library (USL). There is one RBM for each program unit. (A program unit is a self-contained set of statements that is the smallest divisible part of a program or subprogram. In FORTRAN/3000, this can be a main program, subroutine, function, or block data unit. In SPL/3000, it can be an outer block, or procedure. In COBOL/3000, it can be a main program, subroutine, or section.) In USL form, however, the program is not yet executable.

2. The USL is then *prepared* for execution by the MPE/3000 Segmenter. The Segmenter binds the RBM's from the USL into linked, re-entrant code segments organized on a program file. (In preparation, only *one* USL can be used for RBM input to the program file.) At this point, the special segment for the input of user data (the stack) is also initially defined.

3. The program file is *allocated and executed.* In allocation, the segments from the program file are bound to referenced external segments from segmented libraries (SL's). Then the first code segment to be executed, and the stack are moved into main memory and execution begins.

Over a period of time, the user can produce RBM's on the same USL through several compilations. If there is only one main program or outer block RBM containing active entry points, this USL can be prepared onto a program file and run. This allows the user to compile his main program and procedures separately.

During compilation, if a program unit is designated as privileged, the corresponding RBM is flagged as privileged in the USL.

During preparation, the USL and program files are opened as job temporary files. For both files, the Segmenter first searches for a file of the proper name that exists as a job temporary file; if such a file cannot be found, the Segmenter searches for a permanent file of the appropriate name. If no program file of the referenced name exists, the Segmenter creates a new program file that is saved in the job temporary file domain, and prepares into this. If any RBM from the USL is flagged as privileged, the user must have the privileged mode (PM) optional capability to prepare it; once prepared, the entire segment containing the privileged RBM is flagged as privileged. During preparation, the capability-class attributes of the program are also determined (by the user or by the default option).

A particular program can be run by many user processes at the same time, because code in a program is inherently sharable.

Before allocation/execution is initiated, MPE/3000 checks to verify the following points:

a. *For program files that are permanent files*, the capabilities assigned to the program must not exceed those assigned to the group to which the program file belongs; otherwise, an error occurs.

b. *For program files that are temporary files in the job/session domain*, the capabilities assigned to the program must not exceed those of the user running the program; otherwise, an error occurs.

c. *For privileged segments, when the NOPRIV parameter is omitted from the :RUN (program execution) command*, the capabilities assigned to the program must include the privileged mode capability for the segment to be loaded in privileged mode; otherwise, an error occurs.

The compilation, preparation, and execution of a program can be requested by individual commands or by a single command that performs all three operations. In the following pages, all of these commands are described.


## Compilation Only

To compile a source-language program, the user enters a command of the following format:

*:compiler [textfile] [,[uslfile] [,[listfile] [,[masterfile] [,newfile] ] ] ]*

| | |
|---|---|
| *compiler* | FORTRAN for the FORTRAN/3000 compiler; SPL for the SPL/3000 compiler; or COBOL for the COBOL/3000 compiler. |
| *textfile* | Input file (device) from which the source program is to be read. This can be any ASCII file from the input set. (The following designators, however, must *not* be used: *SPLTEXT* (SPL/3000), *FTNTEXT* (FORTRAN/3000), or *COBTEXT* (COBOL/3000).) If omitted, the default file $STDIN (the current input device) is used. (Optional parameter.) |
| *uslfile* | The name of the USL file on which the object program is written. This can be any binary file from the output set. (The following designators, however, must *not* be used: *SPLUSL* (SPL/3000), *FTNUSL* (FORTRAN/3000), or *COBUSL* (COBOL/3000).) If the *uslfile* parameter is entered, it must indicate a file previously created by the user in one of three ways: |

        1. By saving a USL file (through the :SAVE command, discussed in Section V) created by a previous compilation where the default value was used for the *uslfile* parameter.

2. By building the USL (through the segmenter subsystem command — BUILDUSL, discussed in Section VII.)

3. By creating a new file of USL type (through the :BUILD command discussed in Section V, with the decimal code of 1024 or the mnemonic *USL* used for the *filecode* parameter).

If the *uslfile* parameter is omitted, the default file $OLDPASS is assigned, unless there is no passed file (in which case, $NEWPASS is used). (Optional parameter.)

*listfile*          The name of the file to which the program listing is to be generated. This can be any ASCII file from the output set. (The following designators, however, must *not* be used: *SPLLIST* (SPL/3000), *FTNLIST* (FORTRAN/3000), or *COBLIST* (COBOL/3000).) If omitted, the default file $STDLIST (typically the terminal in a session or the printer in a batch job) is assigned. (Optional parameter.)

*masterfile*        The name of a file to be optionally merged with *textfile* and written onto a file named *newfile*, as discussed in the manuals covering compilers. (The following designators, however, must *not* be used: *SPLMAST* (SPL/3000), *FTNMAST* (FORTRAN/3000), or *COBMAST* (COBOL/3000).) If *masterfile* is omitted, no merging takes place. (Optional parameter.)

*newfile*           The file on which the (re-sequenced) records from *textfile* (and *masterfile*) are placed. (The following designators, however, must *not* be used: *SPLNEW* (SPL/3000), *FTNNEW* (FORTRAN/3000), or *COBNEW* (COBOL/3000).) When *newfile* is omitted, no new file is created. (Optional parameter.)

*EXAMPLES:*

*The following command compiles a FORTRAN/3000 source program entered by a user through the job/session input device, into an object program in the USL file $NEWPASS. It also transmits listing output to the job/session list device. (If the next command is one to prepare an object program, $NEWPASS can be passed to it as an input file. Note that a file can only be passed between commands or programs within the same job or session.)*

**:FORTRAN**

*The following example illustrates the passing of a file between two successive compilations. In this example, the :FORTRAN command compiles a program into the USL file $NEWPASS (because the default value for uslfile is taken, and no passed file presently exists). The :SPL command compiles another program into the same USL, passed to this command under the redesignation $OLDPASS. (When no passed file exists, $NEWPASS is used; when a passed file exists, that file ($OLDPASS) is used.) After the second compilation, $OLDPASS contains RBM's from both compilations:*

```
:JOB MYNAME.MYACCT
:FORTRAN

        (FORTRAN SOURCE PROGRAM)
                 •
                 •
                 •
:EOD
:SPL
                 •
                 •
                 •
        (SPL SOURCE PROGRAM)
:EOD
                 •
                 •
                 •
:EOJ
```

*The following command compiles a COBOL/3000 source program residing on the disc file SOURCE into an object program on the USL file OBJECT, with a program listing generated on the disc file LISTFL.*

```
:COBOL SOURCE, OBJECT, LISTFL
```

## Compilation/Preparation

To compile and prepare for execution a source-language program, the user enters a command in the following format. (The USL file created during compilation is a temporary file passed directly to the preparation mechanism; it is accessible to the user as $OLDPASS only if the *progfile* is not $NEWPASS.)

*:compiler [textfile] [,[progfile] [,[listfile] [,[masterfile] [,newfile] ] ] ]*

*compiler*          FORTPREP for the FORTRAN/3000 compiler; SPLPREP for the SPL/3000 compiler; or :COBOLPREP for the COBOL/3000 compiler.

*textfile*    Input file (device) from which the source program is read. This can be any ASCII file from the input set. (The following designators, however, must *not* be used: *SPLTEXT* (SPL/3000), *FTNTEXT* (FORTRAN/3000), or *COBTEXT* (COBOL/3000).) If omitted, the default file $STDIN (the current input device) is used. (Optional parameter.)

*progfile*    The name of the program file onto which the prepared program segments are to be written. This can be any binary file from the output set. The user must create this program file (unless the default $NEWPASS is taken). He does this in one of two ways:

    1.  By creating a new file of program-file type (through the :BUILD command discussed in Section V, with the decimal code of 1029 or the mnemonic *PROG* used for the *filecode* parameter).

    2.  By specifying a non-existent file in the *progfile* parameter of the :FORTPREP, :SPLPREP, or :COBOLPREP command, in which case a file of the correct size and type is created.

If omitted, the default file $NEWPASS is assigned. (Optional parameter.)

*listfile*    The name of the file to which the program listing is written. This can by any ASCII file from the output set. (The following designators, however, must *not* be used: *SPLLIST* (SPL/3000), *FTNLIST* (FORTRAN/3000), or *COBLIST* (COBOL/3000).) If omitted, the default file assigned is $STDLIST (usually the terminal in a session or the printer in a batch job). (Optional parameter.)

*masterfile*
*newfile*    } The same meanings described under *compilation.*

NOTE:   *This command is equivalent to a compilation command followed by a :PREP command (as described later in this section). Thus, for FORTRAN/3000, the command*

     *:FORTPREP*   *[textfile]*
                         *[,[progfile]*
                         *[,[listfile]*
                         *[,[masterfile]*
                         *[,newfile] ] ]*

*is equivalent to*

     *:FORTRAN*   *[textfile]*
                         *,$NEWPASS*
                         *,listfile*
                         *,masterfile*
                         *,newfile*
     *:PREP*       *$OLDPASS*
                         *,progfile*

*EXAMPLES:*

*The following command compiles and prepares an SPL/3000 program entered through the job/session input device. The resulting program file is named $NEWPASS, and the program listing is printed on the job/session list device. (If the next command is one to execute the program, the file $NEWPASS is referenced in the execute command under the name $OLDPASS.)*

    <u>:SPLPREP</u>

*The next command compiles and prepares a FORTRAN/3000 source program input from a source file named SFILE into a program file named $NEWPASS. The resulting program listing is printed on the job/session listing device.*

    <u>:FORTPREP</u> SFILE

## Compilation/Preparation/Execution

To compile, prepare, and execute a program, a command of the following format is entered. (This command creates a temporary USL file that is not accessible by the user, and a program file available to the user as $OLDPASS.)

    *:compiler [textfile] [,[listfile] [,[masterfile] [,newfile] ] ] ]*

| | |
|---|---|
| *compiler* | FORTGO for the FORTRAN/3000 compiler; SPLGO for the SPL/3000 compiler; or :COBOLGO for the COBOL/3000 compiler. |
| *textfile* | Input file (device) from which source program is read. This can be any ASCII file from the input set. (The following designators, however, must *not* be used: *SPLTEXT* (SPL/3000), *FTNTEXT* (FORTRAN/3000), or *COBTEXT* (COBOL/3000).) If omitted, the default file $STDIN (the current input device) is assigned. (Optional parameter.) |
| *listfile* | The name of the file to which the program listing is transmitted. This can be any ASCII file from the output set. (The following designators, however, must *not* be used: *SPLLIST* (SPL/3000), *FTNLIST* (FORTRAN/3000), or *COBLIST* (COBOL/3000).) If omitted, the default file assigned is $STDLIST (normally the terminal for sessions or printer for batch jobs). (Optional parameter.) |
| *masterfile* ⎫<br>*newfile* ⎭ | The same meanings described under *compilation*. |

*NOTE:* This command is equivalent to a compilation command, followed by a :PREP command, followed by a :RUN command (as described later in this section). Thus, for FORTRAN/3000, the command

    _:FORTGO     [textfile]
                 [,[listfile]
                 [,[masterfile]
                 [,newfile]]]]

is equivalent to

    _:FORTRAN    textfile,
                 $NEWPASS
                 ,listfile
                 ,masterfile
                 ,newfile

    _:PREP $OLDPASS, $NEWPASS

    _:RUN $OLDPASS

*EXAMPLES:*

The command shown below compiles, prepares, and executes a COBOL/3000 program entered through the job/session input device. The program listing is printed on the job/session list device.

    **:COBOLGO**

The following command compiles, prepares, and executes an SPL/3000 program residing in the disc file SOURCE and transmits the program listing to the job/session list device.

    **:SPLGO SOURCE**

## Preparation Only

If a user's source program has been compiled onto a USL file, he can prepare it for execution with the :PREP command. (When writing this command, recall that keyword parameters can appear in any order after the positional parameters.)

    _:PREP uslfile,progfile

        [;ZERODB]
        [;PMAP]
        [;MAXDATA=segsize]
        [;STACK=stacksize]
        [;DL=dlsize]
        [;CAP=caplist]
        [;RL=filename]

*uslfile*        The name of the USL file (from the input set) on which the program
                 has been compiled. (Required parameter.)

*progfile*       The name of the program file onto which the prepared program segments
                 are to be written. This can be any binary file from the output set. The
                 user must create this program file. He does this in one of two ways:

                 1.   By creating a new file of program-file type (through the :BUILD
                      command discussed in Section V, with the decimal code of
                      1029 or the mnemonic *PROG* used for the *filecode* parameter).

                 2.   By specifying a non-existent file in the *progfile* parameter of
                      the :PREP command (or in the *progfile* parameter of the
                      segmenter subsystem command — PREPARE , discussed in
                      Section VII), in which case a file of the correct size and type
                      is created. (Required parameter.)

*ZERODB*         An indication that the initially-defined user-managed (DL-DB) area, and
                 uninitialized portions of the DB-Q (initial) area, will be initialized to
                 zero. If this parameter is omitted, these areas are not affected.
                 (Optional parameter.)

*PMAP*           An indication that a descriptive listing of the prepared program will be
                 produced on the file whose formal designator is SEGLIST; if no :FILE
                 command referencing SEGLIST is encountered, the listing is produced on
                 the job/session listing device. If this parameter is omitted, the listing is
                 not produced. (Optional parameter.)

*segsize*        Maximum stack area (Z–DL) size permitted, in *words*. This parameter
                 is included if the user expects to change the size of the DL–DB or
                 Z–DB areas during process execution. If omitted, MPE/3000 assumes
                 that he will not change these areas. (Optional parameter.)

*stacksize*      The size of the user's initial local data area (Z-Q (initial)) in the stack,
                 in *words*. (This value *must* exceed 511 words.) This overrides the
                 *stacksize* estimated by the segmenter, which applies if the stacksize
                 parameter is omitted. (The default is a function of estimated stack
                 requirements for each program unit in the program. Since it is difficult
                 for the system to predict the behavior of the stack at run time, the user
                 may want to override the default by supplying his own estimate with
                 *stacksize*.) (Optional parameter.)

*dlsize*         The DL-DB area to be initially assigned to the stack. This area is mainly
                 of interest only in programmatic applications, and is discussed in detail
                 in Section II. If the *dlsize* parameter is omitted, a value estimated by
                 the segmenter applies. (Optional parameter.)

| caplist | The *capability-class attributes* associated with the user's program; specified as two character mnemonics. If more than one mnemonic is specified, each must be separated from its neighbor by a comma. The mnemonics are |

IA  = Interactive access  }
BA  = Local batch access  } Standard Capabilities
PH  = Process Handling
DS  = Data Segment Management
MR  = Multiple resource management
PM  = Privileged-mode operation

The user who issues the :PREP command can only specify capabilities that he himself possesses (through assignment by the System or Account Manager). If the user does not specify any capabilities, the IA and BA capabilities (if possessed by the user) will be assigned to this program. (Optional parameter.)

| *filename* | The name of a relocatable procedure library (RL) file to be searched to satisfy external references during preparation as defined in Section VII. This can be any binary permanent file of type RL, as described in Section VII. It need not belong to the log-on group, nor does it have a reserved, local name. This file yields a single segment that is incorporated into the segments of the program file. If *filename* is omitted, no library is searched. (Optional parameter.) |

For the *stacksize, dlsize,* and *maxdata* parameters, a value of −1 indicates that the Segmenter is to assign the default value; it is equivalent to omitting the parameter.

*EXAMPLES:*

*The following command prepares a program from the USL file named USEFILE and stores it in a program file named PROGFILE. The optional parameters will be those assigned by default. The program's capability-class attributes will be the standard capabilities of the preparing user.*

   **:PREP USEFILE, PROGFILE**


*The next command will accomplish the same function as the previous command, except that the prepared program will be listed, a stacksize of 500 words will be established, and the program will be assigned only the batch-access capability.*


   **:PREP USEFILE, PROGFILE; PMAP; STACK=500; CAP=BA**

An example of the listing of the prepared program, requested by the PMAP parameter of the :PREP (and :PREPRUN) command, is shown in Figure 4-1. On this listing, significant entries are indicated with arrows, keyed to the discussion below.

*NOTE:* *All numbers in the listing except Item 13 (elapsed time) and Item 20 (central processor time) are octal values.*

| Item No. | Meaning |
|---|---|
| 1 | The name of the program file (filename.groupname.accountname). |
| 2 | The segment name. |
| 3 | The (logical) segment number. |
| 4 | The program unit entry-point name or external procedure name. |
| 5 | The assigned entry number in the Segment Transfer Table (STT). |
| 6 | The beginning location of the procedure code in the segment. |
| 7 | The location of the entry point in this segment. |
| 8 | The (logical) segment number of the segment containing this *external* procedure. If this entry is a number, then the procedure is external to the segment but internal to the program file; if it contains a question mark (?), then the procedure is external to the segment *and* external to the program file. |
| 9 | The segment length (in words). |
| 10 | The primary DB area size. |
| 11 | The secondary DB area size. |
| 12 | The total DB area size. |
| 13 | The time elapsed during preparation process. |
| 14 | The initial stack size. |
| 15 | The initial DL size. |
| 16 | The maximum area available for data (maximum Z-DL size). |
| 17 | Capability of program file. |
| 18 | Total code in file. |
| 19 | Total records in file. |
| 20 | Total central processor time used during preparation process. |

```
PROGRAM FILE SCR4.MPE.SYS        (1)  (3)
SEG40 (2)                         0

     NAME           STT  CODE ENTRY SEG
     LISTRL (4)      1 (5)  0    32
     MAKER(X)MINDL   30              10 (8)
     FGETINFO        31    (6)  (7)  ?
     FREADMR         32              10
     NTOA            33              10
     BLANKLINE       34              10
     TESTBIT         35              10
     DNTOA           36              10
```

```
     OPENRL          27   2523  2523
     FOPEN           70              ?
     FLOCK           71              ?
     FREADMR'        72              10
     SEGMENT LENGTH       3130 (9)
SEG30                1
     NAME           STT  CODE ENTRY SEG
     LISTSL'         1     0    106
     FGETINFO        33              ?
     NTOA            34              10
     BLANKLINE       35              10
     TESTBIT         36              10
     DNTOA           37              10
     PRINTLINE       40              10
     EJECTPAGE       41              10
     CLEANUPRTBUF    2    545   545
     FWRITEDIR'      42              10
     GETREFTABENTRY  3    556   556
     FEOF            43              10
```

```
     FPOINT          46              ?
     FCLOSE          47              ?
     NTOA            50              10
     EJECTPAGE       51              10
     SEGMENT LENGTH       2030
```

```
                  (10)        (11)              (14)          (15)             (17)
PRIMARY DB          350   INITIAL STACK    1440   CAPABILITY           101
SECONDARY DB       1216   INITIAL DL          0   TOTAL CODE  (18) 27320  (19)
TOTAL DB           1566 (12) MAXIMUM DATA  40000 (16) TOTAL RECORDS      155
ELAPSED TIME   00:14:29.887               PROCESSOR TIME    00:24.156 (20)
:
                        (13)
```

Figure 4-1.  Listing of Prepared Program

## Preparation/Execution

Programs compiled in USL files can be prepared *and* executed with the :PREPRUN command. This prepares a temporary program file and then executes the program in that file. (The Segmenter creates $NEWPASS and prepares into it, and the Loader then executes $OLDPASS. The program file is available to the user as $OLDPASS.) The command format is:

   *:PREPRUN uslfile[,entrypoint]*

       *[;NOPRIV]*
       *[;PMAP]*
       *[;LMAP]*
       *[;ZERODB]*
       *[;MAXDATA=segsize]*
       *[;PARM=parameternum]*
       *[;STACK=stacksize]*
       *[;DL=dlsize]*
       *[;LIB=library]*
       *[;CAP=caplist]*
       *[;RL=filename]*

| | |
|---|---|
| *uslfile* | The name of the USL file (from the input set) on which the program has been compiled. (Required parameter.) |
| *entrypoint* | The program entry-point where execution is to begin. This may be the primary entry point of the program, or any secondary entry point in the program's outer block. If the parameter is omitted, the primary entry point is assigned by default. (Optional parameter.) |
| *NOPRIV* | A declaration that the program segments will be placed in *non-privileged* (user) mode. This parameter is intended for programs prepared with the privileged-mode capability. Normally, program segments containing privileged instructions are executed (run) in privileged mode *only* if the program was *prepared* with the privileged-mode (PM) capability-class. (A program containing legally-compiled privileged code, placed in non-privileged mode, may abort when an attempt is made to execute it.) If NOPRIV is specified in the :PREPRUN command, all program segments are placed in *non-privileged* mode. (Library segments are not affected since their mode is determined independently.) (Optional parameter.) |
| *PMAP* | An indication that a listing describing the *prepared* program will be produced on the file whose formal designator is SEGLIST; if no :FILE command referencing SEGLIST is encountered, the listing is produced on the job/session listing device. If this parameter is omitted, the listing is not produced. (Optional parameter.) |

4-17

| | |
|---|---|
| *LMAP* | An indication that a descriptive listing of the *allocated* (loaded) program will be produced on the file whose formal designator is LOADLIST; if no :FILE command referencing LOADLIST is encountered, the listing is produced on the job/session listing device. If LMAP is omitted, the listing is not produced. (Optional parameter.) |
| *ZERODB* | An indication that the initially-defined user-managed (DL-DB) area, and uninitialized portions of the DB-Q (initial) area, will be initialized to zero. If this parameter is omitted, these areas are not affected. (Optional parameter.) |
| *segsize* | Maximum stack area (Z–DL) size permitted. This parameter is included if the user expects to change the size of the DL–DB or Z–DB areas during process execution. If omitted, MPE/3000 assumes that he will not change these areas. (Optional parameter.) |
| *parameternum* | A value that can be passed to the user's program as a general parameter for control or other purposes; when the program is executed, this value can be retrieved from the address Q (initial)-4, where Q (initial) is the Q-address for the outer block of the program. The value can be an octal number or a signed or unsigned decimal number. If *parameternum* is omitted, the Q (initial)-4 address is filled with zeros. (Optional parameter.) |
| *stacksize* | The initial size of the user's initial local data area (Z-Q (initial)) in the stack, in words, as described in the discussion of the :PREP command. (This value must exceed 511 words.) (Optional parameter.) |
| *dlsize* | The initial DL–DB area size, as described in the discussion of the :PREP command. (Optional parameter.) |
| *library* | The order in which applicable segmented procedure libraries are searched to satisfy external references during allocation, where: |

        G = Group library, followed by account public library, followed by system library. (These libraries are defined in Section VII.)

        P = Account public library, followed by system library.

        S = System library.

        If no parameter is specified, *S* is assumed. (Optional Parameter.)

| | |
|---|---|
| *caplist* | The capability-class attributes associated with the user's program, as described in the discussion of the :PREP command. (Optional parameter.) |
| *filename* | The name of a relocatable procedure library (RL) file to be searched to satisfy external references during preparation, as described in the discussion of the :PREP command. (Optional parameter.) |

For *stacksize, dlsize,* and *maxdata,* a value of −1 indicates the default value; it is equivalent to omitting the parameter.

*EXAMPLES:*

*The following command prepares and executes a program on the USL file USEF, with no special parameters declared. (All default values apply.)*

**:PREPRUN USEF**

*The next command prepares and executes a program on the USL file UBASE, beginning execution at the entry-point RESTART, declaring a stacksize of 800 words, and specifying that the library LIBA will be searched to satisfy external references:*

**:PREPRUN UBASE, RESTART; STACK=800; RL=LIBA**

An example of the listing of the loaded program, requested by the LMAP parameter of the :PREPRUN (and :RUN) command, is shown in Figure 4-2. This listing shows the externals referenced by the program and the segments from segmented libraries to which they were bound. On this listing, significant entries are indicated with arrows, keyed to the discussion below.

| Item No. | Meaning |
|---|---|
| 1 | The name of the program file (filename.groupname.accountname). |
| 2 | The name of the external procedure. |
| 3 | The type of the segment referencing the external procedure, where: |

        PROG  = program segment.
        GSL    = group segmented library segment.
        PSL    = public segmented library segment.

| Item No. | Meaning |
|---|---|
| 4 | External parameter checking level. |
| 5 | External segment transfer table (STT) number. |
| 6 | External (logical) segment number. |
| 7 | Entry point segment type, where: |

        GSL    = group segmented library segment.
        PSL    = public segmented library segment.
        SSL    = system segmented library segment.

| Item No. | Meaning |
|---|---|
| 8 | Entry point parameter checking level. |
| 9 | Entry point segment transfer table (STT) number. |
| 10 | Entry point (logical) segment number |
| 11 | A list of the code segment table (CST) numbers to which the program file segments were assigned. The list is ordered by logical segment number. |

PROGRAM FILE SCR4.MPE.SYS



| | ① | ④ | ⑤ | ⑥ | ⑦ | ⑧ | ⑨ | ⑩ |
|---|---|---|---|---|---|---|---|---|
| TERMINATE ② ③ | PROG 0 | 50 | 11 | SSL | 0 | 2 | 37 | |
| SENDMAIL | PROG 0 | 46 | 11 | SSL | 0 | 2 | 40 | |
| DEBUG | PROG 0 | 44 | 11 | SSL | 0 | 1 | 52 | |
| RECEIVEMAIL | PROG 0 | 6 | 11 | SSL | 0 | 1 | 40 | |
| AWAKE | PROG 0 | 5 | 11 | SSL | 0 | 4 | 33 | |
| WHO | PROG 0 | 4 | 11 | SSL | 0 | 4 | 45 | |
| FREADDIR | PROG 0 | 45 | 10 | SSL | 0 | 1 | 0 | |
| FWRITEDIR | PROG 0 | 44 | 10 | SSL | 0 | 2 | 0 | |
| FWRITE | PROG 0 | 43 | 10 | SSL | 0 | 3 | 0 | |
| DLSIZE | PROG 0 | 41 | 10 | SSL | 0 | 15 | 42 | |
| PRINT | PROG 0 | 15 | 7 | SSL | 0 | 11 | 45 | |
| SETJCW | PROG 0 | 13 | 7 | SSL | 0 | 1 | 45 | |
| GETJCW | PROG 0 | 12 | 7 | SSL | 0 | 2 | 45 | |
| ADJUSTUSLF | PROG 0 | 42 | 6 | SSL | 0 | 11 | 46 | |
| FPOINT | PROG 0 | 46 | 2 | SSL | 0 | 5 | 1 | |
| RESETDB | PROG 0 | 34 | 4 | SSL | 0 | 7 | 33 | |
| | | 26 | 2 | | | | | |
| SETSYSDB | PROG 0 | 33 | 4 | SSL | 0 | 11 | 33 | |
| | | 25 | 2 | | | | | |
| FCONTROL | PROG 0 | 14 | 2 | SSL | 0 | 3 | 1 | |
| PROCTIME | PROG 0 | 4 | 2 | SSL | 0 | 15 | 44 | |
| TIMER | PROG 0 | 3 | 2 | SSL | 0 | 23 | 33 | |
| GETUSERMODE | PROG 0 | 10 | 11 | SSL | 0 | 4 | 44 | |
| | | 35 | 4 | | | | | |
| | | 27 | 2 | | | | | |
| | | 55 | 1 | | | | | |
| LOADEDSLSEG | PROG 0 | 53 | 1 | SSL | 0 | 2 | 47 | |
| GETPRIVMODE | PROG 0 | 45 | 11 | SSL | 0 | 5 | 44 | |
| | | 32 | 4 | | | | | |
| | | 24 | 2 | | | | | |
| | | 52 | 1 | | | | | |
| QUIT | PROG 0 | 7 | 11 | SSL | 0 | 11 | 31 | |
| | | 52 | 10 | | | | | |
| | | 25 | 5 | | | | | |
| | | 50 | 1 | | | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| FGETINFO | PROG 0 | 14 | 11 | SSL | 0 | 4 | 2 | |
| | | 46 | 10 | | | | | |
| | | 32 | 6 | | | | | |
| | | 56 | 3 | | | | | |
| | | 15 | 2 | | | | | |
| | | 33 | 1 | | | | | |
| | | 31 | 0 | | | | | |

131 132 133 134 135 155 156 177 200 201 ⑪

Figure 4-2. Listing of Loaded Program

## Execution

Programs that have been compiled and prepared, and that therefore exist on program files, can be executed by the :RUN command, entered as follows:

    :RUN   progfile[,entrypoint]
          [;NOPRIV]
          [;LMAP]
          [;MAXDATA=segsize]
          [;PARM=parameternum]
          [;STACK=stacksize]
          [;DL=dlsize]
          [;LIB=library]

| | |
|---|---|
| *progfile* | The name of the program file (from the input set) that contains the prepared program. (Required parameter.) |
| *entrypoint* | The program entry-point where execution is to begin. This may be the primary entry-point of the program or any secondary entry-point in the program's outer block. If this parameter is omitted, the primary entry-point (where (execution normally begins) is assigned by default. (Optional parameter.) |
| *NOPRIV* | A declaration that the program segments will be placed in *non-privileged* mode; this parameter is intended for programs prepared with the privileged-mode capability. Normally, program segments containing privileged instructions are executed (run) in privileged mode *only* if the program was *prepared* with the privileged-mode (PM) capability-class. (A program containing legally-compiled privileged code, placed in non-privileged mode, may abort when an attempt is made to execute it.) If NOPRIV is specified in the :RUN command, all program segments are placed in the *non-privileged* mode. (Library segments are not affected since their mode is determined independently.) (Optional parameter.) |
| *LMAP* | An indication that a listing of the *allocated* (loaded) program will be produced on the file whose formal designator is LOADLIST; if no :FILE command referencing LOADLIST is encountered, the listing is produced on the job/session listing device. If LMAP is omitted, the listing is not produced. (Optional parameter.) |
| *segsize* | Maximum stack area (Z-DL) size permitted, as described in the discussion of the :PREP command. (Optional parameter.) |
| *parameternum* | A decimal or octal value that can be passed to the user's program as a general parameter for control or other purposes; when the program is executed, this value can be retrieved from the address Q (initial) $-4$, where Q (initial) is the Q-address for the outer block of the program. The value can be an octal number or a signed or unsigned decimal number. If *parameternum* is omitted, the Q (initial) $-4$ address is filled with zeros. (Optional parameter.) |

| | |
|---|---|
| *stacksize* | The initial size of the user's initial local data area (Z-Q (initial)) in the stack, in words, as described in the discussion of the :PREP command. (This value must exceed 511 words.) (Optional parameter.) |
| *dlsize* | The initial DL-DB area size, as described in the discussion of the :PREP command. (Optional parameter.) |
| *library* | The order in which applicable segmented procedure libraries are searched to satisfy external references during allocation, where: |

G = Group library, followed by account public library, followed by system library. (These libraries are defined in Section VII.)

P = Account public library, followed by system library.

S = System library.

If no parameter is specified, S is assumed. (Optional parameter.)

If values for *segsize*, *stacksize*, and *dlsize* are explicitly specified in the :RUN command, they override such values assigned at preparation time (which are recorded in the program file). If any of these parameters are omitted, the corresponding values recorded in the program file take effect.

*EXAMPLES:*

*The following command executes a program existing on the program file PROG3, with no special parameters specified. (All default values are used.)*

**:RUN PROG3**

*The next command executes a program existing on the program file PROG4, beginning at the entry-point SECLAB. All segments in the program will run in non-privileged mode.*

**:RUN PROG4, SECLAB; NOPRIV**

## CALLING MPE/3000 SUBSYSTEMS

In addition to the BASIC/3000 interpreter and the SPL/3000, COBOL/3000, and FORTRAN/3000 compilers, various other programs are available that run as subsystems of MPE/3000. These programs, and the way in which they are accessed, are described below.

## EDIT/3000

EDIT/3000, described in the manual *HP 3000 Text Editor (03000-90012),* is used to create and modify files of upper and lower-case ASCII characters. To call EDIT/3000, the user enters:

:EDITOR *[listfile]*

| listfile | The name of a file that receives any output resulting from the EDIT/3000 command LIST specifying the OFFLINE option. (The designator *EDTLIST* must not be used.) If *listfile* is omitted, such output is transmitted to the job/session list device ($STDLIST). (The file to be edited is specified after EDIT/3000 has been called.) (Optional parameter.) |
| --- | --- |

## STAR/3000

The Statistical Analysis Routines (STAR/3000), described in *HP 3000 Statistical Analysis Routines (03000-90011)*, allows the user to access the statistical functions of the HP 3000 Scientific Library. STAR/3000 is called by this command:

:STAR [listfile] [,NOLIST]

| listfile | File on which output from STAR/3000 is to be written. (Its formal designator is STRLIST.) If omitted, $STDLIST (typically, the terminal for sessions and the printer for jobs) is used. (All commands are input from the job/session input device and results are output to *listfile*.) (Optional parameter.) |
| --- | --- |
| NOLIST | A specification that STAR/3000 command input will *not* be listed on *listfile* when STAR/3000 is run in batch mode. (Optional parameter.) |

## TRACE/3000

TRACE/3000 allows the user to monitor program execution, checking the status of the program whenever a variable is changed or a label is passed. TRACE/3000 is invoked through compiler subsystem command records, as noted in *HP 3000 Symbol Trace (03000-90015)*.

## SORT/3000

SORT/3000, described in *HP 3000 SORT/MERGE (03000-90053)*, allows the user to sort records in a file into prescribed order, or to merge pre-sorted records in multiple files into one sorted file. SORT/3000 can be invoked directly through MPE/3000 commands, or called, in phases, from user programs or procedures.

When invoking SORT/3000 through MPE/3000, the user first specifies the appropriate input/output files through MPE/3000 :FILE commands, or allows the default file designators to take effect (as described in *HP 3000 SORT/MERGE (03000-90053)*.) He then invokes either the Sort program or the Merge program by entering one of the following commands.

*For Sort:*

:RUN SORT [;STACK = size]

*For Merge:*

>  *:RUN MERGE [;STACK = size]*

*size*                  The amount of main-memory storage to be used by the Sort or Merge program, in words. If *size* is omitted, the default value assigned is 1000 words. (Optional parameter.)


## SDM/3000

The execution of on-line diagnostic programs, run by a person with the *Diagnostician* user attribute, is controlled by the HP/3000 Diagnostic Monitor (SDM/3000), described in *HP 3000 System Diagnostic Monitor (03000-90016)*. This program allows the operator to invoke, execute, and modify diagnostic programs. It is called by entering:

>  *:RUN SDM*


## HP/3000 Stand-Alone Diagnostic Utility Program

Magnetic tapes containing stand-alone diagnostic programs are prepared through the HP/3000 Stand-alone Diagnostic Utility Program, described in *HP 3000 Stand-alone Diagnostic Utility Program (03000-90017)*. These programs can then be loaded and executed without any other supporting software. The Utility Program is invoked through this command:

>  *:RUN SDUP; NOPRIV*


## MPE/3000 Segmenter

The MPE/3000 Segmenter can be accessed directly by the user, allowing him to enter commands that

- Create, delete, activate, and deactivate RBM's within a USL.

- Manage procedure libraries used to resolve external program references

The user accesses the segmenter by entering:

>  *:SEGMENTER [listfile]*

*listfile*            An ASCII file from the output set, to which any listable output is written. (The formal file designator is *SEGLIST.*) If *listfile* is omitted, the standard job/session list device ($STDLIST) is assumed. (Optional parameter.)

He then enters commands relating to the USL's or procedure libraries. These commands are explained in Section VII.

## SAMPLE PROGRAMS

The following sample programs illustrate, in context, some of the commands introduced in this and the previous section.

The following program shows the listing from a session where the programmer accessed and used the BASIC/3000 Interpreter:

```
:HELLO MATH,BOBBIE.ACMATH, GPMATH
USER PASSWORD?                              (LOG-ON)
XXXXXXXX
SESSION NUMBER=#S231
THU,MARCH 23, 1972, 2:35 PM
HP32000B.02.08
:BASIC                                     (COMMAND TO ACCESS BASIC)
>                                          (BASIC PROMPT CHARACTER,
                                           FOLLOWED BY A BASIC COMMAND)
.
.        (BASIC COMMANDS AND
.        STATEMENTS)
.
.
.
.
>EXIT                                      (EXIT FROM BASIC)
:BYE
CPU(SEC)=14                                (LOG-OFF)
CONNECT(MIN)=2
THU,MARCH 23, 1972, 2:37 PM
END OF SESSION
```

The next program illustrates the use of EDIT/3000 and BASIC/3000 within a job. The input, in card form, appears as



```
:EOJ
EXIT
RUN PROG 67
:BASIC , FDATA
/EXIT   (Exit from Editor)
(Commands to create and edit a data file
called FDATA, for BASIC)
(Editor Commands)
:EDITOR
:JOB MATH, BOBBIE. ACMATH
```
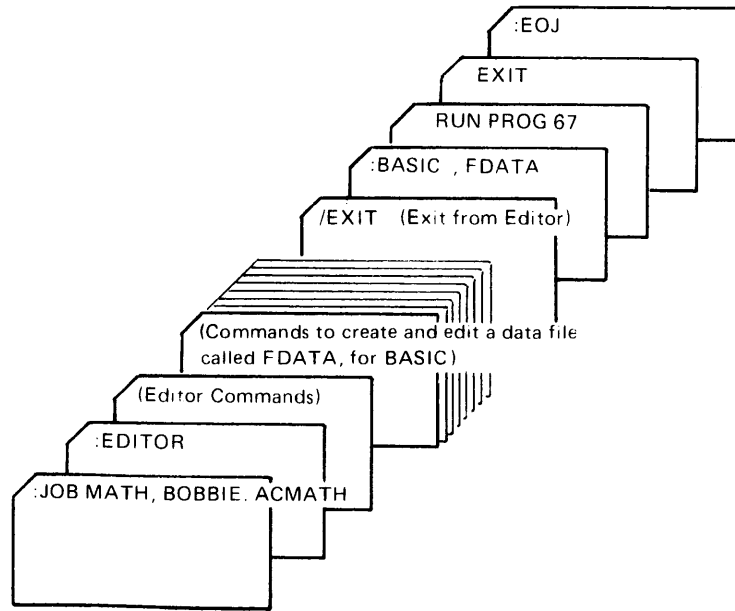
The output listing from this batch job would appear as

```
:JOB MATH,BOBBIE.ACMATH
JOB NUMBER=#J23
THU,MARCH 23, 1972, 2:35 PM
HP32000B.02.08
:EDITOR
  .
  .
  .
  .        (EDITED OUTPUT)
  .
  .
  .
:BASIC ,FDATA
  .
  .
  .
  .        (OUTPUT FROM BASIC PROGRAM)
  .
  .
  .
:EOJ
CPU(SEC)=110
ELAPSED(MIN)=15
THU,MARCH 23,1972, 2:50 PM
END OF JOB
```

# SECTION V
# Managing Files

MPE/3000 organizes and handles as files all information input to and output from the computer, relieving the user of the software-writing effort normally involved when interacting directly with physical peripheral devices. Input and output are generally performed by sharable system code that translates file names specified by the user into actual hardware addresses, allowing the user to remain virtually unaware of detailed file specifications and hardware characteristics.

## FILE CHARACTERISTICS

A file can contain MPE/3000 commands, programs, or data, or any combination of these elements, written in ASCII or binary code. For example, a file input from the card reader might contain MPE/3000 commands, a user program, and user data, all in ASCII code. Compilation of the user program would produce a USL file containing object output in binary code and another file containing a listable copy of the user's source program in ASCII code. Preparation of the object program produces a binary-coded program file containing the user's program in segmented form.

Within a file, information is organized as a set of logical records — fields of data input, processed, and output as a unit. A logical record is the smallest data grouping directly addressable by the user. Its length is specified by the user when he creates or defines the file.

Information is moved between a file and main memory in *physical records;* a physical record is the basic unit that can be transferred to or from the device on which the file resides. Physical records can be longer, shorter, or the same size as the logical records in the file. In files on disc or magnetic tape, physical records are organized as *blocks* that always contain an integral number of logical records; thus, on these devices, physical records are either longer than, or the same size as, logical records. (The block size is specified implicitly by the user in the command or intrinsic call that creates the file. But input/output and blocking are performed automatically by MPE/3000, freeing the user of responsibility for the actual record-handling details.)

On unit-record devices, however, the size of the physical records in a file is determined by the device itself. Thus, each physical record read from a card reader consists of one punched card; each physical record written to a line printer consists of one line of print. On unit record-devices in multi-record mode, *logical* records are *not* blocked. For example, on punched cards, each logical record is assumed to begin at the first column of the card; a single 100-character logical record is read as 80 characters from one card (physical record) and 20 characters from the next card. The next logical record is assumed to begin in the first column of the third card encountered. Similarly, when a file is transmitted to a printer, each logical record appears as one line of print (physical record), left-justified; if the logical record is longer than the print line, the remaining information is continued on the next line, also left-justified.

On disc, files can be organized to permit either sequential or direct access. Direct-access files contain logical records of fixed or undefined length. But sequential-access files contain logical records of fixed or varying length.

MPE/3000 manages each file on disc as a set of *extents;* each extent is an integral number of contiguously-located disc sectors. All extents (except possibly the last) are of equal size. When a file is opened, the first extent (containing at least one sector for file label information) is allocated immediately; other extents, up to a maximum of 16, are allocated as needed. Alternatively, the user can request immediate allocation of more than one extent when the file is opened. (The size of each extent is determined as noted in the discussion of the :FILE command parameter *numextents*, later in this section. All extents are allocated on the same disc drive.)

When a file contains only fixed length records, the user can calculate the effective disc space (in sectors) required by the file, with the following formula. (This formula applies to every type of disc drive supported by MPE/3000.)

$$S = \left\lceil \frac{N+B-1}{B} \right\rceil \ x \ \left\lceil \frac{(LxB) + 127}{128} \right\rceil$$

$S$      The number of disc sectors required by the file.

$N$      The number of fixed-length logical records in the file.

$B$      The number of logical records in a block (blocking factor).

$L$      The length of each logical record, in 16-bit *words.*

The constant 128 is the number of words in a sector. Each expression within brackets is evaluated separately and rounded upward before the final multiplication takes place. (A notation in the form $\lceil x \rceil$ means "ceiling (x)" — the smallest integer greater than or equal to x.)

*EXAMPLE:*

*The effective disc space for a file containing 100 records of 50 words each, with a blocking factor of 3, would be calculated as follows:*

$$S \quad = \quad \left\lceil \frac{(100 + 3) - 1}{3} \right\rceil \quad x \quad \left\lceil \frac{(50 \ x \ 3) + 127}{128} \right\rceil$$

$$= \quad [34] \quad x \quad [3]$$

$$= \quad 102 \ sectors$$

## FILE/DEVICE RELATIONSHIPS

Devices required by files are allocated automatically by MPE/3000. The user can specify these devices by type (such as any card reader or line printer), or by a logical device number related to a particular device (such as a specific line printer). (A unique logical device number is assigned to each device when the system is configured.) Regardless of what device a particular file resides on, when the user's program asks to read that file, it references the file by its formal file designator. MPE/3000 then determines the device on which the file resides, or its disc address if applicable, and accesses it for the user. When the user's 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/3000 then automatically allocates the required device to that file. Throughout its existence, every file remains device-independent; that is, it is always referenced by the same formal file designator regardless of where it currently resides. The user's program always deals with logical records.

## FILE DOMAINS

The set of all permanent disc files in MPE/3000 is known as the *system file domain*. Within this domain, files are assigned to accounts and organized into groups under those accounts. The log-on process associates the user with an account and group which provides the basis for his local file references. The user may be required to supply *passwords* for the account and group to log-on, but thereafter (if the normal (default) MPE/3000 file security provisions are in effect) he can:

- Have unlimited access to any file within his log-on or home group. (If, however, the file is protected by a *lockword*, the user must know this lockword.)

- Read, and execute programs residing in, any file in the public group of his account, and in the public group of the system account.

As noted later, the default MPE/3000 file security provisions can be overridden at the account, group, and/or file level to provide greater or lesser file access restrictions.

Potentially, if the MPE/3000 file security provisions at the account, group, and file levels were all suspended, and the user knew all account and group names and file lockwords, he could

access any permanent file in the system once he logged-on. (Notice that once a user logs-on to an account and is associated with a group, he does *not* need to know the *passwords* for other accounts and groups to access files assigned to them — he only requires their account and group names. But, if any of these files are protected by a file lockword, the user must know that lockword.)

For every job or session running in the system, MPE/3000 recognizes another file domain, called the *job or session file domain*. This domain contains all temporary files opened and closed within the job or session without being saved (declared permanent). Files in these domains are deleted when the job or session terminates (if they are job/session temporary files), or when the creating program ends (if they are regular temporary files).

## FILE LABELS

MPE/3000 reads and writes file labels for files on disc during allocation of the devices on which the files reside. The format and content of file labels is presented in Appendix F.

## FILE ACCESSING

The user accesses files through commands and intrinsic calls. Commands, described in this section, are issued external to the user's program and perform general functions, such as creating, deleting, or listing a file. Intrinsic calls, described in the next section, are issued *programmatically* (within a user's program). Generally, files are opened (through the FOPEN intrinsic); operated on through various intrinsics that read information from them, write information to them, update them, or manipulate them; and finally, they are closed (through the FCLOSE intrinsic).

Within a user program, an MPE/3000 system program such as a compiler, or a command executor, a file is accessed by its *formal file designator* — the name by which the particular program recognizes the file. (In SPL/3000, this is the name associated with it by the FOPEN intrinsic.) At program execution time, this formal file designator is always associated or equated with an *actual file designator* — the name of the actual file to be used and the physical device upon which it resides, as recognized throughout the system by MPE/3000. Thus, the actual file designator is an execute-time re-definition of the file specified in the program by the formal file designator. If the user does not specify an actual file designator for a formal file designator, MPE/3000 uses the formal file designator for the actual file designator.

MPE/3000 recognizes actual file designators for four types of files:

- System-Defined Files

- User Pre-Defined Files

- New Files

- Old Files

The programmer can specify any of these designators programmatically.

## System-Defined Files

System-defined file designators indicate those files that MPE/3000 uniquely identifies as standard input/output devices for a job/session. They are referenced as follows:

| Actual File Designator | Device/File Referenced |
|---|---|
| $STDIN | A file name indicating the standard job or session input device (that from which the job or session is initiated). For a job, this is typically a card reader. For a session, this is typically a terminal. Input data images in the $STDIN file should not contain a colon in column 1, since this indicates the end-of-data. (When data is to be delimited, this should be done through the :EOD command, which performs no other function.) |
| $STDINX | Equivalent to $STDIN, except that MPE/3000 command images (those with a colon in column 1) encountered in a data file, are read without indicating the end of data. (However, the commands :JOB, :DATA, :EOJ, and :EOD are exceptions that always indicate the end-of-data but are otherwise ignored in this context; they are *never* read as data.) |
| $STDLIST | A file name indicating the standard job or session listing device (customarily a printer for a batch job and a terminal for a session). |
| $NULL | The name of a non-existent "ghost" file that is always treated as an empty file. When referenced as an input file by a program, that program receives an end-of-data indication upon each access. When referenced as an output file, the associated write request is accepted by MPE/3000 but no physical output is actually performed. Thus, $NULL can be used to discard unneeded output from a running program. |

## User Pre-Defined Files

A user pre-defined file is any file that was previously defined or re-defined in a :FILE command, as discussed later in this section. In other words, it is a back-reference to that :FILE command. It is referenced by the following file designator format:

*formaldesignator*

| | |
|---|---|
| *formaldesignator* | The name used in the *formaldesignator* parameter of the :FILE command. |

## New Files

New files are files that have not yet been created, and are being created/opened for the first time by the current program. New files can have the following actual file designators:

| Actual File Designator | File Referenced |
|---|---|
| $NEWPASS | A temporary disc file that can be automatically passed to any succeeding MPE/3000 command within the same job/session, which references it by the file name $OLDPASS. Only one such file with this designation can exist in the job/session at any one time. (When $NEWPASS is closed, its name is automatically changed to $OLDPASS, and any previous file named $OLDPASS in the job/session is deleted. (*Passing* is explained in later examples.)) |
| *filereference* | Unless the user specifies otherwise, this is a temporary file, residing on disc, that is destroyed on termination of the creating program. If closed as a job/session temporary file, as shown later in this section, it is purged at the end of the job/session. If closed as a permanent file, it is saved until purged by the user. Typically, this format consists of a file name containing up to eight alphanumeric characters, beginning with a letter, as discussed below. In addition, other elements (such as a group name, account name or lockword) can be specified. |

## Old Files

Old files are existing named files presently in the system. They may be named by the following designators:

| Actual File Designator | File Referenced |
|---|---|
| $OLDPASS | The name of the temporary file last closed as $NEWPASS. |
| *filereference* | Any other old file to which the user has access. (The *filereference* format is discussed below.) It may be a job/session temporary file created in this or a previous program in the current job/session, a permanent file saved by any program, or a permanent file built (with the :BUILD command) in any job/session. |

## Filereference Formats

When the user references a file by the *filereference* designator, he writes *filereference* in any of three formats. (In no case, however, can *filereference* exceed a total of 35 characters.)

The format most commonly used applies to files contained in the user's log-on group:

   *filename [/lockword]*

In this format, *filename* is the name of the file. It is written as a string of up to eight alphanumeric characters, beginning with a letter. (Because a file reference is always qualified by the group and account to which the file belongs, individual file names must be unique only within the file's group.) The lockword need only be specified when referencing an existing (old) disc file protected by a lockword assigned by the user who created the file. Neither the *filename* nor *lockword* should contain embedded blanks. Additionally, the slash mark (/) that separates these two elements should not be preceded nor followed by blanks.

> *NOTE:*   *Whenever a file is referenced in any command, or in the*
> *FOPEN intrinsic that opens the file (described in Section VI),*
> *the lockword (if any) must always be supplied — even if the*
> *accessor is the creator of the file.*

*EXAMPLE:*

*The following three examples illustrate this filereference format:*

   OUTPUT

   AL126797

   PAYROLL/X229AD

The format used when the programmer references a file in his log-on account but within another group is:

   *filename [/lockword].groupname*

Embedded blanks are not permitted. The group to which the file belongs is designated by *groupname.* As an example, if a user logs-on under a group other than his home group, but wants to reference a file in the home group, he must enter the name of the home group as the *groupname* specifier.

Remember that the file *lockword* relates only to the ability to access files, and not to the account and group *passwords* used during log-on.

*EXAMPLES:*

*These file references include groupname specifiers:*

        F ILEB.GRP2

        X3.PUB

        GOFILE/Z22.GR07

The third *filereference* format is

    *filename [/lockword].groupname.accountname*

Embedded blanks are not permitted. This format is used to reference a file assigned to a group that belongs to an account that is *not* the user's log-on account. The account is specified by *accountname.*

*EXAMPLE:*

*The following file references contain accountname specifiers:*

        SMITH.GRP7.ACCT47

        F ILLER.PUB.SYS

        NB3/X23DG.GRP5.ACCT90

## Lockwords

As noted earlier, the creator of a disc file can assign to it a lockword which must thereafter be supplied to access the file in any way. This lockword is independent of, and serves in addition to, the other basic security provisions governing the file. The lockword is assigned by including it in the *filereference* parameter used when the file is created. It can be subsequently changed by the :RENAME command or the FRENAME intrinsic, discussed later. (:RENAME and FRENAME are also used to initially assign a lockword to an existing file.) At any time, a file can have only one lockword. Only the creator of a file can change a lockword for that file.

*EXAMPLE:*

*To assign the lockword SESAME to a new file named FILEA, the user enters the following :BUILD command. (A complete discussion of the :BUILD command, used to create new disc files, appears later in this section.)*

        :BUILD FILEA/SESAME

When initially requesting access to an old disc file protected by a lockword, the user must supply the lockword in the following manner:

- In batch mode, as part of a file name specified in the :FILE command or FOPEN intrinsic call that establishes file access. If a file is protected by a lockword, but no lockword is supplied, access is not granted.

- In session mode, as part of a file name specified in the :FILE command, or FOPEN intrinsic call. If a file is protected by a lockword but no lockword is supplied when the file is opened, MPE/3000 interactively requests the user to supply the lockword, as follows:

    *LOCKWORD: filename.groupname.accountname?*

*EXAMPLE:*

*In the following :FILE command, the old file XREF (protected by lockword OKAY) is referenced.*

```
:FILE INPUT=XREF/OKAY, OLD
```

*Remember that the lockword is always separated from the filename by a slash.*

On terminals with the ESC (Escape) key facility, the user can inhibit the echo facility (and thus suppress printing of the lockword) prior to entering the lockword by pressing the *ESC* and ; keys.

After he enters the lockword, he can restore echoing by pressing the ESC and : keys.

Where the terminal does not support dynamic half-duplex operation, the user can conceal a lockword simply by omitting it from the file reference. In this case, MPE/3000 outputs a request for the lockword, directs the carriage to skip a line, prints an eight-character mask, and returns the carriage to the beginning of the mask. The user then enters the lockword, which is echoed on top of the mask to ensure privacy.

## File System Accounting

When the MPE/3000 default accounting provisions are in effect at the account and group level, the amount of permanent disc file space accumulated by users is monitored by MPE/3000 but is not limited. (The default provisions are described fully in *HP 3000 Multiprogramming Executive System Manager/Supervisor Capabilities (03000-90038).*) However, limits on the amount of permanent file space allotted can be established at the account level (by System Manager Users) and the group level (by Account Manager Users). The limits are established in terms of disc sectors. When an attempt is made to save a new disc file or to create, rename, or add extents to a permanent file, if either the account or group disc file space count exceeds the current limit, the file request is denied. Otherwise, the account and group disc file space counts are updated.

### Input/Output Sets

All file designators described above can be classified as those used for input files (*Input Set*) and those used for output files (*Output Set*).  These sets, referred to frequently throughout this manual, are defined as follows:

#### Input Set

| | |
|---|---|
| $STDIN | The job/session input device. |
| $STDINX | The job/session input device with commands allowed. |
| $OLDPASS | The last $NEWPASS file closed. |
| $NULL | A constantly-empty file that will return an end-of-file indication whenever it is read. |
| *formaldesignator* | A back-reference to a previously-defined file. |
| *filereference* | A file name, and perhaps account and group names and a lockword. |

#### Output Set

| | |
|---|---|
| $STDLIST | The job/session listing device. |
| $OLDPASS | The last file passed. |
| $NEWPASS | A new temporary file to be passed. |
| $NULL | A constantly-empty file that returns a successful indication whenever information is written to it. |
| *formaldesignator* | A back-reference to a previously-defined file. |
| *filereference* | A file name, and perhaps account and group names and a lockword. |

## SPECIFYING FILE CHARACTERISTICS

Formal (programmatic) file designators can be equated with actual file designators through the :FILE command.  This command enables the user to

*   Write programs that reference files whose actual names and characteristics he may not yet know.  This allows him to remain uncommitted to specific disc files or devices until run-time, when their names are equated with the user's programmatic references.

● Issue detailed file specifications at run-time that override any corresponding specifications declared within the user's program (through an FOPEN intrinsic call) or in a previous MPE/3000 :FILE command.

The specifications in a :FILE command do not take effect until the user's program is running and opens the file referenced. The :FILE command specifications hold throughout the entire job/session, unless superceded (by another :FILE command) or revoked by the user (through the :RESET command). At job or session termination, however, all :FILE commands are cancelled.

If the user does not include in the job stream any file command referencing a particular formal file designator named in his program, that formal file designator will be used as the actual file designator, and any file characteristics specified (explicitly or by default) within the program will apply.

When two (or more) :FILE commands referencing the same formal designator appear in a job/session, the second command replaces the first one.

The :FILE command can be written in any of the following formats, depending on the type of file referenced, and applies to files on disc, tape, or any other device.

**For new files:**

$$:FILE\ formaldesignator \quad \left[ \begin{array}{l} = \$NEWPASS \\ [\ = filereference][,NEW] \end{array} \right]$$

$$\left[ ;REC = \left[ recsize \right] \left[ , \left[ blockfactor \right] \left[ , \left[ \begin{array}{c} F \\ U \\ V \end{array} \right] \left[ \begin{array}{c} ,BINARY \\ ,ASCII \end{array} \right] \right] \right] \right]$$

$$\left[ \begin{array}{l} ;CCTL \\ ;NOCCTL \end{array} \right]$$

$$[;ACC = accesstype]$$

$$\left[ \begin{array}{l} ;NOBUF \\ ;BUF\ [= numbuffers] \end{array} \right]$$

$$\left[ \begin{array}{l} ;EXC \\ ;EAR \\ ;SHR \end{array} \right]$$

$$\left[ \begin{array}{l} ;MR \\ ;NOMR \end{array} \right]$$

$$\begin{bmatrix} ;DEL \\ ;SAVE \\ ;TEMP \end{bmatrix}$$

[;DEV = device]

[;CODE = filecode]

[;DISC = [filesize] [,[numextents] [,initalloc]]]


**For old files:**

$$;FILE\ formaldesignator \quad \begin{bmatrix} = \$OLDPASS \\ \begin{bmatrix} = filereference \end{bmatrix} \begin{bmatrix} ,OLD \\ ,OLDTEMP \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} ;REC = \begin{bmatrix} recsize \end{bmatrix} \begin{bmatrix} , \begin{bmatrix} blockfactor \end{bmatrix} \begin{bmatrix} , \begin{bmatrix} F \\ U \\ V \end{bmatrix} \begin{bmatrix} ,BINARY \\ ,ASCII \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} ;CCTL \\ ;NOCCTL \end{bmatrix}$$

[;ACC = accesstype]

$$\begin{bmatrix} ;NOBUF \\ ;BUF\ [= numbuffers] \end{bmatrix}$$

$$\begin{bmatrix} ;EXC \\ ;EAR \\ ;SHR \end{bmatrix}$$

$$\begin{bmatrix} ;MR \\ ;NOMR \end{bmatrix}$$

$$\begin{bmatrix} ;DEL \\ ;SAVE \\ ;TEMP \end{bmatrix}$$

[;DEV = device]

[;CODE = filecode]

*[;DISC = [filesize] [,[numextents] [,initalloc]]]*


NOTE:   The parameter group *[;DISC = [filesize] [,[numextents]*
        *[,initalloc]]]* cannot be included if the parameter group
        $\left[= filereference\right] \left[\begin{matrix} ,OLD \\ ,OLDTEMP \end{matrix}\right]$ is specified.


**For user pre-defined files:**

   *:FILE formaldesignator*     =     *\*formaldesignator$_1$*


**For system-defined files:**

   *:FILE formaldesignator*     =     *$NULL*


   *:FILE formaldesignator*     =     $\left\{\begin{matrix} \$STDIN \\ \$STDINX \\ \$STDLIST \end{matrix}\right\}$


$$\left[,REC = \left[recize\right] \left[, \left[blockfactor\right] \left[, \left[\begin{matrix} F \\ U \\ V \end{matrix}\right] \left[\begin{matrix} ,BINARY \\ ,ASCII \end{matrix}\right]\right]\right]\right]$$


$$\left[\begin{matrix} ,CCTL \\ ,NOCCTL \end{matrix}\right]$$


*[;ACC = accesstype]*


$$\left[\begin{matrix} ,NOBUF \\ ,BUF \ [= numbuffers] \end{matrix}\right]$$


$$\left[\begin{matrix} ,EXC \\ ,EAR \\ ,SHR \end{matrix}\right]$$


$$\left[\begin{matrix} ,MR \\ ,NOMR \end{matrix}\right]$$

## Command Parameters

The variable parameters for the :FILE command are

| | |
|---|---|
| *formaldesignator* | The programmatic formal file designator, as referenced in the user's program. (Required parameter.) |
| *filereference* | A file name (and perhaps account and group names and a lockword) in the *filereference* format. If a new file is referenced, but the *filereference* parameter is omitted from the :FILE command, the name is equated to the formal designator. (Optional parameter.) |
| *NEW* | A specification that the file is a new file. (Required parameter if *filereference* is used in the *new file* format.) |
| *OLD* | A previously-existing permanent file saved in the system file domain. The file continues to exist after the current job/session terminates. (Optional parameter.) |
| *OLDTEMP* | A previously-existing temporary file in the job/session file domain. This file is deleted at the end of the current job/session. (Optional parameter.) |
| *recsize* | The size of the logical records in the file. If a positive number, this represents *words*. If a negative number, this represents *bytes*. (If the records are undefined-length, this represents their maximum size. For variable-length records, the maximum size is *recsize* x *blockfactor.)* The default value is a value supplied at configuration time. |

The values generally specified by HP are

| | | |
|---|---|---|
| Disc | | 128 |
| Tape | = | 128 |
| Printer | = | -132 |
| Card Reader | = | -80 |
| Card Punch | = | -80 |
| Terminal | = | -72 |

(Optional parameter.)

| | |
|---|---|
| *blockfactor* | The size of each buffer to be established for the file, specified as an integer 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* x *blockfactor*). For undefined-length records, *blockfactor* is always one logical record per block; any unused portion of the block is filled with ASCII blanks or binary zeros.) The *blockfactor* value specified may be overridden by MPE/3000. The default value is calculated by dividing the specified *recsize* into the configured physical record size; this value is rounded downward to an integer that is never less than 1. (Optional parameter.) |

| | |
|---|---|
| *F* | Specifies fixed-length records. (Optional parameter.) |
| *U* | Specifies records of undefined length (no blocking). (Optional parameter.) |
| *V* | Specifies variable-length records. (No blocking on unit-record devices.) (Optional parameter.) |
| *BINARY* | Specifies binary-coded records, always odd-parity. (Optional parameter.) |
| *ASCII* | Specifies ASCII-coded records, always even parity. (Optional parameter.) |
| *CCTL* | Indicates that the user is supplying carriage-control characters with his write requests, valid for any ASCII list file. (Optional parameter.) |
| *NOCCTL* | Indicates that no carriage-control characters are supplied with write requests. (Optional parameter.) |
| *accesstype* | The type of access allowed users of this file: |

| | | |
|---|---|---|
| IN | = | Read-access only. (The FWRITE, FUPDATE and FWRITEDIR intrinsic calls cannot reference this file.) |
| OUT | = | Write-access only. Any data on the file prior to the current FOPEN request is deleted. (The FREAD, FREADSEEK and FREADDIR intrinsic calls cannot reference this file.) |
| OUTKEEP | = | Write-access only, but previous data in the file is not deleted. (The FREAD, FREADSEEK and FREADDIR intrinsics cannot reference this file.) |
| APPEND | = | Append-access only. The FREAD, FREADSEEK, FREADDIR, FPOINT, FSPACE, and FWRITEDIR intrinsic calls cannot be issued for this file. |
| INOUT | = | Input/output access. Any file intrinsic except FUPDATE can be issued against this file. |
| UPDATE | = | Update access. All file intrinsics, including FUPDATE, can be issued for this file. |

If *accesstype* is omitted in the :FILE command (and in the FOPEN intrinsic call that opens the file), the default values assigned are IN for input devices such as card readers, OUT for output devices such as printers, and INOUT for input/output devices such as discs, magnetic tapes, and terminals.

If a process attempts to violate the *accesstype* of a file, an error is returned. (Optional parameter.)

| | |
|---|---|
| *NOBUF* | Specifies that *no* input/output buffering is to take place, and no buffers are allocated for the file. (Optional parameter.) |
| *numbuffers* | The number of buffers to be allocated to the file. This must be an integer value. The maximum value is 16. If omitted or set to 0, a default value of 2 is assigned. This parameter is not used for files representing interactive terminals, since a system-managed buffering method is always used in such cases. (Optional parameter.) |
| *EXC* | After the file is opened, prohibits concurrent access (in any mode), to this file through another FOPEN request, whether issued by this or another process, until this process issues an FCLOSE request or terminates. (Optional parameter.) |
| *EAR* | After the file is opened, prohibits concurrent write-access to this file through another FOPEN request within this or another process, until this process issues an FCLOSE request or terminates. (Optional parameter.) |
| *SHR* | After the file is opened, permits concurrent access to this file through another FOPEN request issued by this or another process. (Optional parameter.) |
| *MR* | Specifies that individual read or write requests are not confined to record boundaries, as explained under the FOPEN request discussion in Section VI. (Restricted to NOBUF files.) (Optional parameter.) |
| *NOMR* | Specifies that individual read or write requests are confined to record boundaries. (Optional parameter.) |
| *DEL* | Specifies that the file is to have regular temporary file disposition; it will be deleted when the user's program closes it. |
| *SAVE* | Specifies that the file is to have permanent file disposition; when the user's program closes it, the file remains in the system file domain, potentially available to other users. (Optional parameter.) |
| *TEMP* | Specifies that the file is to have job/session temporary file disposition. When the user's program closes the file, it remains in the job/session file domain. But when the job/session terminates, the file is deleted. (Optional parameter.) |

*device*

A *device class name* designating the type of device, or a *logical device number* indicating the specific device, on which the file resides.

The device class name is used to make a non-specific, generic reference to a *type* of device (such as any disc drive or magnetic tape unit). (These names are defined and related to specific sets of devices when the system is configured. All must contain from one to eight alphanumeric characters, begin with a letter, and terminate with any non-alphanumeric character such as a blank. Examples are CARD, LP, TTY, and TAPE2.) The default is DISC.

The logical device number refers to a specific device. This is a number assigned to each device when the system is generated. This specification is only used when assignment of a particular device is truly necessary. For example, a user would specify the logical device number of a specific device when running a hardware diagnostic program for checking that device. Note that the device specification is *not* used if the file is an old disc file or if the actual file designator used is $STDIN, $STDINX, $STDLIST, $NEWPASS, $OLDPASS, or $NULL (since these names are already assigned to devices by the system). (Optional parameter.)

*filecode*

An integer or mnemonic recorded in the file label and made available to processes accessing the file through the FGETINFO intrinsic (Section VI). If an integer is used, it must be a positive value ranging from 0 to 1023, or one of the HP-defined integers shown below. These HP-defined integers also have corresponding mnemonics that can be used in their place:

| Mnemonic | HP-defined Integer | Defines the File as: |
|----------|--------------------|----------------------|
| USL | 1024 | A USL file. |
| BASD | 1025 | A BASIC/3000 data file. |
| BASP | 1026 | A BASIC/3000 program file. |
| BASFP | 1027 | A BASIC/3000 fast program file. |
| RL | 1028 | A relocatable library (RL) file. |
| PROG | 1029 | A program file. |

| Mnemonic | HP-defined Integer | Defines the File as: |
|----------|--------------------|-----------------------|
| STAR | 1030 | A STAR/3000 file. |
| SL | 1031 | A segmented library (SL) file. |

If this parameter is omitted from the :FILE command, and from the FOPEN intrinsic call that opens the file, the default value is 0. (Optional parameter.)

*filesize*    The total maximum file capacity, specified only for a NEW file, in terms of physical records (for files containing variable-length and undefined-length records), and logical records (for files containing fixed-length records). Must be a double-word integer. The default value is 1023. The maximum capacity allowed is 184,000 sectors. (The number of sectors in a file is found by the formula shown earlier under *FILE CHARACTERISTICS*.) (Optional parameter.)

*numextents*    The number of extents (integral number of contiguously-located disc sectors) that can be dynamically allocated to the file as logical records are written to it. The size of each extent (in terms of records) is determined by the *filesize* parameter value divided by the *numextents* parameter value. If specified, *numextents* must be an integer value from 1 to 16. The default is 8. (Optional parameter.)

*initalloc*    The number of extents to be allocated to the file *at the time it is opened*. This must be an integer from 1 to 16. The default value is 1. If an attempt to allocate the requested space fails, the FOPEN intrinsic that opens the file returns an error condition code to the user's program when that program runs. (Optional parameter.)

*\*formaldesignator₁*    The name of a file defined in a previous :FILE command, preceded by an asterisk.


## Accessing Files Already in Use

When a user's process attempts to access a file already being accessed by another process, the action taken by MPE/3000 depends on the current use of the file, as shown in Figure 5-1.

REQUESTED ACCESS GRANTED, UNLESS NOTED

| Requested Access | Current Use | FOPENed for Input | | FOPENed for Output | | FOPENed for Input/Output | | Program File Loaded | Being :STOREd | Being :RESTOREd |
|---|---|---|---|---|---|---|---|---|---|---|
| | | SHR | EAR | SHR | EAR | SHR | EAR | | | |
| FOPEN for Input | SHR | Requested Access Granted | Requested Access Granted | Requested Access Granted | Requested Access Granted | Requested Access Granted | Requested Access Granted | Requested Access Granted | Requested Access Granted | Error 91 |
| | EAR | Requested Access Granted | Requested Access Granted | Error 90 | Error 90 | Error 90 | Error 90 | | | |
| FOPEN for Output | SHR | Requested Access Granted | Error 91 | Requested Access Granted | Error 91 | Requested Access Granted | Error 91 | Error 91 | Error 91 | Error 91 |
| | EAR | Requested Access Granted | Error 91 | Error 90 | Error 90 | Error 90 | Error 90 | | | |
| FOPEN for Input/Output | SHR | Requested Access Granted | Input Granted | Requested Access Granted | Input Granted | Requested Access Granted | Input Granted | Input Granted | Input Granted | Error 91 |
| | EAR | Requested Access Granted | Input Granted | Error 90 | Error 90 | Error 90 | Error 90 | | | |
| :RUN,CREATE | | Requested Access Granted | | Error Message | | Error Message | | Requested Access Granted | Only if Loaded | Error Message |
| :STORE | | Requested Access Granted | | Error Message | | Error Message | | Requested Access Granted | Error Message | Error Message |
| :RESTORE | | Error Message | | Error Message | | Error Message | | Error Message | Error Message | Error Message |

NOTES:
1. SHR = Share; EAR = Exclusive, allow reading.
2. Fully exclusive accesses cause any succeeding access (except :STORE) to fail.
3. Append access treated like output; Update treated like input/output.
4. Error 90 = Calling process requested exclusive access to a file to which another process has access.
5. Error 91 = Calling process requested access to a file to which another process has exclusive access.

Figure 5-1. Actions Resulting From Multi-access of Files

## Re-Specifying File Names

One common application of the :FILE command is to allow the user to reference, within his program, files whose names and characteristics he may not know at the time he writes his program.


*EXAMPLE:*

*Suppose, for example, that the user is writing a generalized program that reads input from a different file each time it is executed. Within the program, the user could reference the input file by the formal designator INPUT, and an output file by the formal designator OUTPUT. After the program is compiled and prepared on disc, the user could issue :FILE commands to equate INPUT and OUTPUT with the corresponding actual file designators of the files to be used on this run. Then, he could issue the command to execute the program. Suppose that the user wanted to execute such a program (called PROG1) with an actual file named MYFILE used as input. He also wants to designate a disc file actually named TEMP for output. To do this, he could enter the following:*


```
    •
    •
    •
:FILE INPUT=MYFILE
:FILE OUTPUT=TEMP
:RUN PROG1
    •
    •
    •
```

**Passing Files**

Another example illustrates how the :FILE command can be used to pass files between programs.

*EXAMPLE:*

*In this example, two programs, PROG1 and PROG2, are executed. PROG1 receives input from the actual disc file DSFIL (through the programmatic name SOURCE1) and writes output to an actual file $NEWPASS, to be passed to PROG2. ($NEWPASS is referenced programmatically in PROG1 by the name INTERFIL.) When PROG2 is run, it receives $NEWPASS (now known by the actual designator $OLDPASS), referencing that file programmatically as SOURCE2. (Note that only one file can be designated for passing.)*

```
      •
      •
      •
:FILE SOURCE1=DSFIL
:FILE INTERFIL=$NEWPASS
:RUN PROG1
:FILE SOURCE2=$OLDPASS
:RUN PROG2
      •
      •
      •
```

## Issuing Detailed File Specifications

The user can use the :FILE command to designate, for a file, various detailed specifications (such as device type, file type, code, or access mode). This command overrides any existing specifications defined for the file in the user's program.

*EXAMPLE:*

*Suppose a BASIC/3000 user is accessing MPE/3000 from a terminal. He knows that during the course of his session, he will want to list his program and its output on a line printer. Before calling the BASIC/3000 Interpreter, he uses the :FILE command to specify a file for printout as follows:*

```
                  .
                  .
                  .
     :FILE PRINTER; DEV=LP
     :BASIC ,,*PRINTER
                  .
                  .
                  .
     >10 FOR I = 1 TO 10
                  .
                  .
                  .
     >LIST, 10-50
                  .
                  .
                  .
```

*The :FILE command will direct the BASIC/3000 program listing and output to a line printer file. (Once the user has entered BASIC, he may transmit output from his program to the line printer by entering the LIST command.)*

All :FILE commands must precede the :RUN or subsystem execution commands to which they apply. Each :FILE command remains active throughout the entire job or session unless that command is revoked or superceded. For this reason, the same program cannot be executed twice within the same job if that program references different files each time — *unless* a new :FILE command or a :RESET command is issued prior to the second execution.

As another example of this technique:

*EXAMPLE:*

*Two FORTRAN/3000 programs are to be executed within one job. Before the first program is executed, a new file must be created with the following specifications: a binary disc file, direct access, with fixed-length records of 100 words each. Maximum size is 5000 records, divided into 10 extents, but only one extent (500 records) is to be allocated initially. The initial FORTRAN/3000 program expects this file to be accessed as logical unit number 8. The user wishes to save the file under the name FDATA1. To specify this file, the user enters:*

```
:FILE FTN08=FDATA1, NEW; SAVE; REC=100,, F, BINARY; DISC=5000, 10, 1
```

*This file will be created when opened by the :FORTRAN program; it will be permanently saved under the user's log-on group when closed by the FORTRAN program.*

*The first FORTRAN program may now be executed, accessing this file for disc input/output. Before the second program is executed, however, the :FILE command must be used again since the second program expects to access this file through logical unit numbers 20 and 21. Rather than re-enter the above specification, which is quite lengthy, the user simply enters these references and then executes his program:*

```
:FILE FTN20=FDATA1
:FILE FTN21=*FTN20       (OR,   :FILE FTN21=FDATA1)
```

## Implicit :FILE Commands

As mentioned earlier, compilers and other subsystems that run under MPE/3000 accept actual file designators as parameters in the commands that call these programs. Although the user is not generally aware of this fact, when an actual file designator appears as a command parameter, it is *automatically* equated to a formal file designator, used within the subsystem, by an *implicit :FILE command* issued by the command executor. For instance, within the FORTRAN/3000 compiler, the formal file designator for the *textfile* input is FTNTEXT. In the :FORTRAN command, this is related to the *textfile* parameter shown below:

> :FORTRAN [textfile] [,[uslfile] [,[listfile] [,[masterfile] [,newfile] ] ] ]

When the user specifies:

        :FORTRAN ALSFILE

MPE/3000 implicitly issues the following :FILE command, invisible to the user:

        :FILE FTNTEXT=ALSFILE

When calling a compiler or subsystem, any valid actual file designators (except the formal designator used by the subsystem) can be used as command parameters, including those of the *\*formaldesignator* (back-reference) format.

*EXAMPLE:*

*In the following code, the user specifies a file on magnetic tape used as a source file during a FORTRAN compilation:*

        :FILE SOURCE=TAPE1,OLD; DEV=TAPE; REC=80
        :FORTRAN *SOURCE

*When these commands are encountered, the compiler executor issues the following implicit :FILE command, back-referencing the user's previous :FILE command:*

        :FILE FTNTEXT=*SOURCE

*When a compiler or subsystem terminates, MPE/3000 implicitely resets each formal designator previously set by an implicit :FILE command issued as a result of the compiler/subsystem call. This minimizes confusion between the subsystem's designator and the user's.*

## Command File Parameters

The programmer can avoid issuing :FILE commands that conflict with certain pre-defined :FILE commands, if he knows the formal file designators used by MPE/3000 subsystems and commands. The formal file designators for compilers, interpreter, and other subsystem commands are:

| Command | Parameters | Formal File Designator |
|---------|-----------|------------------------|
| :BASIC | *commandfile* | BASCOM |
| | *inputfile* | BASIN |
| | *listfile* | BASLIST |
| :FORTRAN | *textfile* | FTNTEXT |
| | *uslfile* | FTNUSL |
| | *listfile* | FTNLIST |
| | *masterfile* | FTNMAST |
| | *newfile* | FTNNEW |
| :SPL | *textfile* | SPLTEST |
| | *uslfile* | SPLUSL |
| | *listfile* | SPLLIST |
| | *masterfile* | SPLMAST |
| | *newfile* | SPLNEW |
| :COBOL | *textfile* | COBTEXT |
| | *uslfile* | COBUSL |
| | *listfile* | COBLIST |
| | *masterfile* | COBMAST |
| | *newfile* | COBNEW |
| :FORTPREP | *textfile* | FTNTEXT |
| | *progfile* | FTNPROG |
| | *listfile* | FTNLIST |
| | *masterfile* | FTNMAST |
| | *newfile* | FTNNEW |
| :SPLPREP | *textfile* | SPLTEXT |
| | *progfile* | SPLPROG |
| | *listfile* | SPLLIST |
| | *masterfile* | SPLMAST |
| | *newfile* | SPLNEW |
| :COBOLPREP | *textfile* | COBTEXT |
| | *progfile* | COBPROG |
| | *listfile* | COBLIST |
| | *masterfile* | COBMAST |
| | *newfile* | COBNEW |

| Command | Parameters | Formal File Designator |
|---|---|---|
| :FORTGO | *textfile* | FTNTEXT |
| | *listfile* | FTNLIST |
| | *masterfile* | FTNMAST |
| | *newfile* | FTNNEW |
| :SPLGO | *textfile* | SPLTEXT |
| | *listfile* | SPLLIST |
| | *masterfile* | SPLMAST |
| | *newfile* | SPLNEW |
| :COBOLGO | *textfile* | COBTEXT |
| | *listfile* | COBLIST |
| | *masterfile* | COBMAST |
| | *newfile* | COBNEW |
| :EDITOR | *listfile* | EDTLIST |
| :SEGMENTER | *listfile* | SEGLIST |
| :STAR | *listfile* | STRLIST |

The formal file designators for optional parameters in the commands that prepare and run programs are as follows. Implicit :FILE commands are not issued for these designators. Rather, the formal designator specifies the destination of listings generated by the command executor if the user does not re-specify the destination by issuing an *explicit* file command. (This :FILE command remains in effect throughout the job/session, unless an applicable :RESET command or another :FILE command is issued.)

| Command | Parameters | Formal File Designator |
|---|---|---|
| :PREP<br>:PREPRUN | PMAP | SEGLIST |
| :PREPRUN<br>:RUN | LMAP | LOADLIST |
| :RESTORE<br>:STORE | SHOW | SYSLIST |

## RESETTING A FORMAL FILE DESIGNATOR

A formal file designator referenced in a prior :FILE command can be reset to the meaning
defined by its original actual file designator. This is requested by issuing the :RESET command.
The :RESET command effectively nullifies any previous explicit or implicit :FILE command
referencing that formal designator, and applies to files on disc, tape, or any other device. The
format of the :RESET command is

$$\underline{:}RESET \left\{ \begin{array}{l} formaldesignator \\ @ \end{array} \right\}$$

*formaldesignator*        The formal file designator to be reset.

@        An indication that the formal file designators referenced in all
prior :FILE commands in the job/session are to be reset.

(Either the *formaldesignator* or @ parameter must be entered.)


*EXAMPLE:*

*Suppose that a user runs two programs, both referencing a file defined within these programs
by the designator DFILE, a new, temporary file on disc. Before he runs the first program, the
user wants to redefine the file so that it is output to the standard list device. To do this, he
issues a :FILE command equating DFILE with $STDLIST. In the second program, the file
is again to be a temporary file on disc. The user issues a :RESET command to re-establish
the programmatic specifications:*

```
:JOB JNAME, UNAME.ANAME
 •
 •
 •
:FILE DFILE=$STDLIST
:RUN PROG1
:RESET DFILE
:RUN PROG2
 •
 •
 •
:EOJ
```

## CREATING A NEW FILE

The programmer can create a new disc file and specify its characteristics by issuing the
:BUILD command. This command, when encountered, results in the *immediate* allocation
of an empty file having the specifications supplied by the user as command parameters.
(This is unlike the :FILE command, which does not take effect until the file is opened by
the user's program and whose specifications override those supplied by the FOPEN intrinsic.)
The user can specify the new file as a job/session temporary file, or as a permanent file.

To issue the :BUILD command, the user must have SAVE access (as defined later in this
section) to the group to which the new file is to belong (unless the TEMP parameter is
specified). The :BUILD command is written in this format:

:BUILD *filereference*

$$\left[;REC = \left[recsize\right] \left[, \left[blockfactor\right]\right] \left[, \begin{bmatrix} F \\ U \\ V \end{bmatrix} \left[\begin{array}{c} ,BINARY \\ ,ASCII \end{array}\right]\right]\right]$$

[;CCTL]

[;NOCCTL]

[;TEMP]

[;DEV = device]

[;CODE = filecode]

[;DISC = [filesize]  [,[numextents]  [,initalloc]]]

| | |
|---|---|
| *filereference* | The filename in the *filereference* format, optionally including the account and group identifiers, and lockword. If specified, the account name must be that of the log-on account. (Required parameter.) |
| *recsize*<br>*blockfactor*<br>*F*<br>*U*<br>*V*<br>*BINARY*<br>*ASCII*<br>*CCTL*<br>*NOCCTL*<br>*device*<br>*filesize*<br>*numextents*<br>*initalloc*<br>*filecode* | The same meanings and default values as the corresponding parameters described in the :FILE command discussion. (Optional parameters.) |

TEMP                          A specification that the file is a job *temporary* file, entered in
                              the job/session temporary file directory; when job/session
                              terminates, the file is deleted. If TEMP is omitted, the file is
                              declared *permanent*; it is saved in the system file domain.
                              (Optional Parameter.)


*EXAMPLE:*

*To create a new job/session temporary file named NFILE, containing 500 records of 160
words each, the user enters:*

> **:BUILD NFILE; DISC=500; REC=160; TEMP**


## SAVING A FILE

A temporary disc file can be changed to a permanent file by issuing the :SAVE command,
immediately following execution of the program that opens the file. (To use this command,
the user must have SAVE access to the group to which the referenced file is to belong.)

$$\underline{:SAVE} \left\{ \begin{array}{l} \$OLDPASS, newfilereference \\ \\ tempfilereference \end{array} \right\}$$


$OLDPASS                      The file currently being passed, used only if this is the file to be
                              made permanent. (After it is saved, no file is in the pass condition.)
                              (Optional parameter.)

*newfilereference*            The new file name to be assigned to $OLDPASS when it is made
                              permanent. (Required if $OLDPASS is used.)

*tempfilereference*           The name of the temporary file to be made permanent under
                              the same filename. This file is deleted from the temporary file
                              domain. (Optional parameter.)

If no account and group identifiers are entered as part of the *newfilereference* or *tempfile-
reference* parameters, the file will be assigned to the log-on account and group. (If the
account name is specified, it must be the name of the log-on account.)

*EXAMPLES:*

*The following command assigns the most recently $PASSed file to permanent status under the
name PERMFILE: (Now, $OLDPASS can no longer be referenced.)*

> **:SAVE $OLDPASS, PERMFILE**

*The next command changes the temporary file DATAFILE.GROUPX to permanent status.*

> **:SAVE DATAFILE.GROUPX**

## DELETING A FILE

To delete a disc file from the system, the user enters the :PURGE command:

    *:PURGE filereference[,TEMP]*

| | |
|---|---|
| *filereference* | The name (and, if required, the group, account and lockword) of the file to be deleted. This must be a file to which the user has write (W) access, as defined in the MPE/3000 security provisions. (Required parameter.) |
| *TEMP* | An indication that the file is a temporary job file. (Required parameter for temporary files.) |

*EXAMPLE:*

*The following command deletes the temporary file TFILE:*

    **:PURGE TFILE, TEMP**


## LISTING FILE SETS

The user can obtain descriptions of one or more disc files in the system by issuing the :LISTF command. This command provides a listing that shows, for each file referenced, various items requested at the user's option. Among these items are: the file name, the type, size, number of records it contains, and other information found in the file label. To obtain this information for a file, the user need not have access to the file. Therefore, when a user issues a command to list descriptions of several files, the resulting listing may contain information about some files that are not accessible to him.

The :LISTF command is issued in this format:

    *:LISTF [fileset] [,detail] [;listfile]*

| | |
|---|---|
| *fileset* | One or more files, referenced by filename, group, and account, as described below. If this parameter is omitted, all files in the user's log-on group are listed. (Optional parameter.) |
| *detail* | A number indicating the amount and type of information to be listed, as follows: |

        0   =    The filename. (An asterisk (*) denotes that the file is open.)

        1   =    The above, plus the file code, record size (bytes), record type (F, U, or V), whether ASCII or binary records (A or B), whether

carriage control option is taken (C, if so), the current end-of-file pointer, and the maximum number of records allowed in the file.

2 = The above, plus the blocking factor, number of disc sectors in use (including the file label and user headers), the number of extents currently allocated, the maximum number of extents allowed.

−1 = An octal listing of the file label (if the user has System Manager or Account Manager Capability). Account-Manager Users can issue only :LISTF, −1 for files in their account; they cannot specify an account in *fileset* for this detail. (The format of file labels is shown in Appendix F.)

A *detail* specification greater than 2 defaults to 2; a *detail* specification less than −1 defaults to −1.

The default parameter is 0. (Optional parameter.)

*listfile*        The file on which the listing is to be written. This must be an ASCII file from the output set. It is automatically specified as a new, ASCII file with variable-length records, user-supplied carriage control characters (CCTL), OUT access type, and the EXC option; all other specifications are the same as the :FILE command default specifications. If omitted, $STDLIST is assigned by default. (Optional parameter.)

The *fileset* parameter allows the user to request descriptions of one file alone, or various sets of files. It contains three positional-fields separated by periods: the file, group, and account fields. The file field permits the user to indicate a specific file or all files within the units designated by the other fields. The group field denotes the group to which the files belong. This can be the user's log-on group, any other group, or all groups within the accounts specified by the account field. The account field indicates the account or accounts to which the groups belong. This can be the log-on account, any other account, or all accounts in the system. (To specify *all* files, groups, or accounts, the user enters the character @ in the appropriate field. The omission of an entry in the group or account field denotes the log-on group or account.) For the *fileset* parameter, the following combinations of entries are possible.

| File Field | Group Field | Account Field | Entry Example | Meaning |
|---|---|---|---|---|
| *filename* | *groupname* | *accountname* | FILE.GROUP.ACCT | The file named, in the group and account designated. |
| *filename* | *groupname* | | FILE.GROUP | The file named, in the group designated under the log-on account. |
| *filename* | | | FILE | The file named, under the log-on group. |
| @ | *groupname* | *accountname* | @.GROUP.ACCT | All files in the group named, under the designated account. |
| @ | *groupname* | | @.GROUP | All files in the group named, under the log-on account. |
| @ | | | @ | All files in the log-on group. |
| @ | @ | *accountname* | @.@.ACCT | All files in all groups under the account named. |
| @ | @ | | @.@ | All files in all groups under the log-on account. |
| @ | @ | @ | @.@.@ | All files in the system. |

The following example shows how the complete :LISTF command is used:

*EXAMPLE:*

*To list the file name, file code, size, type of records, ASCII vs. binary code, carriage-control option, end-of-file pointer, and maximum number of records for the file named BASE in the group USERGP under the programmer's log-on account, the following command is issued:*

:LISTF BASE.USERGP, 1

*To list the filename of all files in all groups the user's log-on account, this command is entered.*

:LISTF @.@

## DUMPING FILES OFF-LINE

To obtain a back-up copy of a particular user disc file or fileset, the user can copy the fileset off-line to a magnetic tape unit by issuing the :STORE command. The files are copied in a special format, along with all descriptive information (such as account name, group name, and lockword), permitting them to be read back into the system later (by the :RESTORE command).

Multi-file reels and multi-reel files are both supported by :STORE and :RESTORE.

> *NOTE:* *The :STORE and :RESTORE commands are used primarily as a back-up for files. They can be used to interchange files between installations if the accounts, groups, and creators of the files to be restored are defined in the destination system. Furthermore, if no destination device is specified in the :RESTORE command, MPE/3000 does not guarantee which devices will actually receive the files — if a device of the same type as the original device with sufficient storage space cannot be found, the file is restored to any device that is a member of the device class DISC.*

The :STORE command is written as follows:

*:STORE   [filesetlist]  ;tapefile[;SHOW]*

*filesetlist*     A list of one or more files or filesets to be copied, written in this format:

*[fileset[,fileset] ...]*

In this list, *fileset* must be written as one of the combinations noted for *fileset* in the discussion of the :LISTF command. If the entire *filesetlist* parameter is omitted, the default value is @ (all files of the log-on group are copied). If any file requires a lockword, the user may supply that lockword. If he fails to supply it (while in batch mode), the file will not be stored. If he fails to supply it (in interactive mode), he is prompted for the lockword. (Optional parameter.)

*tapefile*        The name of the destination tape file onto which the stored files are written. This can be any magnetic tape file from the output set. This file must be referenced in the *\*formaldesignator* format described earlier. (Required parameter.)

*SHOW*        A request that the names of the stored files, plus the total number of files stored, the names of the files not stored, and the number of files not stored, be listed. If SHOW is omitted, only the total number of files stored, the names of the files not stored, and the number of files not stored, are listed. (Optional parameter.)

> *NOTE:*    *Before issuing a :STORE command, the user must identify*
> *tapefile as a magnetic tape file. He does this through the*
> *:FILE command. In this case, the :FILE command should*
> *be written in the following format, including no parameters*
> *other than those shown:*
>
> *:FILE formaldesignator  [=filereference]  ;DEV=device*
>
> *The device parameter must be the device class name or logical*
> *unit number of a magnetic tape unit.*
>
> *All other parameters for tapefile are supplied by the :STORE*
> *command executor; if the user attempts to supply any of these*
> *himself, the :STORE command is rejected.*

A typical user can dump any file to which he has read-access. If a file has a negative file code, however, the user must have the *Privileged Mode Optional Capability*. A user with the System Manager or System Supervisor Capability can dump any user file in the system. An Account-Manager User can dump any file in his account (but cannot dump those with negative file codes unless he also has the privileged mode capability).

Files currently open for output, input/output, update, or append access cannot be acted upon by a :STORE command. Files currently being stored or restored cannot be acted upon by a :STORE command. However, files loaded into memory (currently running programs) and files open for input only can be stored, since their contents cannot be altered.

While a file is being dumped, it is locked by MPE/3000 so that it cannot be altered or deleted until safely copied to tape. If the job performing the :STORE function is aborted (through the =ABORTJOB console command), some of the files being stored may remain locked until the next cold-load.

5-34

The flow chart below shows the checks performed against a file to ensure its eligibility for dumping:

```
                                  ┌─────────────────────┐
                                  │      Does user      │        Yes
                                  │ have System Manager ├──────────────────────────────────────────►
                                  │ or System Supervisor│
                                  │    Capability ?     │
                                  └──────────┬──────────┘
                                             │ No
                                             ▼
                                  ┌─────────────────────┐
                                  │      Does user      │
                                  │ have Account Manager │        Yes
                                  │ Capability and is this├────────────────────────────────────────►
                                  │    file in this     │
                                  │      Account?       │
                                  └──────────┬──────────┘
                                             │ No
                                             ▼
                                  ┌─────────────────────┐      Yes
                                  │         Is          ├──────────┐
                                  │    File Secured     │
                                  │          ?          │
                                  └──────────┬──────────┘
                                             │ No
┌──────────────────────────┐                ▼
│ File cannot be dumped.    │      No  ┌─────────────────────┐
│ Checking terminates for   │◄─────────┤        Read         │
│ this file, an error message│         │   Access Avail-     │
│ is printed, and command    │         │       able?         │
│ execution continues.       │         └──────────┬──────────┘
└──────────────────────────┘                      │ Yes
                                                   ▼
                           No    ┌─────────────────────┐
                        ◄────────┤      Does File      │
                                 │     have Lockword    │
                                 │          ?          │
                                 └──────────┬──────────┘
                                            │ Yes
                                            ▼
                                 ┌─────────────────────┐   No    ┌─────────────────────┐   No
                                 │         Is          ├────────►│   Interactive Mode  ├──────►
                                 │      Lockword       │         │          ?          │
                                 │      Supplied       │         └──────────┬──────────┘
                                 │          ?          │                    │ Yes
                                 └──────────┬──────────┘                    ▼
                                            │ Yes              ┌─────────────────────┐
                                            │                  │     Prompt  User    │
                                            │                  └─────────────────────┘
                                            ▼
                       No    ┌─────────────────────┐
                    ◄────────┤        Does         │
                             │      Lockword        │
                             │        Match         │
                             │          ?          │
                             └──────────┬──────────┘
                                        │ Yes
                                        ▼
                             ┌─────────────────────┐   Yes   ┌─────────────────────┐   No
                             │        Does         ├────────►│    Privileged       ├──────►
                             │  File have Nega-    │         │       Mode          │
                             │    tive File        │         │        ?            │
                             │      Code?          │         └──────────┬──────────┘
                             └──────────┬──────────┘                    │ Yes
                                        │ No                            │
                                        ▼
             Yes    ┌─────────────────────┐   No    ┌─────────────────────┐
          ◄─────────┤         Is          ├────────►│     Store File      │
                    │     File Busy       │         │     Off-Line        │
                    │          ?          │         └─────────────────────┘
                    └─────────────────────┘
```

After the tape is written, data showing the results of the :STORE command is printed. By default, this output is sent to the standard list device ($STDLIST). However, the user can override this default and transmit the output to another file by issuing a :FILE command equating SYSLIST (the formal file designator by which the :STORE command executor references this list file) to another file. For example, a user at a terminal might transmit this output to a line printer by entering

```
:FILE SYSLIST=MYFILE; DEV=LP
```

If the *SHOW* parameter is omitted from the :STORE command, only the total number of files actually stored, a list of files not stored, and a count of files not stored, are printed. But if *SHOW* is included, the listing of files appears, in the following format:

*FILES STORED* = *xxx*

| FILE | .GROUP | .ACCOUNT | LDN | ADDRESS |
|------|--------|----------|-----|---------|
| *filename1* | *.groupname1* | *.acctname1* | *ldn1* | *addr1* |
| *filename2* | *.groupname2* | *.acctname2* | *ldn2* | *addr2* |
| • | | | | |
| • | | | | |
| • | | | | |
| *filenamen* | *.groupnamen* | *.acctnamen* | *ldnn* | *addrn* |

*FILES NOT STORED* = *yyy*

| FILE | .GROUP | .ACCOUNT | FILESET | CONDITION |
|------|--------|----------|---------|-----------|
| *filename1* | *.groupname1* | *.acctname1* | *fileset#* | *msg* |
| *filename2* | *.groupname2* | *.acctname2* | *fileset#* | *msg* |
| • | | | | |
| • | | | | |
| • | | | | |
| *filenamen* | *.groupnamen* | *.acctnamen* | *fileset#* | *msg* |

In this format, *xxx* is a value denoting the total number of files dumped onto tape; *yyy* denotes the number of files requested that were not dumped. The notations *filename*, *groupname*, and *acctname* under the FILES STORED heading name the individual files dumped, and their groups and accounts, respectively. The notation *ldn* indicates the logical device number (in decimal) of the device on which the file resides, and *addr* is the absolute address (in octal) of the file label. The notations *filename*, *groupname*, and *acctname* under the FILES NOT STORED heading, indicate the individual files not dumped, and their groups and accounts. The notation *fileset#* shows the number of the fileset to which the particular file belongs (relative to its position in the *filesetlist* parameter). The notation *msg* is a message denoting the reason that the file was not dumped, as follows. (These errors do not abort the file storing operation, which continues.)

| Message | Meaning |
|---|---|
| ACCOUNT NOT IN DIRECTORY | Specified account does not exist. |
| GROUP NOT IN DIRECTORY | Specified group does not exist. |
| FILE NOT IN DIRECTORY | Specified file does not exist. |
| BUSY | File is open for output, or is currently being stored or restored. |
| FILE CODE < 0 AND NO PRIVMODE | A user without Privileged Mode Capability is attempting to STORE a file with a negative file code. |
| LOCKWORD WRONG | The file lockword either was not provided or was specified incorrectly. |
| READ ACCESS FAILURE | The user does not have read access to the specified file. |

The following catastrophic errors abort the :STORE command:

Command system error.

Disc input/output error (in system).

File directory error.

File system error on the tape file, list file, or temporary disc files used by the :STORE command executor.

The following example illustrates the use of the :STORE command.


*EXAMPLE:*

*To copy all files in the group GR4X (in the user's log-on account) to a tape file named BACKUP, the user enters the following commands. A listing of the files copied appears on the standard list device.*

```
:FILE BACKUP; DEV=TAPE
:STORE @.GP4X; *BACKUP; SHOW
```

Explicit or implicit redundancies are permitted in *filesetlist*, but once a file has been locked down, any subsequent reference to it in *filesetlist* results in the message BUSY even though execution of the :STORE command continues.


*EXAMPLE:*

*Suppose the file identified as FN.GN.AN is a member of the fileset referenced by @ in the following command.*

```
:STORE @,FN.GN.AN; *DUMPTAPE; SHOW
```

*The command is executed successfully, but the fileset# and msg notations under FILES NOT STORED on the listing will show:*

| filename | groupname | acctname | fileset# | msg |
|----------|-----------|----------|----------|-----|
| FN | GN | AN | 2 | BUSY |

*This same file will also be noted under FILES STORED.*

## RETRIEVING DUMPED FILES

The user can read back into the system, onto disc, any file, fileset, or filesetlist that has been stored off-line (on tape) by :STORE. The files referenced are attached to the appropriate groups and accounts, with previous account and group names, and lockwords all re-instated. File retrieval is requested with the :RESTORE command. This command does not create any new accounts or groups. Any tape file to be restored will only be restored if the account name and group name exist on disc (in the system directory).

   *:RESTORE tapefile [;[filesetlist] [;KEEP] [;DEV=device] [;SHOW] . . . ]*

| | |
|---|---|
| *tapefile* | The name of the tape file on which the *filesetlist* to be retrieved now resides. This file must be referenced in the *\*formaldesignator* format described earlier. A message is output to the console operator telling him which tape to mount, and the device it should be mounted on. (Required parameter.) |
| *filesetlist* | The file, fileset, or filesetlist to be restored from tape. Each file is restored as a permanent file in the system file domain. The parameter is written in the same format, and subject to the same constraints as the *filesetlist* parameter of the :STORE command. The number of filesets specified is limited as follows: up to 10 by account name; up to 15 by account name and group name; up to 20 by account name, group name, and file name. If the *filesetlist* parameter is omitted, the default value is @.@.@. (all files on the tape). (Optional parameter.) |
| *KEEP* | A specification that if a file referenced in *filesetlist* currently exists on disc, the file on disc is kept and the corresponding tape file is not copied into the system. If KEEP is omitted, and an identically-named file exists in the system, that file is replaced. If KEEP is omitted, AND a file on tape is eligible for restoring AND a file of the same name exists on disc AND this disc file is busy, the disc file is kept and the tape file is not restored. (Optional parameter.) |
| *device* | The device class name or logical device number of the device on which all files are to be restored. If omitted, an attempt is made to replace the files on the same device type (and subtype) on which |

5-38

they were originally stored in the system; if this cannot be done, an attempt is made to restore the files to the same device type (ignoring sub-type); if this fails, an attempt is made to restore it to the device class DISC; if this fails, the file is not restored. (Optional parameter.)

*SHOW*    A request that the names of the restored files, the total number of files restored, the names of the files not restored, and the total number of files not restored be listed. If *SHOW* is omitted, only the total number of files restored, a list of the files not restored, and a count of the files not restored, are listed. (Optional parameter.)

A user with System Manager or System Supervisor Capability can restore any file from a :STORE tape, assuming the account and group to which the file belongs, and the user who created the file exist in the system. A user with Account-Manager Capability can restore any file in his account (but cannot restore those with negative file codes unless he also has the privileged mode capability). Any other user can restore any tape file in his log-on account if he has save-access to the group to which the file belongs (but cannot restore those with negative file codes unless he also has the privileged mode capability). If the file on tape is protected by a lockword, the lockword must be supplied in the :RESTORE command. (Users logged-on interactively are prompted for omitted lockwords.)

If a copy of a file to be restored already exists on disc, the user must have write access to the disc file (since it will be purged by :RESTORE). If this disc copy has a negative file code, the user must have privileged mode capability to restore it.

Files *currently* open, loaded into memory, or being stored or restored, cannot be acted upon by a :RESTORE command.

The :RESTORE command performs the same checking performed by the :STORE command, to ensure a file's eligibility for retrieval. If the SHOW parameter is included in the :RESTORE command, a listing is produced showing the names of the files restored, a count of files restored, a list of files not restored, and a count of files not restored. Otherwise, a count of files restored, a list of files not restored, and a count of files not restored, are supplied. As with the listing produced by :STORE, the listing output by :RESTORE is transmitted to a file whose formal designator is SYSLIST; if the user does not specify otherwise, this file is equated, by default, to the standard list device ($STDLIST). The listing appears in the format shown below.

*FILES RESTORED = xxx*

| FILE | .GROUP | .ACCOUNT | LDN | ADDRESS |
|------|--------|----------|-----|---------|
| *filename1* | *.groupname1* | *.acctname1* | *ldn1* | *addr1* |
| *filename2* | *.groupname2* | *.acctname2* | *ldn2* | *addr2* |
| . | | | | |
| . | | | | |
| . | | | | |
| *filenamen* | *.groupnamen* | *.acctnamen* | *ldnn* | *addrn* |

*FILES NOT RESTORED = yyy*

| *FILE* | *.GROUP* | *.ACCOUNT* | *FILESET* | *CONDITION* |
|--------|----------|------------|-----------|-------------|
| *filename1* | *.groupname1* | *.acctname1* | *fileset#* | *msg* |
| *filename2* | *.groupname2* | *.acctname2* | *fileset#* | *msg* |
| . | | | | |
| . | | | | |
| . | | | | |
| *filenamen* | *.groupnamen* | *.acctnamen* | *fileset#* | *msg* |

In this format, *xxx* denotes the total number of files restored; *yyy* denotes the number of files requested that were not restored. The notations *filename, groupname,* and *acctname* under the FILES RESTORED heading name the individual files restored, and their groups and accounts, respectively. The notation *ldn* indicates the logical device number (in decimal) of the device on which the file now resides, and *addr* is the absolute address (in octal) of the file label. The notations *filename, groupname,* and *acctname* under the FILES NOT RESTORED heading, indicate the individual files not restored, and their groups and accounts. The notation *fileset#* shows the number of the fileset to which the particular file belongs (relative to its position in the *filesetlist* parameter.) The notation *msg* is an error message denoting the reason that the file was not restored, as follows. (These errors do not abort the file-restoring operation.)

| Message | Meaning |
|---------|---------|
| ACCOUNT DIFFERENT FROM LOGON | The file's account name is different from the name of the user's log-on account. (Users do not have save-access to groups outside of their log-on accounts.) |
| ACCOUNT DISC SPACE EXCEEDED | The account's disc space limit would be exceeded by restoring this file. |
| ACCOUNT NOT IN DIRECTORY | The account specified does not exist in the system. |
| ALREADY EXISTS | A copy of the file specified already exists on disc, and KEEP was also specified. The file was not replaced. |
| BUSY | The disc file is open, loaded, or being stored or restored at present. |
| CATASTROPHIC ERROR | A catastrophic error occurred while the system was restoring either this file or one previous to it on the tape, and the :RESTORE command was aborted. (Examples of such catastrophic errors are listed below.) |

| Message | Meaning |
|---|---|
| CREATOR NOT IN DIRECTORY | The creator of the file is not defined in the system. |
| DISC FILE CODE <0 AND NO PRIV MODE | One of the files (on disc) to be replaced has a negative file code, and the user does not have Privileged Mode Capability. |
| DISC FILE LOCKWORD WRONG | The disc file has a lockword that does not match the lockword for the file on tape. |
| GROUP DISC SPACE EXCEEDED | The group's disc space limit would be exceeded by restoring this file. |
| GROUP NOT IN DIRECTORY | The group specified does not exist in the system. |
| NOT ON TAPE | The file specified is not on the tape. |
| OUT OF DISC SPACE | There is insufficient disc space to restore this file. |
| SAVE ACCESS FAILURE | The user does not have save-access to the group to which the file belongs. |
| TAPE FILE CODE <0 AND NO PRIV MODE | One of the files (on tape) to be restored has a negative file code, and the user does not have Privileged Mode Capability. |
| TAPE FILE LOCKWORD WRONG | The tape file has a lockword that was not supplied by the user, or was specified incorrectly. |
| WRITE ACCESS FAILURE | The user does not have write-access to the copy of the file on disc. |

The following catastrophic errors abort the :RESTORE command:

Command syntax error.

Disc input/output error (in system).

File directory error.

File system error on the tape file (TAPE), list file (SYSLIST), or any of the three temporary files (GOOD, ERROR, and CANDIDAT) used by the :RESTORE command executor.

Improper tape; the tape used for input was not written in :STORE/:RESTORE format.

No continuation reel; the computer operator could not find a continuation reel for a multi-reel tape set.

Device reference error; the specification for the *device* parameter is illegal, or the device requested is not available.

Too many *filesets* specified.

*EXAMPLE:*

*To retrieve from the file named BACKUP all files formerly belonging to the user's log-on group, the user enters:*

```
:FILE BACKUP; DEV=TAPE
:RESTORE *BACKUP; @; KEEP; DEV=MDISC; SHOW
```

*If a tape file satisfying the @ specification already exists in the system, it is not restored.*

> *NOTE:* *Tapes created by the :STORE command and :SYSDUMP command (discussed in HP 3000 System Manager/System Supervisor Capabilities (03000-90038)), are compatible. Thus, a tape dumped through :SYSDUMP can be used as input for the :RESTORE command.*

## RENAMING A FILE

The user can change the name (*filereference*-format identity) of any disc file which he has created. When he does this, he effectively removes the file with the old name from the system and creates another file with identical contents and a new name. This command can be used also to move a file from one group to another by specifying different group names in the first two parameters, or to change the lockword of a file. Renaming is accomplished with the :RENAME command:

*:RENAME oldfilereference,newfilereference[,TEMP]*

oldfilereference      The current name of the file including optional group and account identifiers, and lockwords. (If an account is specified, it must be the log-on account.) (Required parameter.)

newfilereference      The new name to be assigned to the file, including optional group and account identifiers, and lockwords. If an account identifier is specified, it must be that of the log-on account.

If a group identifier is used, it must be one to which the user has save access, as defined under the discussion of file security. If the group and account identifiers are omitted, the log-on group and account are assumed. (Required parameter.)

*TEMP*                  Indicates that the old file was, and the new file will be, a temporary file local to the job or session. (Optional parameter.)


*NOTE:*   *The user can apply the :RENAME command only to files that he himself has created.*


*EXAMPLES:*

*To change the name of a temporary job file from OLDFILE to NEWFILE. ORGB, the user enters:*

```
:RENAME OLDFILE, NEWFILE.ORGB, TEMP
```

*To change the lockword of the permanent file XFILE from LKD to BARX, the user enters:*

```
:RENAME XFILE/LKD, XFILE/BARX
```


## SPECIFYING FILE SECURITY

As previously noted, when a user logs onto the system (or submits a job to it through the computer operator), he is related to an account and to a group of files owned by that account. Associated with each account, group, and individual file, there is a set of security provisions that specifies any restrictions on access to the files in that account or group, or to that particular file. (Notice that these provisions apply to disc files only.) These restrictions are based upon two factors:

1.  Modes of Access (reading, writing, or saving, for example).

2.  Types of Users (users with Account Librarian or Group Librarian Capability, or creating users, for instance) to whom the access modes specified are permitted.

The security provisions for any file describe *what modes of access* are permitted to *which users* of that file.

The access modes possible, the mnemonic codes used to reference them in MPE/3000 commands relating to file security, and the complete meanings of these modes are listed below:

| Access Mode | Mnemonic Code | Meaning |
| --- | --- | --- |
| Reading | R | Allows users to read files. |
| Locking | L | Permits a user to prevent concurrent access to a file by himself and another user. Specifically, it permits use of the FLOCK and FUNLOCK intrinsics, and the exclusive-access option of the FOPEN intrinsic, all described in the next section. |
| Appending | A | Allows users to add information and disc extents to files, but prohibits them from altering or deleting information already written. This access mode implicitly allows the locking (L) access mode described above. |
| Writing | W | Allows users general writing access, permitting them to add to, delete, or change any information on files. This includes removing entire files from the system with the :PURGE command. Writing access also implicitly allows the locking (L) and appending (A) access modes described above. |
| Saving | S | Allows users to declare files *within their group* permanent, and to rename such files. This ability includes the creation of a new permanent file with the :BUILD command. |
| Executing | X | Allows users to run programs stored on files, with the :RUN command or CREATE intrinsic. |

The types of users recognized by the MPE/3000 security system, the mnemonic codes used to reference them, and their complete definitions are listed below.

| User Type | Mnemonic Code | Meaning |
| --- | --- | --- |
| Any User | ANY | Any user defined in the system; this includes all categories defined below. |
| Account Librarian User | AL | User with Account Librarian Capability, who can manage certain files within his account that may or may not all belong to one group. |

| User Type | Mnemonic Code | Meaning |
|---|---|---|
| Group Librarian User | GL | User with Group Librarian Capability, who can manage certain files within his home group. |
| Creating User | CR | The user who created this file. |
| Group User | GU | Any user allowed to access this group as his log-on or home group, including all GL users applicable to this group. |
| Account Member | AC | Any user authorized access to the system under this account; this includes all AL, GU, GL, and CR users under this account. |

Users with System or Account Manager Capability bypass the standard security mechanism; a System Manager User always has R, A, W, L, X access to any file in the system, and S access to any group in his account; an Account Manager User always has unlimited (R, A, W, L, X) access to any file in his account, and S access to any group in his account.

The user-type categories that a user satisfies depend on the file he is trying to access. For example, a user accessing a file that is not in his home group is not considered a group librarian for this access even if he has the Group Librarian User Attribute.

Notice that in order to extend a file, either W or A access to that file is required.

> *NOTE:* *In addition to the above restrictions, in force at the account, group, and file level, a file lockword can be specified for each file. Users must then specify the lockword as part of the filename to access this file. The way in which lockwords are assigned to files is discussed earlier in this section.*

The security provisions for the account and group levels are managed only by users with the System Manager and the Account Manager Capabilities respectively, and can only be changed by those individuals. The manner in which they are implemented is described in *HP 3000 System Manager/Supervisor Capabilities*. Because they also relate to the security provisions at the file level (which are the responsibility of the standard user), the account and group level provisions are also summarized in this section.

## Account-Level Security

The security provisions that broadly apply to all files within an account are set by a System Manager User when he creates the account. The initial provisions can be changed at any time, but only by that user.

At the account level, five access modes are recognized:

Reading (R)

Appending (A)

Writing (W)

Locking (L)

Executing (X)

Also, at the account level, two user types are recognized:

Any User (ANY)

Account Member (AC)

If no security provisions are explicitly specified for the account, the following provisions are assigned by default:

- For the system account (named SYS), through which the System Manager User initially accesses the system, reading and executing access are permitted to all users; appending, writing, and locking access are limited to account members. (Symbolically, these provisions are expressed as follows:

    R,X:ANY;  A,W,L:AC

    In this format, colons are interpreted to mean ": . . . is permitted only to . . .", or ". . . is limited to . . .". Commas are used to separate access modes or user types from each other. Semicolons are used to separate entire access mode/user type groups from each other.)

- For all other accounts, the reading, appending, writing, locking, and executing access are limited to account members. (R,A,W,L,X:  AC).

## Group-Level Security

The security provisions that apply to all files within a group are initially set by an Account Manager User when he creates the group. They can be equal to or more restrictive than the provisions specified at the account level. (The group's security provisions also can be less restrictive than those of the account — but this effectively results in *equating* the group restrictions with the account restrictions, since a user failing security checking at the account level is denied access at that point, and is not checked at the group level.) The initial group provisions can be changed at any time, but only by an Account-Managing User for that group's account.

At the group level, six access modes are recognized:

Reading (R)

Appending (A)

Writing (W)

Locking (L)

Executing (X)

Saving (S)

Also, at the group level, five user types are recognized:

Any User (ANY)

Account Librarian User (AL)

Group Librarian User (GL)

Group User (GU)

Account Member (AC)

If no security provisions are explicitly specified, the following provisions apply by default.

- For a public group (named PUB), whose files are normally accessible in some way to all users within the account, reading and executing access are permitted to all users; appending, writing, saving, and locking access are limited to Account Librarian Users and Group Users (including Group Librarian Users). (R,X:ANY; A,W,L,S:AL,GU).

- For all other groups in the account, reading, appending, writing, saving, locking, and executing access are limited to group users. (R,A,W,L,X,S:GU).

## File-Level Security

When a file is created, the security provisions that apply to it are the default provisions assigned by MPE/3000 at the file level, coupled with the user-specified or default provisions assigned to the account and group to which the file belongs. At any time, however, the creator of the file (and *only* this individual) can change the file-level security provisions. Thus, the total security provisions for any file depend upon specifications made at all three levels — the account, group, and file levels. A user must pass tests at all three levels — account, group, and file security, in that order — to successfully access a file in the requested mode.

If no security provisions are explicitly specified by the user, the following provisions are assigned at the file level by default:

- For all files, reading, appending, writing, locking, and executing access are permitted to all users. (R,A,W,L,X:ANY).

Because the total security for a file always depends on security at all three levels, a file not explicitly protected from a certain access mode at the file level may benefit from the default protection at the group level. For example, the default provisions at the file level allow the file to be read by any user — but the default provisions at the group level allow access only to group users. Thus, the file can only be read by a group user.

In summary, the default security provisions at the account, group, and file levels combine to result in these *overall* default security provisions:

| Filereference | File | Access Permitted | Save Access to Group |
|---|---|---|---|
| filename.PUB.SYS | Any file in Public Group of System Account. | (R,X:ANY; W:AL,GU) | AL,GU |
| filename.group-name.SYS | Any file in any group in System Account. | (R,W,X:GU) | GU |
| filename.PUB.ac-countname | Any file in Public Group of any account. | (R,X:AC; W:AL,GU) | AL,GU |
| filename.group-name.accountname | Any file in any group in any account. | (R,W,X:GU) | GU |

Stated another way, when the default security provisions are in force at all levels, the standard user (without any other user attributes) has:

- Unlimited access (in all modes) to all files in his log-on group and home group.

- Reading and executing access (only) to all files in the public group of his account and the public group of the System Account.

The user cannot access any other file in the system (in any mode).

A user can only create files within his own account.

The legitimacy of a request to access a file is determined by checking the access mode requested by the user against the final access mode authorized for users of this type by the file security provisions. For various intrinsics, the functions requested versus the access mode necessary for the functions to be honored, are shown below. (The intrinsics are described in detail in Section VI.)

| Intrinsic | Function Requested | Access Mode Required to Honor Function |
|-----------|--------------------|-----------------------------------------|
| FOPEN | Reading | R |
| | Writing | W |
| | Appending (only) | A (or W) |
| | Input/output | R and W |
| | Updating | R and W |
| | Exclusive accessing | L (or W or A) |
| | Semi-exclusive accessing | L (or W or A) |
| FLOCK | File locking | L (or W or A) |
| FUNLOCK | File unlocking | L (or W or A) |
| FCLOSE | Saving | S (relative to desired group) |
| | Any access beyond the current end-of-file. | A or W |

Additionally, any attempt to access a file beyond the current end-of-file indicator requires permission for A or W access mode.

When a file is closed, with permanent file disposition, by the FCLOSE intrinsic with a *seccode* parameter of 0 for normal MPE/3000 security, the security provisions assigned are:

    R,A,W,L,X:  ANY

But when this is done with a *seccode* parameter of 1 for private user file security, the security provisions assigned are:

    R,A,W,L,X:  CR

Then, if the user desires, he can assign more or less restrictive security provisions with the :ALTSEC Command, described next.


## Changing File-Level Provisions

To change the security provisions assigned to any individual disc file, the creating user can enter the :ALTSEC command. This command permanently deletes all previous provisions specified for this file, and replaces them with those defined as the command parameters. The security provided by any *lockword*, however, is not affected. The :ALTSEC command format is:

    :ALTSEC filereference [;( [modelist:userlist [;modelist:userlist] . . . ] )]

where:

*filereference*          The name of the file whose security provisions are to be altered. This is any legal filename (including account and group names, if required.) The lockword, if any, must also be specified. (Required parameter.)

*modelist*               The modes of access that are permitted to the users specified in the immediately following *userlist* parameter. If two or more modes are specified, they must be separated from each other by commas. The modes are denoted by letters, as follows:

R   =   Reading

L   =   Locking

A   =   Appending (implicitly specifies L, also)

W   =   Writing (implicitly specifies A and L, also)

X   =   Executing

If any mode is omitted from this list, this implies (at the file level) that no one is permitted access in this mode. If no mode at all is specified, however, this implies (at the file level) completely unrestricted access to the file. Of course, access can also be limited by provisions specified at the account and group levels. (Optional parameter.)

> *NOTE:   Even though the creator of a file may be barred from accessing the file by the modelist:userlist restrictions, he can still issue an :ALTSEC command against that file, and thus change the security provisions to allow him access.*

*userlist*               The types of users to whom the access modes defined by the immediately-preceding *modelist* parameter apply. If two or more user types are specified, they should be separated by commas. The user types are specified as follows:

ANY   =   Any User.

AC   =   Account Member.

AL   =   Account Librarian User

GU   =   Group User

GL   =   Group Librarian User

CR   =   Creating User

(Required parameter if *modelist* is included.)

Note that more than one modelist:userlist parameter combination can be used, to permit extremely versatile file-security specifications.

If no *modelist:userlist* parameter combination is specified, the following default security provisions are assigned: (R,A,W L,X: ANY).

*EXAMPLES:*

*The following command alters the security provisions for the file named FILEX. This command permits the ability to read, execute, and append information to the file only to the creating user and the log-on or home-group users. (Notice that in the modelist:userlist parameter group, the separating colon can be interpreted as indicating ". . . is permitted only to . . .". Thus, the parameter group in this command implies "The Appending, Reading, and Executing Modes are permitted only to the Creating User and Log-On and Home Group Users.")*

        :ALTSEC FILEX (A,R,X: CR, GU)

*To restore the default security provisions to this file, the user would enter:*

        :ALTSEC FILEX

*The following :ALTSEC command changes the security provisions of :FILEX so that any group user can execute the file, but only the group librarian can read and write on it.*

        :ALTSEC FILEX (X:GU; R,W:GL)

## Suspending Security Provisions

From time to time, the creating user may wish to temporarily suspend all account, group, and file-level security provisions governing a disc file, to allow it to be accessed in any fashion by any user. (Note that this temporary suspension does not require the user to have the System or Account Manager Capability.) This is done with the :RELEASE command. (File lockword protection, however, is *not* removed by this command.) The command format is:

        *:RELEASE filereference*

where:

*filereference*            The name of the file whose security provisions are to be
                           suspended. (As part of *filereference*, the lockword, account,
                           and group may be specified.) (Required parameter.)

*EXAMPLE:*

*The following command releases the security provisions for the file FILEX in the user's log-on group.*

    :RELEASE FILEX

## Restoring Security Provisions

To restore the security provisions suspended by the :RELEASE command, the creating user enters the :SECURE command:

    *:SECURE  filereference*

where:

*filereference*          The name of the disc file whose security provisions are to be restored. (The filereference can include the lockword, account, and group, as needed.) (Required parameter.)

*EXAMPLE:*

*To restore the security provisions of FILEX, the user enters:*

    :SECURE FILEX

## LOCKING FILES

MPE/3000 allows several users to concurrently access a file. But occasionally, a programmer may want to prevent others from accessing such a file while he is performing some critical operation (such as updating) upon that file. To satisfy this need, MPE/3000 permits a user to lock out other users while he is accessing a file, on a continuous or a dynamic basis, by specifying certain parameters in the :FILE command or FOPEN intrinsic call that initiates file access. (These parameters and their use are described in the discussion of the FOPEN intrinsic call, in the next section.) To use these parameters, the programmer must be allowed the locking access mode noted earlier in this section.

## FILE MANAGEMENT COMMAND FILE-TYPE SUMMARY

The file commands described in this section and the types of files/devices to which they apply are summarized below:

| All<br>Devices | Disc<br>Only | Disc Input,<br>Tape Output | Tape Input,<br>Disc Output |
|---|---|---|---|
| :FILE | :BUILD | :STORE | :RESTORE |
| :RESET | :SAVE | | |
| | :PURGE | | |
| | :LISTF | | |
| | :RENAME | | |
| | :ALTSEC | | |
| | :RELEASE | | |
| | :SECURE | | |

# SECTION VI
## Accessing and Altering Files

Within a user's program, the accessing and modification of files is requested through intrinsic calls. Each file referenced is first opened through the FOPEN intrinsic call. Then, other operations such as reading, writing, updating, and spacing forward or backward can be performed upon the file with other intrinsic calls. Finally, the file is closed through the FCLOSE intrinsic call, issued by the user's process or by MPE/3000 when the user's process terminates.

If the user is programming in SPL/3000, he declares the intrinsics and writes the intrinsic calls as he does other statements within his program, as illustrated in the examples throughout this section. If he is programming in another language, such as FORTRAN/3000 or BASIC/3000, any intrinsics required are called automatically by the commands in that language, or are invoked through other provisions described in the manuals covering those languages.

In the FOPEN intrinsic call, the user references a particular file by its formal file designator, described in the preceding section. When the FOPEN intrinsic is executed, it returns to the user's process a *file number* by which the system uniquely identifies the file. The file number, rather than the file designator, is used by subsequent intrinsics in referencing the file. In an SPL/3000 program, the user obtains this number through the normal conventions of that language. One such convention employs an SPL/3000 assignment statement to store the file number into a location specified by an identifier (name) which can then be used as an intrinsic call parameter to reference the file. The format of the assignment statement is discussed in the manual covering SPL/3000. An example is shown below:

*EXAMPLE:*

*Suppose that a user issues an FOPEN call for a file designated FILLER. He could then store the file number assigned to FILLER in a location denoted by an identifier called F1 by writing an assignment statement. In this statement, the identifier appears in the left part and the FOPEN call appears in the right part. (FILLER is the identifier of a byte-array containing "FILLER.")*

```
F1:=FOPEN(FILLER);
```

*In subsequent intrinsic calls, the user refers to the file as F1. For example:*

```
FCLOSE(F1,0,0);
```

6-1

Each intrinsic is declared and called as described in Section III. In this manual, each intrinsic call is shown within the context of its complete declaration head format; the call is distinguished from the remainder of the format by a box. The notation OPTION VARIABLE in the intrinsic head format indicates that certain parameters are optional; the optional parameters are shown in bold-face type. The absence of this notation means that all parameters are required.

The condition codes returned to the user's program by the file system intrinsics have the following general meanings. The specific meanings, of course, depend on the intrinsic:

| Condition Code | Meaning |
|---|---|
| CCE | The function requested by the intrinsic call was completed successfully. |
| CCG | While servicing the request, MPE/3000 encountered the end of the file. |
| CCL | MPE/3000 could not service the request because an error occurred; corrective action may, in some cases, be taken. (By issuing an FCHECK intrinsic call, the user can have a more detailed error description transmitted to his process.) If, however, the error resulted from invalid parameters supplied by the user in the intrinsic call, the error is fatal and the user's process is aborted (or a software error trap, if previously enabled by the user, is activated). |

When a file is accessed by a process running a program written in a language other than SPL/3000, the file is generally (but not always) referenced by a file name. All intrinsic calls needed for opening, accessing, and closing the file are generated automatically by the user's process, and the file name is equated with the file number used by the intrinsics to reference the file.

When a new file is opened but not yet closed, it is always part of the job/session domain. At this time, the designator (SPL/3000 programs) or file name (programs in other languages) assigned by the user need not be unique. But when the file is saved, or closed without being deleted, MPE/3000 determines whether another file with the same designator name exists. If a name conflict occurs, a CCL condition code is returned to the user's process, and the specific error is made available through the FCHECK intrinsic. When a program aborts, old files are returned to the domain in which they were found when opened; new files are deleted.

> *NOTE:* All intrinsics discussed in this section, with the exception of FOPEN, FGETINFO, and FRENAME, can be called (in privileged mode) with the DB register pointing to a data segment other than the calling process' stack. All parameters referenced in any calls to these intrinsics are always accessed using the current DB-register setting.

## OPENING FILES

Before a user's process can read, write on, or otherwise manipulate a file, the process must initiate access to that file by opening it with the FOPEN intrinsic call. (This call applies to files on all devices.) When the FOPEN intrinsic is executed, it returns to the user's process the file number used to identify the file in subsequent intrinsic calls.

If the file is opened successfully (and the CCE condition code results), the file number returned is a positive integer ranging from 1 to 255. If the file cannot be opened (resulting in the CCL condition code), the file number returned is zero.

If a process issues more than one FOPEN call for the same file before it is closed, this results in multiple, logically-separate accesses of that file, and MPE/3000 returns a unique file number for each such access. Also, MPE/3000 maintains a separate logical record pointer (indicating the next sequential record to be accessed) for each such access.

In opening a file, FOPEN establishes a communication link between the file and the user's program by

- Allocating to the user's program the device on which the file resides. If the file resides on magnetic tape, FOPEN determines whether it is present in the system. (If it is not, FOPEN requests the system operator to supply the tape. Cataloging of tapes, however, is not done.) Generally, disc files can be shared concurrently among jobs and sessions. But magnetic tape and unit-record devices are allocated exclusively to the requesting job or session. For example, different processes within the same job may open and have concurrent access to files on the same magnetic tape or unit-record device; but this device cannot be accessed by another job until all accessing processes in this job have issued corresponding close requests (FCLOSE calls).

- Verifying the user's right to access the file under the security provisions existing at the account, group, and file levels.

- Determining that the file has not been allocated exclusively to another process (by the *exclusive* option in an FOPEN call issued by that process).

- Processing file labels (for files on disc). For new files on disc, FOPEN specifies the number of labels to be written.

- Allocating to the file the number of extents initially requested (for new disc files).

- Constructing the control blocks required by MPE/3000 for this particular access of the file. The information in these blocks is derived by merging specifications from four sources, listed below in descending order of precedence:

    1. The file label, obtainable only if the file is an *old* file on disc. This information overrides that from any other source. (Label formats are presented in Appendix F.)

    2. The parameter list of a previous :FILE command referencing the same formal file designator named in this FOPEN call, if such a command was issued in this job or session. This information overrides that from the two sources listed next.

3. The parameter list of this FOPEN intrinsic call.

4. System default values provided by MPE/3000 (when values are not obtainable from the above three sources).

When information in one of these four sources conflicts with that in another, pre-empting takes place according to the order of precedence shown above. To determine the specifications actually taking effect, the user can call the FGETINFO intrinsic, described later in this section. Notice that certain sources do not always apply or convey all types of information. (For instance, no file label exists when a new file is opened and so all information must come from the last three sources above.)

### Files On Non-sharable Devices

When a process opens a disc file, the user specifies whether the file is an old or new file; an old file is an existing, labeled file, and a new file implies that the file is to be created. When a process accesses a file that resides on a non-sharable device, the device's attributes may override the user's old/new specification. Specifically, devices used for input only (such as card readers) automatically imply *old* files; devices used for output only (such as line printers) automatically imply *new* files; serial input/output devices (such as teletype terminals and magnetic tape units) follow the user's old/new specifications.

When a job attempts to open an *old* file on a non-sharable device, MPE/3000 searches for the file in the Virtual Device Directory (VDD). If the file is not found, a message is transmitted to the console operator, asking him to locate the file by taking one of the following steps:

1. Indicate that the file resides on a device that is not in auto-recognition mode. No :DATA command is required—the operator simply allocates the device.

2. Make the file available on an auto-recognizing device, and allocate that device.

3. Indicate that the file does not exist on any device; the user's FOPEN request will be rejected.

When a job opens a *new* file on a non-sharable device (other than magnetic tape), the operator is not required to intervene. In these cases, the first available device is used. (A non-sharable device is considered directly available if it is not being used, or if it is being used by the requesting job and is requested by its logical device number.)

When a job opens a *new* file on a magnetic tape unit, operator intervention is always required; the operator must make the tape available.

## Record Formats

A file can contain records written in one of three formats: *fixed-length*, *variable-length*, and *undefined length*. These formats are described below:

*For fixed-length records*, physical records are blocks containing one or more logical records. The block size is determined by multiplying the block factor by the logical record size. (The block factor and logical record size are specified in the *blockfactor* and *recsize* parameters of the FOPEN intrinsic.) On any one file, fixed-length records are all the same size. A 128-word physical record (block) containing three 80-byte, fixed-length logical records is illustrated below:

<div align="center">

Physical Record
*(blockfactor x recsize)*

| Logical<br>Record 0 | Logical<br>Record 1 | Logical<br>Record 2 | ///// |
|---|---|---|---|

recsize= -80   recsize= -80   recsize= -80

Disc Sector

</div>
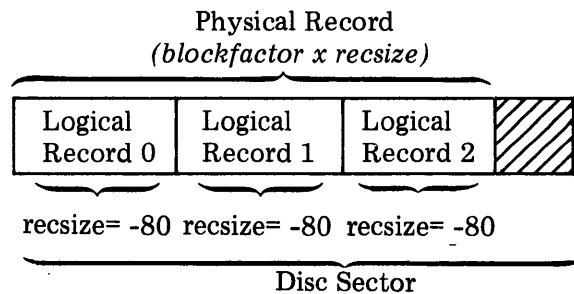
*For variable-length records* (as for fixed-length records), physical records are blocks containing one or more logical records--but, on any one file, the record size may vary from record to record. The block size is determined by multiplying the blockfactor by the *recsize* parameter specified by the user, and adding two words (reserved for file-system use).

Actual Blocksize = (blockfactor x recsize) +2
(in words)

In a block containing variable-length records, each logical record is preceded by a one-word *byte-count* showing the length of that record in bytes. The last record in the block is followed by a word containing "-1", acting as the *block terminator;* the next logical record encountered will be the first record in the next block. The block format is:

<div align="center">

Physical Record
[*(blockfactor x recsize)*+2]

| Byte<br>Count | Logical<br>Record 0 | Byte<br>Count | Logical<br>Record 1 | Byte<br>Count | Logical<br>Record 2 | -1 | |
|---|---|---|---|---|---|---|---|

</div>
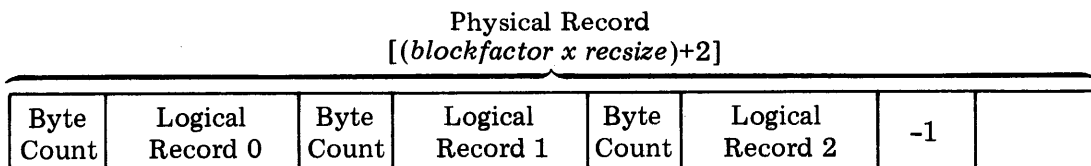
*For undefined-length records, physical records and logical records are synonymous*--that is, Physical Record A is the same as Logical Record A. For records of this type, the *recsize* parameter specified by the user denotes the size of the longest record to be transferred. The format of undefined records written to disc, with respect to the disc sectors occupied, can be illustrated by three cases in which the user-specified *recsize* is 256.
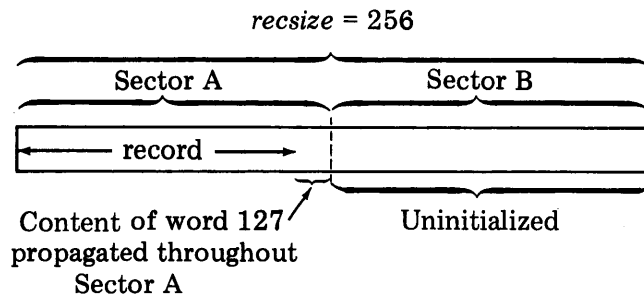
*Case 1:*    The user writes a record 256 words long.  The full record completely fills two disc sectors.

$$recsize = 256$$

| Sector A | Sector B |
|----------|----------|

|◄——————————— rec̦ord ———————————►|

words written = 256
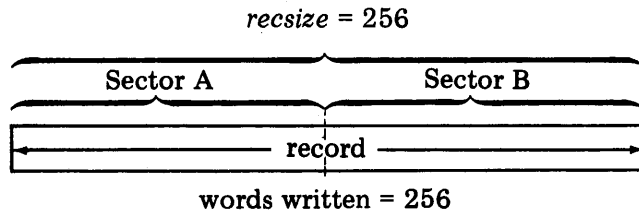

*Case 2:*    The user writes a record 129 words long.  The record written occupies all of Sector A and the first word of Sector B; the last word written is propagated throughout the remainder of Sector B. (The rule is: if (*reclength*) modulo 128 is not zero, then the last word written is propagated through the current sector.)

$$recsize = 256$$

| Sector A | Sector B |
|----------|----------|

|◄——————— record ———————►|

Content of word 129
propagated through
Sector B

words written = 129
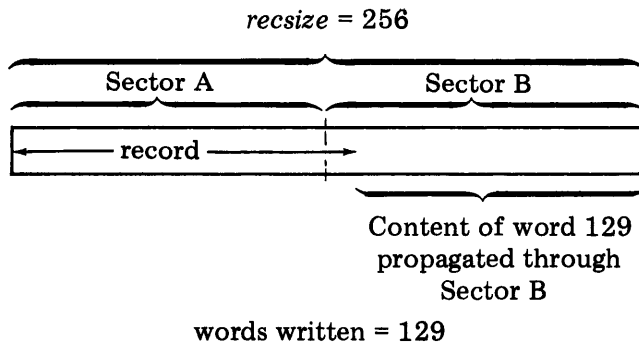

*Case 3:*    The user writes a record 127 words long.  The record written occupies 127 words of Sector A; the last word of the record is propagated throughout the remainder (word 128) of Sector A.  Sector B contains uninitialized data. (The rule is:  any sector not written into will remain uninitialized to 0 (binary files) or blanks (ASCII files).)

$$recsize = 256$$

| Sector A | Sector B |
|----------|----------|

|◄——— record ———►|

Content of word 127
propagated throughout
Sector A                    Uninitialized

The FOPEN intrinsic call is written in this format:

| INTEGER PROCEDURE | FOPEN | *(formaldesignator, foptions,* aoptions,recsize, device,formmsg,userlabels,blockfactor, numbuffers,filesize,numextents,initalloc, filecode*)* |
|---|---|---|

| VALUE | *foptions,* aoptions,recsize,userlabels,blockfactor,numbuffers, filesize,numextents,initalloc,filecode; |
|---|---|
| BYTE ARRAY | *formaldesignator,* device,formmsg; |
| LOGICAL | *foptions,* aoptions; |
| INTEGER | recsize,userlabels,blockfactor,numbuffers,numextents,initalloc, filecode; |
| DOUBLE | filesize; |

*OPTION VARIABLE,PRIVILEGED,EXTERNAL;*

This intrinsic returns (as the value of FOPEN) an integer file number used to identify the opened file in other intrinsic calls, (and changes the condition code in the status register as noted later in this discussion).

The FOPEN intrinsic parameters specify the elements shown below.

| *formaldesignator* | A byte-array containing a string of ASCII characters, interpreted as a formal file designator (as defined in Section V). This string must begin with a letter, contain alphanumeric characters, slashes, or periods and terminate with any non-alphanumeric character except a slash or a period. (If the string names a system-defined file, it can begin with a dollar sign ($); if it names a user-predefined file, it can begin with an asterisk (*).) This parameter can be omitted if *foptions* is present; the default value assigned is a temporary nameless file that can be read or written on, but not saved. |
|---|---|
| *foptions* | A 16-bit value that denotes a combination of file characteristics, including the type, recording code, and record format of the file. The meaning of the bit groups, and the way the user sets them are illustrated later. This parameter can be omitted if *formaldesignator* is present; as a default value, all bits are set to zero. |
| **aoptions** | A 16-bit value that denotes a combination of access options associated with the file. These options include restrictions on reading or writing on the file, or exclusive access provisions. The meaning of the bit groups and the way the user sets them are illustrated later. If this parameter is omitted, all bits are set to zero. |

| | |
|---|---|
| **recsize** | An integer indicating the size of the logical records in the file. If a positive number, this represents *words;* if a negative number, this indicates *bytes.* If the file is a new file, this value is permanently recorded in the file label. (If the records in the file are of undefined length, this value indicates the maximum size. For variable-length records, the maximum size is *recsize x blockfactor.*) The default value is the configured record width of the associated device. (The record size is always adjusted to a word unit.) |
| **device** | A byte array containing a string of ASCII characters terminating with any non-alphanumeric character except a slash or period, designating the device on which the file is to reside. The string may represent a device class name (up to eight alphanumeric characters beginning with a letter) or a logical device number (a three-byte numeric string) as described under the *device* parameter for the :FILE command. The default value is a byte-array containing the string DISC. (Device class names and logical device numbers are defined and assigned to devices during system configuration.) |
| **formmsg** | A byte array containing a forms message that can be used for purposes such as telling the console operator what type of paper to use in the line printer. This message must be displayed to the operator and verified before this file can be printed on a line printer. The message itself is a string of ASCII characters terminated by a period. If this parameter is omitted, no forms message is available. |
| **userlabels** | An integer specifying the number of user-label records to be written for this file. The default number is 0. |
| **blockfactor** | The size of each buffer to be established for the file, specified as an integer 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 x blockfactor*). For undefined-length records, *blockfactor* is always one logical record per block. The *blockfactor* value specified by the user may be overridden by MPE/3000. The default value is calculated by dividing the specified *recsize* into the configured physical record size; this value is rounded downward to an integer, but is never less than 1. Specification of a negative or zero value results in the default *blockfactor* setting. |
| **numbuffers** | An integer specifying the number of buffers to be allocated to the file. This parameter is not used for files representing interactive terminals, since a system-managed buffering method is always used in such cases. If omitted, or set to zero or a negative number, the default value is 2. |
| **filesize** | A double-word integer (as defined in SPL/3000) specifying the maximum file capacity in terms of physical records (for files containing variable-length and undefined-length records) and logical records (for files containing fixed-length records). The default value is 1023. A zero or negative value results in the default *filesize* setting. The maximum capacity allowed is 184,000 sectors. (The number of sectors in a file is found by the formula shown under FILE CHARACTERISTICS in Section V.) |

| | |
|---|---|
| **numextents** | An integer specifying the number of extents (integral number of contiguously-located disc sectors) that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *filesize* parameter value divided by the *numextents* parameter value. If specified, *numextents* must be an integer from 1 to 16. The default is 8. A zero or negative value results in the default *numextents* setting. |
| **initalloc** | An integer specifying the number of extents to be allocated to the file when it is opened (rather than dynamically, as needed). This must be an integer from 1 to 16. The default value is one. If an attempt to allocate the requested space fails, the FOPEN intrinsic returns an error condition code to the user's program. |
| **filecode** | An integer recorded in the file label and made available for general use to anyone accessing the file through the FGETINFO intrinsic call, described later. This parameter is used for new files only. For this parameter, any user can specify a positive integer ranging from 0 to 1023. (The default value is 0.) If a user's process is running in privileged mode, the user can specify a negative integer for *filecode* upon initially opening a file. Then, any future accesses of the file must be requested in privileged mode and must also specify the correct *filecode*. Certain positive integers beyond 1023 have particular HP-defined meanings |

These special integers are:

| Integer | Meaning |
|---|---|
| 1024 | A USL file. |
| 1025 | A BASIC/3000 data file. |
| 1026 | A BASIC/3000 program file. |
| 1027 | A BASIC/3000 fast program file. |
| 1028 | A relocatable library (RL) file. |
| 1029 | A program file. |
| 1030 | A STAR/3000 file. |
| 1031 | A segmented library (SL) file. |

## Foptions Parameter

The foptions parameter allows the user to specify six different file characteristics, by setting corresponding bit groupings in a 16-bit word. The correspondence is from right to left, beginning with Bit 15. If this parameter is omitted, all bits are set to zero. These characteristics are as follows, proceeding from the rightmost bit groups to the leftmost bit groups in the word. (The bit settings are also summarized, for the convenience of the user, in Figure 6-1.)

*NOTE:* *Bits groups are denoted using the standard SPL/3000 notation. Thus Bits (14:2) indicates Bits 14 and 15; Bits (10:3) indicates Bits 10, 11, and 12.*

| Bits | Characteristics |
|---|---|

(14:2)   *Domain Foption.* The file domain to be searched by MPE/3000 to locate the file, indicated by these bit settings:

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

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

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

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

(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 effects padding that can occur when a direct-write intrinsic call (FWRITEDIR) is issued to a record that lies beyond the current logical end-of-file indicator. In ASCII files, any dummy records between the previous end-of-file and the newly-written record are padded with blanks; in binary files, such records are padded with binary zeros. (All files not on disc or tape are treated as ASCII files.) For ASCII, this bit is 1; for binary, it is 0. The default value is 0.

(10:3)   *Default File Designator Foption.* The *actual* file designator equated with the formal file designator specified in FOPEN, if

1.   No explicit or implicit :FILE command equating the formal file designator to a different actual file designator is encountered in the job or session; *or*

2.   The *Disallow File Equation Foption* (below) is specified.

The bit settings are

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

001 =   The actual file designator is $STDLIST.

010 =   The actual file designator is $NEWPASS.

011 =   The actual file designator is $OLDPASS.

100 =   The actual file designator is $STDIN.

101 =   The actual file designator is $STDINX.

110 =   The actual file designator is $NULL.

| Bits | Characteristics |
|---|---|

(8:2)      *Record Format Foption.* The format in which the records in the file are recorded, indicated by these bits:

> 00 =    Fixed length records; the file is composed of logical records of uniform length.
>
> 01 =    Variable-length records; the file contains logical records of varying length. This format is restricted to sequential access only. The records are written sequentially and the size of each is recorded internally. (The actual record size used is determined by multiplying the *recsize* (specified or default) by the *blockfactor*, and adding two words reserved for system use. This option is not allowed when NOBUF is specified. In such a case, the record format used is *undefined-length records*, discussed below.)
>
> 10 =    Undefined-length records. The file contains records of varying length that were not written using the variable-length foption (01) just described. The undefined-length foption is typically used when reading a magnetic tape written under another operating system. All files not on disc or tape are treated as containing undefined-length records.

(7:1)      *Carriage Control Foption.* If selected, this specifies that the user will supply a carriage-control character in the calling sequence of each FWRITE call that writes records onto the file. (If *not* selected this specifies that the *control* parameter of the FWRITE intrinsic is ineffective and single-spacing prevails by default. In such a case, the user can embed control characters in his data.)

This *foption* prefixes each record written with the *right* control byte in the FWRITE control word. This prefix is interpreted and stripped when the data is sent to an output device, and thus remains invisible to the user unless he *saves* the file on disc or tape and subsequently reads it. In such a case, the prefix is included in the data read, and the effective *recsize* of the file is one word larger than the *recsize* originally specified. The bit settings for the carriage control foption are

> 1 = Carriage-control character expected.
>
> 0 = No carriage control-character expected.

Whenever an *old* file containing a carriage-control specification in its label is opened, MPE/3000 checks to ensure that the user also specified the *carriage-control foption* in his FOPEN intrinsic call.

Further information on carriage-control is presented in the discussion of the FREAD and FWRITE intrinsics later in this section.

(6:1)      This bit is reserved for system use.

(5:1)      *Disallow File Equation.* 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 disallowing :FILE, the bit is set to 1; for allowing :FILE, the bit is set to 0. The default value is 0.

(0:5)      These bits are reserved for system use.

| BITS | (0:5) | (5:1) | (6:1) | (7:1) | (8:2) | (10:3) | (13:1) | (14:2) |
|------|-------|-------|-------|-------|-------|--------|--------|--------|
| FIELD | (Unused) | Disallow :FILE | (Unused) | Carriage Control | Record Format | Default Designator | ASCII/ Binary | Domain |
| MEANING | | 1 = No :FILE<br>0 = :FILE | | 0 = NOCCTL<br>1 = CCTL | 00 = Fixed<br>01 = Variable<br>10 = Undefined | 000 = filename<br>001 = $STDLIST<br>010 = $NEWPASS<br>011 = $OLDPASS<br>100 = $STDIN<br>101 = $STDINX<br>110 = $NULL | 0 = Binary<br>1 = ASCII | 00 = New file<br>01 = Old System File<br>10 = Temporary File<br>11 = Old User File |

Figure 6-1. Foptions Bit Summary

The programmer can establish the bit settings desired through various SPL/3000 conventions. In one such convention, he enters the octal equivalents of the bit settings as the *foptions* parameter in the FOPEN call (unless he wishes the default foptions assigned). As with all octal values entered in SPL/3000, this parameter must be preceded by a percent sign to denote the octal base.

*EXAMPLE:*

*Suppose that the user wants to specify, for foptions, that the system file domain is to be searched for the referenced file (Bits 14-15 = 01), and that the default actual file designator is to be $STDIN (Bits 10-12 = 100), he would first determine the following complete bit setting, calculate its octal equivalent, and write this (preceded by %) as the foptions parameter in the FOPEN intrinsic call.*

| Bit No. | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit Setting | (Reserved for system use—always zeros.) | | | | | | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| Octal Equivalent | 0 | 0 | | 0 | | | 0 | | | 4 | | | 1 | | | |

Sign Bit ——————↑

*The value used for the foptions parameter is %000041. (The leading zeros are optional.)*

## Aoptions Parameter

The aoptions parameter permits the user to specify up to five different access options established by bit groupings in a 16-bit word. If this parameter is omitted, all bits are set to zero. These access options are as follows. (The bit settings are also summarized in figure 6-2.)

| Bits | Characteristics |
|---|---|

(12:4)  *Access Type Aoptions.* The type of access allowed users of this file:

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

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

0010 = Write-access only, but previous data in the file is *not* deleted. (The FREAD, FREEDSEEK, FUPDATE, and FREADDIR intrinsics cannot reference this file.)

| Bits | | | Characteristics |
|------|---|---|-----------------|

| | | | |
|------|---|---|---|
| 0011 | = | Append-access only. The FREAD, FREADDIR, FREAD-SEEK, FUPDATE, FSPACE, FPOINT and FWRITEDIR intrinsic calls cannot be issued for this file. | |
| 0100 | = | Input/Output access. Any file intrinsic except FUPDATE can be issued against this file. | |
| 0101 | = | Update access. All file intrinsics, including FUPDATE, can be issued for this file. | |
| 0110 | = | Execute access. Allows users with *Privileged Mode Capability* Input/Output access to any loaded file. | |

(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 record referenced, the remaining words or bytes are taken from subsequent successive records until the number specified by *tcount* have been transferred. This option is available only if the Inhibit-Buffering Aoption (below) is also selected. For multirecord mode, this bit is set to 1; for non-multirecord mode, it is set to 0. The default value is 0.

(10:1)     *Dynamic Locking Aoption.* Indicates that the user wants to use the FLOCK and FUNLOCK intrinsics to dynamically permit or restrict concurrent access to the file by other processes at certain times. The user's process can continue this temporary locking/unlocking until it closes the file. Dynamic locking/ unlocking is made possible through a resource identification number (RIN) assigned to the file and temporarily acquired by the FOPEN intrinsic. The calling process must use the RIN in cooperation with other processes also using it to guarantee the integrity of the file, as discussed in Section IX. Non-cooperating processes are allowed concurrent access at all times (unless other provisions prohibit this). The bit settings are

      1 = Allow dynamic locking/unlocking.

      0 = Disallow dynamic locking/unlocking.

A file may be multiply-accessed only if all FOPEN requests for the file specify dynamic locking, or if none of them do. An FOPEN request that disagrees with the current access (if any) will fail.

(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. This option is often used when performing some critical operation, such as updating the file. The bit settings allow these choices:

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

| Bits | Characteristics |
|------|-----------------|

10 = *Semi-Exclusive Access.* After the file is opened, prohibits con-current output access to this file through another FOPEN request, whether issued by this or another process, until this process issues the FCLOSE request or terminates. A subsequent request for the *input/output* or *update* access-type aoption will obtain read-only access. Other read-accesses, however, are allowed. If any process already has output access to the file when this FOPEN call is issued, a CCL error code is returned to the calling process. If another FOPEN call that violates the read-only restriction is issued while the *semi-exclusive* aoption is in effect, that call fails and an error code is returned to the calling process. The *semi-exclusive* access can only be requested by users allowed the file-locking access mode by the security pro-visions for the file.

11 = *Share Access.* After the file is opened, permits concurrent ac-cess to this file by any process, in any access mode (subject to other basic MPE/3000 security provisions in effect.)

00 = *Default Value*, relating to *Access-Type Aoption*. If the *read-access only* access-type aoption is selected, *share-access* (11) takes effect. Otherwise, *exclusive access* (01) takes effect.

(7:1)   *Inhibit-Buffering Aoption.* When selected, this option inhibits automatic buffering by MPE/3000 and allows input/output to take place directly between the user's stack and the applicable hardware device. This method is used in applications such as creation of large physical records on magnetic tape; it *always* applies to terminal input/output.

The *inhibit-buffering aoption* implies the following:

1. Record blocking is inhibited, so that logical records are treated as physical records.

2. Fixed-length or undefined-length records are treated as specified; variable-length records are treated as undefined-length records.

3. A request for the *carriage-control foption* is overridden (equivalent to specifying NOCCTL in the :FILE command).

Selection of this option equates the logical record size to the physical record (block) size until this process closes the file. In sequential accessing, the logical record pointer that indicates the position of the file, is incremented in terms of blocks rather than logical records. Additionally if the *recnum* parameter is specified in another intrinsic call (such as FREADDIR or FWRITEDIR), it is interpreted as a relative *block* number rather than a relative logical record number. When accessing this file through intrinsics containing *recsize* and *tcount* parameters, the user must specify each of their parameters as an even number of bytes (an integral number of words); if he specifies an odd number of bytes, the CCL error code is returned.

| BITS | (0:7) | (7:1) | (8:2) | (10:1) | (11:1) | (12:4) |
|---|---|---|---|---|---|---|
| FIELD | (Unused) | Inhibit Buffering | Exclusive Access | Dynamic Locking | Multi-record Access | Access Type |
| MEANING | | 1 = NOBUF<br>0 = BUF | 01 = Exclusive<br>10 = Semi-exclusive<br>11 = Share<br>00 = Default | 0 = No Dynamic Lock<br><br>1 = Dynamic Lock | 1 = Multi-record<br>0 = No multi-record | 0000 = Read only<br>0001 = Write only<br>0010 = Write (save) only<br>0011 = Append only<br>0100 = Read/write<br>0101 = Update<br>0110 = Execute |

Figure 6-2.  Aoptions Bit Summary

Since logical record requests issued when buffering is inhibited imply *physical* record requests, and since the logical record pointer is incremented in terms of *physical* records, the relationship of the user's file blocking to his input/ output requests is very important. For example, for a file that is blocked three records per block, each read request transfers only every third logical record. Thus, the *inhibit aoption should be used with caution since it can prevent certain data transfers, where records may be blocked in a manner unknown to the program accessing this file.*

The bit setting is

      1    =    To inhibit buffering.

      0    =    To allow normal buffering.

(0:7)        These bits are reserved for system use.

The bits in the aoptions parameter are set in the same manner as those in the foptions parameter.

The FOPEN intrinsic can return the following condition codes:

CCE        Request granted; the file was opened.

CCG        (Not returned by this intrinsic.)

CCL        Request denied. Another process already has exclusive or semi-exclusive access to the file, or an odd number of bytes was specified in the *inhibit-buffering* aoption. When the condition code CCL occurs, a file number of 0 is returned to the user's process.

*EXAMPLE:*

*To open a particular file, the user issues the following intrinsic call.*

```
FILEA := FOPEN (F1,%41,,REC,DV,,,,,7000D);
```

*The filenumber is returned to the word FILEA, and the condition code CCE results. The file characteristics are*

*formaldesignator*    = *FX22 (contained in the byte-array F1)*

*foptions*          = *Domain: system domain*
                 *ASCII/Binary: Binary*
                 *Default designator: $STDIN*
                 *Record format: Fixed-length*
                 *Carriage control: None*
                 *Disallow :FILE: Not active*

| | | |
|---|---|---|
| aoptions | = | *(Default Values)*<br>*Access type: Read access only*<br>*Multirecord: Non-multirecord mode*<br>*Dynamic locking: Disallowed*<br>*Exclusive: Share-access specified*<br>*Inhibit-buffering: Not selected* |
| recsize | = | *As specified by the value of REC.* |
| device | = | *DISKA, as specified in the byte-array DV* |
| formmsg | = | *None (default)* |
| userlabels | = | *None (default)* |
| blockfactor | = | *128/recsize (default)* |
| numbuffers | = | *1 (default)* |
| filesize | = | *7000 logical records* |
| numextents | = | *System default (8)* |
| initalloc | = | *1 (default)* |
| filecode | = | *0 (default)* |

## CLOSING FILES

To terminate access to a file, the user issues the FCLOSE intrinsic call. (This intrinsic applies to files on all devices.) This intrinsic deletes the buffers and control blocks through which the user's process accessed the file. It also deallocates the device on which the file resides. Additionally, it may change the disposition of the file. If the programmer does not issue FCLOSE calls for all files opened by his process, such calls are automatically issued by MPE/3000 when the process terminates. (When a file on tape is saved, the tape is rewound. New, saved tape files are not loaded, and remain off-line.)

The FCLOSE intrinsic call is written in this format:

*PROCEDURE* | *FCLOSE*      *(filenum, disposition, seccode);*

*VALUE filenum,disposition,seccode;*

*INTEGER filenum,disposition,seccode;*

*OPTION, EXTERNAL;*

The FCLOSE call parameters are shown below:

*filenum*          A word identifier supplying the file number of the file to be closed, through SPL/3000 conventions.

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

| Bit Nos. | Option |
|---|---|
| (13:3) | *Domain Disposition* |

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

1 = *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 already exists in the 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.

2 = *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, an error code is returned. If the file resides on magnetic tape, the tape is rewound but not unloaded.

3 = *Temporary Job File (Not Rewound).* This option has the same effect as Disposition Code 2, except that tape files are *not* rewound.

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

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

| (12:1) | *Disc Space Disposition* |
|---|---|

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

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

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

If more than one access is in effect for the file, its disposition is not affected until the last access terminates (with an FCLOSE call). Then, the effective disposition is the smallest non-zero disposition parameter specified among all FCLOSE calls issued against the file.

*seccode*        An integer denoting the type of security initially applied to the file, significant only for new permanent files. The options are:

> 0    Unrestricted access--the file can be accessed by any user, unless prohibited by current MPE/3000 security provisions.

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

The default *seccode* value is 0.

The following condition codes can be returned:

CCE        The file was closed successfully.

CCG        (This code is not returned.)

CCL        The file was not closed, perhaps because an incorrect *filenum* parameter was specified; or because another file with the same name and disposition exists in the system.

When a process is ended within a job, FCLOSE (0,0) is issued against all open files in the job temporary file domain. When a job is terminated, all job temporary files are purged.

*EXAMPLE*

*The following intrinsic call closes a file whose filenumber is supplied through the identifier FILEX. The file is to be saved as a permanent file (Disposition Code 1), having private creator file security.*

**FCLOSE (FILEX,1,1);**

*The CCE condition code is returned to the user's process.*


## READING SEQUENTIAL FILES

To read a logical record (or a portion of such a record) from a sequential file (on any device) to the user's data stack, the user issues the FREAD intrinsic call. The format of this call is

*INTEGER PROCEDURE*  | *FREAD (filenum,target,tcount)  ;*

*VALUE*        *filenum,tcount;*

*INTEGER*        *filenum,tcount;*

*ARRAY*        *target;*

*OPTION EXTERNAL;*

The FREAD intrinsic returns (as the value of FREAD) an integer showing the length of the information read.

The FREAD call parameters are

*filenum*          A word identifier supplying the filenumber of the file to be read, through SPL/3000 conventions.

*target*          The array to which the record is to be transferred.

*tcount*          An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies the length in *words*; if it is negative, it signifies the length in *bytes*; if it is zero, no transfer occurs. If tcount is less than the size of the record, only the first tcount bytes or words 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. (But if the *multirecord* aoption *was* specified, the remaining words in tcount will be read from the next successive records.)

When a record is read, the FREAD intrinsic returns to the user's program a positive value showing the length of the information transferred. If the *tcount* parameter in the FREAD call was positive, the value returned represents a *word* count; if the *tcount* parameter was negative, the value returned is a *byte* count. The logical record pointer is now set at the beginning of the next logical record in the file.

When the logical end-of-data is encountered during reading, the CCG condition code is returned to the user's process. On magnetic tape, the end-of-data can be denoted by a physical indicator such as a tape mark; on disc, it occurs when the last logical record written to the file is passed. If the file is imbedded in an input source containing MPE/3000 commands, the end-of-data is indicated when an :EOD command is encountered (but that command itself is not returned to the user). (The end-of-data is indicated, on $STDIN by any MPE/3000 command; on $STDINX, it is indicated by :JOB, :EOJ, :EOD, and :DATA.)

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



In any other case, the data read is only that requested by the user:
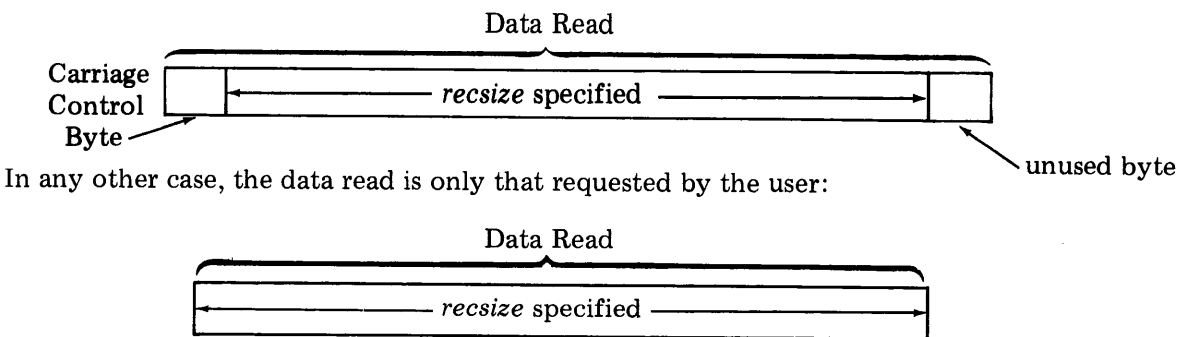
The condition codes possible are

CCE       The information was read.

CCG       The logical end-of-data was encountered during reading.

CCL       The information was not read because an error occurred; a terminal read was terminated by a special character (as specified in the FCONTROL intrinsic); or a tape error was recovered and the FSETMODE option was enabled.

*EXAMPLES:*

*To read a 20-word logical record from a sequential file whose filenumber is supplied through the identifier F1 to the array R1 in the stack, the user enters the following intrinsic call. The length of the information to be transferred is 10 words. The length of the information actually transferred is stored in the word COUNT.*

**COUNT := FREAD (F1,R1,20);**

*To read the first ten words from the next record in this file to the array R2, the user enters the following call. The length of the record transferred is stored in the word COUNTX.*

**COUNTX := FREAD (F1,R2,10);**

## READING DIRECT-ACCESS FILES

To read a logical record (or a portion of such a record) from a direct-access disc file to the user's data stack, the user enters the FREADDIR intrinsic call. This call may only be issued for a disc file composed of fixed-length or undefined-length records. The format of FREADDIR is

*PROCEDURE*     | *FREADDIR (filenum,target,tcount,recnum)*   ; |

*VALUE filenum,tcount,recnum;*

*INTEGER filenum;*

*ARRAY target;*

*INTEGER tcount;*

*DOUBLE recnum;*

*OPTION EXTERNAL;*

The FREADDIR call parameters are

*filenum*　　A word identifier supplying the filenumber of the file to be read, through SPL/3000 conventions.

*target*　　An array to which the logical record is to be transferred.

*tcount*　　An integer specifying the number of words or bytes to be transferred. If this value is positive, it signifies *words*; if negative, it denotes bytes; if zero, no transfer occurs. If tcount is less than the size of the record, only the first tcount bytes or words are read from the record. If tcount is larger than the actual size of the logical record and the *multirecord* aoption was *not* specified in FOPEN, the transfer is limited to the length of the logical record; otherwise, the excess words or bytes will be read from the next successive records.

*recnum*　　A double-word integer indicating the relative number, in the file, of the logical record to be read; the first record is indicated by 0D.

After the FREADDIR intrinsic is executed, the logical record pointer is now set at the beginning of the next logical record in the file.

As disc extents are initially allocated to a direct-access file, they are filled with binary zeros (for binary files) or ASCII blanks (for ASCII files). Thus, if a record is read that has never been written upon, binary zeros or ASCII blanks are transmitted to the user's stack.

The condition codes possible are

CCE　　The information specified was read.

CCG　　The logical end-of-data was encountered during reading.

CCL　　The information was not read because an error occurred.

*EXAMPLES:*

*To read the first logical record (containing 60 words) from a direct-access file whose file-number is supplied through the identifier FNUM, to a stack array named R19, the user enters the following. (In SPL/3000, a double-word integer is always indicated by the suffix D.)*

**FREADDIR (FNUM,R19,60,0D);**

*To read the first 20 bytes from logical record 24 in the same file, the user enters:*

**FREADDIR (FNUM,R19,-20,24D);**

## OPTIMIZING DIRECT-ACCESS FILE READING

The user can enhance the direct-access of disc files 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. FREADSEEK directs MPE/3000 to move the record from disc into a buffer in anticipation of the FREADDIR call, which subsequently moves the record directly to the stack.

> *NOTE:* *The FREADSEEK intrinsic call can only be issued against a file for which input/output buffering and fixed/undefined length record formats are in effect.*
>
> *The format of the FREADSEEK call is*

*PROCEDURE* | FREADSEEK (filenum,recnum)

*VALUE filenum,recnum;*

*INTEGER filenum;*

*DOUBLE recnum;*

*OPTION EXTERNAL;*

The FREADSEEK parameters are

*filenum*   A word supplying the filenumber of the file to be read, through SPL/3000 conventions.

*recnum*   A double-word integer indicating the relative number of the logical record to be read; the first record is indicated by 0D.

The condition codes possible are

CCE   The request was granted.

CCG   A logical end-of-file indication was encountered.

CCL   The request was not granted because of error.

*EXAMPLE:*

*To perform pre-read buffering on the third record of the file whose filenumber is supplied through the identifier ISAAC, the user enters:*

```
FREADSEEK (ISAAC,3D);
  •
  ••
  •
FREADDIR (ISAAC,STAK,60,3D);
```

6-24

## WRITING ON SEQUENTIAL FILES

To write a logical record (or a portion of such a record) from the user's stack to a sequential file, (on any device) the user issues the FWRITE intrinsic call:

PROCEDURE     | *FWRITE (filenum,target,tcount,control)  ;* |

*VALUE filenum,tcount,control;*

*INTEGER filenum,tcount;*

*ARRAY target;*

*LOGICAL control;*

*OPTION EXTERNAL;*

The FWRITE call parameters are

*filenum*       A word identifier supplying the filenumber of the file to be written on, through SPL/3000 conventions.

*target*        An array in the stack that contains the record to be written.

*tcount*       An integer specifying the number of words or bytes to be written from the record. If this value is positive, it signifies *words*; if it is negative, it signifies *bytes*; if zero, 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 recsize value, and the *multirecord* aoption was *not* specified in FOPEN, the FWRITE request is refused and condition code CCL is returned; otherwise, the excess words or bytes are written to the next successive records.

*control*      A logical value representing a carriage control code, effective if the file is transferred to a line printer or terminal. The options are

                     0 = To print the full record transferred, using single spacing. This results in a maximum of 132 characters per printed line.

                     1 = To use the first character of the data written to signify space control, and suppress this character on the printed output. This results in a maximum of 132 characters of data per printed line. (Permissible control characters are shown in Figure 6-3.)

Any other octal     = To use this code to determine space control, and print the full
code from             record transferred. This results in a maximum of 132 characters
Figure 6-3.         per printed line.

If the *control* parameter is not 1, and *tcount* is 0, only the space control is executed--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 6-4.

The user determines whether the carriage-control directive takes effect before printing (pre-space movement) or after printing (post-space movement), through the FCONTROL intrinsic, described later in this section.

When information is written to a fixed-length record (and *nobuff* 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.

When an FWRITE call writes a record beyond the current logical end-of-file indicator, this indicator is advanced to a further location.

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

The condition codes possible are

<blockquote>

CCE       The request was granted.

CCG       The physical bounds of the file prevented further writing; all disc extents were filled.

CCL       The request was not granted because an error occurred (such as *tcount* exceeding the size of the record in non-multirecord mode); or the FSETMODE option is enabled to signify recovered tape errors; or the end-of-tape marker was sensed.

</blockquote>

| Octal Code | ASCII Symbol | Carriage Action |
|---|---|---|
| %40 | " " | *Single-space. |
| %60 | " 0 " | *Double-space |
| %61 | " 1 " | Page-eject (form-feed). |
| %53 | " + " | No space, return (next printing at column 1). |
| %2nn | | Space nn lines. (No automatic page eject.) |
| (where n is any | | |
| digit from | | |
| 0 through 7) | | |
| %300 | | Page-eject (Tape Channel 1). |
| %301 | | Skip to bottom of form (Tape Channel 2). |
| %302 | | Single-spacing (with automatic page eject). (Tape channel 3.) |
| %303 | | Single-space on next odd-numbered line (with automatic page eject). (Tape Channel 4.) |
| %304 | | Triple-space (with automatic page eject). (Tape Channel 5.) |
| %305 | | Space ½ page (with automatic page eject). (Tape Channel 6.) |
| %306 | | Space ¼ page (with automatic page eject). (Tape Channel 7.) |
| %307 | | Space 1/6 page (with automatic page eject). (Tape Channel 8.) |
| %320 | | No space, no return. (Next printing physically follows this.) |
| %0 – %37 | | |
| %41 – %52 | | |
| %54 – %57 | | **Same as %40 |
| %62 – %77 | | |
| %310 – %317 | | |
| %321 – %377 | | |
| %400 | | Set post-space movement option; this first prints, then spaces. If previous option set was pre-space movement option, the driver outputs a line (and suppresses spacing) to clear the buffer. |
| %401 | | Set pre-space movement option; this first spaces, then prints. |
| %402 | | Set single-space option, with automatic page eject (60 lines per page). |
| %403 | | Set single-space option *without* automatic page eject (66 lines per page). |

\*     Spacing with or without automatic page eject can be selected.

\*\*    Future MPE/3000 requirements may necessitate redefinition of these octal codes.

Figure 6-3. Carriage-Control Directives

| FOPEN or :FILE | FWRITE *Control* Parameter | | |
|---|---|---|---|
| | = 0 | = 1 | = Greater than 1 |
| Carriage-Control Foption Specified or CCTL | Byte 1 <br> [ Ø \| record ] <br><br> Data output contains 132 characters; the first byte contains 0. | [ record ] <br><br> Data output contains 132 characters; the carriage control character in the first byte is suppressed. | Byte 1 <br> [ con-trol \| record ] <br><br> Data output contains 132 characters; the first character is a carriage-control character specified by the FWRITE *control* parameter. |
| Carriage Control Foption *not* specified or NOCCTL | [ record ] <br><br> Data output contains 132 characters. | [ record ] <br><br> Data output contains 132 characters. | [ record ] <br><br> Data output contains 132 characters. |

EFFECT ON DATA OUTPUT

Figure 6-4. Carriage-Control Summary

*EXAMPLE:*

*To write a 20-word logical record from the user's array INFO to the file designated by the identifier OUTGO, the user issues the following intrinsic call. When the record is printed, single spacing is used.*

**FWRITE (OUTGO,INFO,20,0);**

*To write the six bytes from the next record in the array NEWINFO, to the file designated by OUTGO2, the user writes the following. (The first character in the record is used as a control character.)*

**FWRITE (OUTGO2,NEWINFO,-6,1);**

## WRITING ON DIRECT-ACCESS FILES

To write a logical record (or a portion of such a record) from the user's stack to a direct-access disc file, the user issues the FWRITEDIR intrinsic call. This call may only be issued for a disc file composed of fixed or undefined length records. Its format is

*PROCEDURE*  | *FWRITEDIR (filenum,target,tcount,recnum) ;*

*VALUE filenum,tcount,recnum;*

*INTEGER filenum,tcount;*

*ARRAY target;*

*INTEGER tcount;*

*DOUBLE recnum;*

*OPTION EXTERNAL;*

The FWRITEDIR call parameters are

*filenum*     A word identifier supplying the filenumber of the file to be written on, through SPL/3000 conventions.

*target*      An array in the stack that contains the record to be written.

*tcount*      An integer specifying the number of words or bytes to be written from the record. If this value is positive, it signifies *words*; if it is negative, it signifies *bytes*; if zero, 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 *recsize* value and the *multirecord* aoption was not specified in FOPEN, the FWRITEDIR request is refused and condition code CCL is returned; otherwise, the excess words or bytes are written to the next successive records.

recnum        A double-integer indicating the relative number of the logical record to be written; the first record is indicated by 0D.

When information is written to a fixed-length record (and *nobuff* 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 FWRITEDIR intrinsic is executed, the logical record pointer is set to the record immediately following the record just written.

When a FWRITEDIR call writes a record beyond the current logical end-of-file indicator, this indicator is advanced to a further location. This can result in 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 (on a binary file) or with ASCII blanks (on an ASCII file).

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

The condition codes possible are

    CCE        The request was granted.

    CCG        The physical end-of-file was encountered during writing.

    CCL        The request was not granted because an error occurred.


*EXAMPLES:*

*To write logical record number 15 (containing 70-words) from the array DATA to the file identified by OUTX, the user writes:*

    FWRITEDIR (OUTX,DATA,70,15D);

*To write three words from the array NSTACK to record 16 in the file identified by OUTDATA, the user writes:*

    FWRITEDIR (OUTDATA,NSTACK,3,16D);

## READING LABELS

To read a user-defined label from a disc file, the user calls the FREADLABEL intrinsic. Before reading occurs, the user's *read-access* capability is verified. (Note that MPE/3000 automatically skips over any unread user labels when the first FREAD call is issued against a file.)

PROCEDURE | FREADLABEL *(filenum,target,* tcount,labelid);

*VALUE filenum,* tcount, labelid;

*INTEGER filenum,* tcount, labelid;

*ARRAY target;*

*OPTION VARIABLE, EXTERNAL;*

The FREADLABEL parameters are as follows: (Note that *tcount* and *labelid* are optional.)

| | |
|---|---|
| *filenum* | A word identifier supplying the filenumber of the file whose label is to be read, through SPL/3000 conventions. |
| *target* | An array (in the stack) to which the label is to be transferred. |
| tcount | An integer specifying the number of words to be transferred from the label. For disc files, *tcount* is limited to 128 words. If *tcount* is omitted, a default value of 128 words is assigned. |
| labelid | An integer specifying the label number required. If *labelid* is omitted, a default value of *0* is assigned. |

The possible condition codes are:

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

## WRITING LABELS

To write a user-defined label onto a disc file, the user calls the FWRITELABEL intrinsic. (This intrinsic does not overwrite nor destroy old user labels.)

*PROCEDURE*          | *FWRITELABEL (filenum, target, tcount,labelid);*

*VALUE filenum,* tcount,labelid;

*INTEGER filenum,* tcount,labelid;

*ARRAY target;*

*OPTION VARIABLE, EXTERNAL;*

The FWRITELABEL parameters follow. (Note that *tcount* and *labelid* are optional.)

*filenum*   A word identifier supplying the filenumber of the file to which the label is to be written, through SPL/3000 conventions.

*target*    An array in the stack from which the label is to be transferred.

tcount      An integer specifying, for disc files, the number of words to be transferred from the array. The default value is 128 words.

labelid     An integer specifying the number of the label required. If omitted, a default value of *0* is assigned.

The condition codes possible are:

CCE         The label was written.

CCG         The calling process attempted to write a label beyond the limit specified (in FOPEN) when the file was opened.

CCL         The label was not written because an error occurred.

# UPDATING FILES

To update a logical record residing on a direct-access device (disc), the user issues the FUPDATE intrinsic call. This affects the logical record last referenced by any intrinsic call for the file named; it 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. The FUPDATE call format is

PROCEDURE | FUPDATE (filenum,target,tcount) ;

*VALUE filenum,tcount;*

*INTEGER filenum,tcount;*

*ARRAY target;*

*OPTION EXTERNAL;*

The FUPDATE call parameters are

| | |
|---|---|
| *filenum* | A word identifier supplying the filenumber of the file to be updated, through SPL/3000 conventions. |
| *target* | An array in the stack that contains the record to be written in the updating. |
| *tcount* | An integer specifying the number of words or bytes to be written from the record. If this value is positive, it signifies *words;* if it is negative, it signifies bytes; if zero, 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 recsize value and the *multirecord* aoption is *not* specified in FOPEN, the FUPDATE request is refused and condition code CCL is returned; otherwise, the excess words or bytes are written to the next successive records. |

The condition codes possible are

| | |
|---|---|
| CCE | The request was granted. |
| CCG | An end-of-file condition was encountered during updating. |
| CCL | The request was not granted because of an error, such as the file not residing on a direct-access device, or *tcount* exceeding the size of the record when multirecord mode is not in effect. |

*EXAMPLE:*

*To update the last record referenced in the file identified by VNAME, the user wants to write a record from the array SAREA (in his stack) onto this record. He can do so by entering:*

```
FUPDATE (VNAME, SAREA, 50);
```

## SPACING ON SEQUENTIAL FILES

The user can space forward or backward on a sequential disc or tape file by using the FSPACE intrinsic call. (This results in re-setting the logical record pointer.) The FSPACE call format is

*PROCEDURE*  | *FSPACE (filenum,displacement) ;*

*VALUE filenum, displacement;*

*INTEGER filenum,displacement;*

*OPTION EXTERNAL;*

The FSPACE call parameters are

| | |
|---|---|
| *filenum* | A word identifier supplying the filenumber of the file on which spacing is to be done, through SPL/3000 conventions. |
| *displacement* | A signed integer indicating the number of logical records to be spaced over, relative to the current position of the file. A positive value signifies forward spacing; a negative value requests backward spacing. (For positive values, the sign is optional.) |

The condition codes possible are

| | |
|---|---|
| CCE | The request was granted. |
| CCG | A logical end-of-file indicator was encountered during spacing. |
| CCL | The request was not granted because an error occurred; the file resides on a device that prohibits spacing. |

*EXAMPLE:*

*To space forward 16 logical records on the file identified by F20, the user issues:*

```
FSPACE (F20, 16);
```

## RESETTING LOGICAL RECORD POINTER

The programmer can reset the logical record pointer for a sequential disc file, containing only fixed-length records, at any logical record in the file. When the next FREAD or FWRITE request is issued for the file, this record will be the one read or written. Positioning is done through the FPOINT intrinsic call:

*PROCEDURE*     | *FPOINT (filenum,recnum) ;* |

*VALUE filenum,recnum;*

*INTEGER filenum;*

*DOUBLE recnum;*

*OPTION EXTERNAL;*

The FPOINT call parameters are

*filenum*     A word identifier supplying the filenumber of the file on which spacing is to be done, through the conventions of SPL/3000.

*recnum*     A positive double integer representing the relative logical record at which the file is to be positioned. (The number of the first record is zero.)

The condition codes are

CCE     The request was granted.

CCG     The user requested positioning at a point beyond the physical end-of-file.

CCL     The request was not granted because an error occurred.


*EXAMPLE:*

*To position a file identified by FILE2 at the 40th logical record within it, the user enters:*


FPOINT (FILE2,39D)$

# OBTAINING FILE ACCESS INFORMATION

Once the user opens a file on any device, he can request access and status information about that file at any point in his program.

The parameters supplied in the FOPEN call, or corresponding information from any :FILE command or file label that overrides these parameters, are included in the information returned. The intrinsic call format is

PROCEDURE

| FGETINFO | (filenum, filename,foptions,aoptions,recsize,devtype, ldnum,hdaddr,filecode,recpt,eof,flimit,logcount, physcount,blksize,extsize,numextents,userlabels, creatorid,labaddr); |
|---|---|

VALUE filenum;

INTEGER filenum,recsize,devtype,filecode,blksize,numextents, userlabels;

BYTE ARRAY filename,creatorid;

LOGICAL foptions,aoptions,ldnum,hdaddr,extsize;

DOUBLE recptr,eof,flimit,logcount,physcount,labaddr;

OPTION VARIABLE, EXTERNAL;

The FGETINFO call parameters are

filenum
: A word identifier supplying the filenumber of the file about which information is requested, through SPL/3000 conventions.

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

    f.g.a

    f = The local file name.

    g = The group name (supplied or implicit).

    a = The account name (supplied or implicit).

    The byte-array must be 28-bytes long. When the actual designator is returned, unused bytes in the array are blank-filled to the right. A nameless file will return an empty string.

foptions
: A word to which is returned a bit string representing the file options currently effective. The format is the same as that of the *foptions* parameter of FOPEN.

| aoptions | A word to which is returned a bit string representing the access options which are currently effective. The format is the same as that of the aoptions parameter of FOPEN. |
|---|---|
| recsize | A word to which is returned the logical record size associated with the file. If the file was created as a binary type, this value is positive and expresses the size in *words*. If the file was created as an ASCII type, this value is negative and expresses the size in *bytes*. |
| devtype | A word to which is returned a positive integer representing the type of device being used for the file, |

<div style="margin-left: 4em;">

| | | |
|---|---|---|
| 0 | = | moving-head disc |
| 1 | = | fixed-head disc |
| 8 | = | card reader |
| 9 | = | paper tape reader |
| 16 | = | terminal |
| 24 | = | magnetic tape |
| 32 | = | line printer |
| 33 | = | card punch |
| 34 | = | paper tape punch |
| 35 | = | plotter |

</div>

| ldnum | A word to which is returned the logical device number associated with the device on which the file resides. |
|---|---|
| hdaddr | A word to which is returned the hardware address of the device, where: |

<div style="margin-left: 4em;">

| | | |
|---|---|---|
| Bits (0:8) | = | DRT number |
| Bits (8:8) | = | Unit number |

</div>

| filecode | A word to which is returned the value recorded with the file as its *file code* (for disc files). |
|---|---|
| recptr | A double word to which is returned a double integer representing the current logical record pointer setting. This is the displacement in logical records from Record No. 0 in the file. It identifies the record that would next be accessed by an FREAD or FWRITE call. |
| eof | A double word to which is returned a double positive integer representing the number of the last logical record currently in the file. (If the file does not reside on disc, this value will be zero.) |
| flimit | A double word to which is returned a double positive integer representing the number of the last logical record that could ever exist in the file, because of the physical limits of the file. (If the file does not reside on disc, this value will be zero.) |

| logcount | A double word to which is returned a double positive integer representing the total number of logical records passed to and from the user during the current access of the file. |
|---|---|
| physcount | A double word to which is returned a double positive integer representing the total number of physical input/output operations performed within this process against the file since the last FOPEN call. |
| blksize | A word to which is returned the block size associated with the file. If the file was created as a binary type, this value is positive and expresses the size in words. If the file was created as an ASCII type, this value is negative and shows the size in bytes. |
| extsize | A word to which is returned the disc extent size associated with the file (in sectors). |
| numextent | A word to which is returned the maximum number of disc extents allowable for the file. |
| userlabels | A word to which is returned the number of user header labels defined for the file when it was created. If the file is not a disc file, this number is 0. (When an old file is opened for overwrite-output, the value of *userlabels* is not reset; old user labels are not destroyed.) |
| creatorid | A byte-array to which is returned the eight-byte name of the user who created the file. If the file is not a disc file, blanks are returned. |
| labaddr | A double word to which is returned the sector address of the label of the file. If the file is not a disc file, this value is 0. The high-order eight bits show the logical device number; the next 24 bits show the absolute disc address. |

The condition codes are

| CCE | The request was granted. |
|---|---|
| CCG | (This condition code is not returned.) |
| CCL | The request was not granted because an error occurred. |

*EXAMPLE:*

*To request file access and status information about a file identified as F5, the user can enter the following call. This will return the file designator to the word FNDATA, the logical record size to the word FRDATA, and the type of device on which the file resides to FDDATA.*

**FGETINFO (FS,FNDATA,,,FRDATA,FDDATA);**

*After the intrinsic is executed, the values returned are as follows:*

| Byte-Array or Word | Content | Interpretation |
|---|---|---|
| *FNDATA* | *FILESAM.GP1* | *The file named FILESAM in the group GP1 in the user's log-on account.* |
| *FRDATA* | *100 (Decimal equivalent of content)* | *The file is a binary file containing fixed-length records each 100 words long.* |
| *FDDATA* | *24 (Decimal equivalent of content)* | *The file resides on magnetic tape.* |

## OBTAINING FILE-ERROR INFORMATION

When a file intrinsic returns a condition code indicating a physical input/output error, the programmer may require more details about the error in order to correct it. He can request this information with the FCHECK intrinsic call. (This intrinsic applies to files on any device.)

FCHECK is the only intrinsic that accepts zero as a legal *filenum* parameter value. When zero is specified, the returned error code reflects the status of the last call to FOPEN; zero is necessary to allow one to reference an erroneously-opened file. The format of FCHECK is

*PROCEDURE* | FCHECK *(filenum,* errorcode,tlog,blknum,numrecs) *;*

*VALUE filenum;*

*INTEGER filenum,* errorcode,tlog,numrecs;

*DOUBLE* blknum;

*OPTION VARIABLE, EXTERNAL;*

The FCHECK call parameters are

| | |
|---|---|
| *filenum* | A word identifier supplying the file number of the file for which error information is to be returned, through SPL/3000 conventions. |
| errorcode | A word to which is returned a 16-bit code, explained below, specifying the type of error that occurred. If the previous operation was successful, all bits are set to zero. |
| tlog | A word to which is returned the transmission-log value recorded when an erroneous data transfer occurs. This specifies the number of words not read or written (those left over) as the result of the input/output error. |
| blknum | A double word to which is returned the relative number of the block involved in the error. |
| numrec | A word to which is returned the number of logical records in the bad block. |

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

The following codes are returned by FCHECK:

| Code (Decimal) | Meaning |
|---|---|
| 20 | Invalid operation. |
| 21 | Data parity error. |
| 22 | Software time-out. |
| 23 | End of tape. |
| 24 | Unit not ready. |
| 25 | No write-ring on tape. |
| 26 | Transmission error. |
| 27 | Input/output time-out. |
| 28 | Timing error or data overrun. |
| 29 | Start input/output (SIO) failure. |
| 30 | Unit failure. |
| 31 | End-of-time (special character terminator). |
| 32 | Software abort. |
| 33 | Data lost. |
| 34 | Unit not on-line. |
| 35 | Data set not ready. |
| 36 | Invalid disc address. |
| 37 | Invalid memory address. |
| 38 | Tape parity error. |
| 39 | Recovered tape error. |
| 40 | Operation inconsistent with access-type. |
| 41 | Operation inconsistent with record type. |
| 42 | Operation inconsistent with device type. |
| 43 | The *tcount* parameter value exceeded the *recsize* parameter value in this intrinsic, but the *multirecord access* aoption was not specified in the currently-effective FOPEN intrinsic. |
| 44 | The FUPDATE intrinsic was called, but the file was positioned at record zero. (FUPDATE must reference the last record read, but no previous record was, in fact, read.) |
| 45 | Privileged-file violation. |
| 46 | Insufficient disc space. |
| 47 | Input/output error occurs on a file label. |
| 48 | Invalid operation due to multiple file access. |
| 49 | Unimplemented function. |
| 50 | The account referenced does not exist. |
| 51 | The group referenced does not exist. |
| 52 | The file referenced does not exist in the system file domain. |
| 53 | The file referenced does not exist in the job temporary file domain. |

| Code (Decimal) | Meaning |
|---|---|
| 54 | The file reference is invalid. |
| 55 | The device referenced is not available. |
| 56 | The device specification is invalid. |
| 57 | Virtual memory is not sufficient for the file specified. |
| 58 | The file was not passed. |
| 59 | Standard label violation. |
| 60 | Global RIN not available. |
| 61 | Group disc file space exceeded. |
| 62 | Account disc file space exceeded. |
| 63 | Non-sharable device capability not assigned. |
| 71 | Too many files for process. |
| 72 | Invalid file number. |
| 73 | Bounds-check violation. |
| 90 | The calling process requested exclusive access to a file to which another process has access. |
| 91 | The calling process requested access to a file to which another process has exclusive access. |
| 92 | Lockword violation. |
| 93 | Security violation. |
| 94 | Creator conflict in use of FRENAME intrinsic. |
| 100 | Duplicate file name in the system file directory. |
| 101 | Duplicate file name in the job temporary file directory. |
| 102 | Directory input/output error. |
| 103 | System directory overflow. |
| 104 | Job temporary directory overflow. |
| 110 | The intrinsic attempted to save a system file in the job temporary file directory. |

The condition codes possible are

CCE  The request was granted.

CCG  (This condition code is not returned.)

CCL  The request was not granted because of an error.

*EXAMPLE:*

*To return information about an error reported to his program for the file identified by F7, the user can issue the following call. This returns the error-code to the word TYPE, the number of words not transferred to the word RCOUNT, and the relative number of the bad block to the double-word ERBLOCK.*

**FCHECK (F7,TYPE,RCOUNT,ERBLOCK);**

*After the intrinsic is executed, the values returned are as follows:*

| Word/<br>Double Word | Content<br>(Decimal Values) | Meaning |
|---|---|---|
| *TYPE* | *73 (in low-order bits)*<br>*0 (in high-order bits)* | *A bounds-check violation occurred.* |
| *RCOUNT* | *53* | *Fifty-three records remain to be written, as a result of this error.* |
| *ERBLOCK* | *20* | *The 21st block was involved in the error.* |

## DIRECTING FILE CONTROL OPRERATIONS

The user can perform various control operations on a file (or the device on which it resides) by issuing the FCONTROL intrinsic call. These include: supplying a printer or terminal carriage-control directive, verifying input/output, reading the hardware status word pertaining to the device on which the file resides, setting a terminal's time-out interval, rewinding the file, writing and end-of-file indicator, and skipping forward or backward to a tape mark. The FCONTROL intrinsic applies to files on disc, tape, terminal, or line printer. (The FCONTROL intrinsic can also be used to perform various terminal functions, such as changing the terminal speed or enabling parity-checking. These applications of FCONTROL are described in Section VIII.) The FCONTROL intrinsic format is

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

*VALUE filenum,controlcode;*

*INTEGER filenum,controlcode;*

*LOGICAL param;*

*OPTION EXTERNAL;*

The FCONTROL call parameters are

*filenum*     A word identifier supplying the filenumber of the file for which the control operation is performed, through SPL/3000 conventions.

*controlcode*  An integer identifying the operation to be performed:

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

  1 = *Line Control.* A request to send the value specified in the *param* parameter (below) to the terminal or line printer driver as a carriage-control directive.

  2 = *Complete Input/Output.* This ensures that requested input/output has been physically completed.

  3 = *Read Hardware Status Word.* This operation returns to the calling process the hardware status word from the device on which the file resides.

  4 = *Set Time-Out Interval.* This code indicates that a time-out interval is to be applied to input from the terminal. If input is requested from the terminal but is not received in this interval, the input is again requested. The interval itself is specified (in seconds) in a word in the user's stack, indicated by *param*. If this interval is *zero*, any previously-established interval is cancelled, and no time-out occurs. *Controlcode 4* is ignored if the addressed file is not being read from the terminal.

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

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

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

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

  9 = *Rewind and Unload Tape File.* This repositions the tape file at its beginning and places the tape unit off-line.

param *If controlcode is 1, param* denotes a word containing the value to be transmitted to the terminal or line printer driver as a carriage-control or mode-control directive. The carriage-control directive is selected from Figure 6-3.

The mode control determines whether any carriage-control directive transmitted through the FWRITE intrinsic takes effect before printing (pre-space movement) or after printing (post-space movement). The mode control directive is selected from the octal codes %400 or %401 in Figure 6-3.

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

| Value | Meaning |
|-------|---------|
| 0 | = Post-spacing |
| 1 | = Pre-spacing |

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

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

The condition codes possible are

| | |
|------|-----|
| CCE | The request was granted. |
| CCG | (This condition code is not returned.) |
| CCL | The request was denied because an error occurred. |

*EXAMPLES:*

*The user can write a tape mark on the file identified as F47 by issuing this call;*

**FCONTROL (F47,6,X);**

*He can verify that the most recently-requested FWRITE operation against this file is actually completed by entering:*

**FCONTROL (F47,2,X);**

*The programmer can transmit a mode-control code to a line-printer file (F48) by issuing the following call.(The mode-control code is found in the word PRTCN. If this code is %401, the pre-space movement option is set.) If, after the FCONTROL intrinsic is executed, a value of 1 is returned to PRTCN, this indicates that the previous mode setting was the pre-spacing option.)*

**FCONTROL (F48,1,PRTCN);**

## DECLARING ACCESS-MODE OPTIONS

The user can activate or deactivate the following access-mode options by issuing the FSETMODE call: automatic error recovery, critical output verification, and terminal control by the user. The access mode established remains in effect until another FSETMODE is issued or until the file is closed. (This intrinsic applies to files on all devices.) The format is

*PROCEDURE*  | *FSETMODE (filenum,modeflags) ;*

*VALUE filenum,modeflags;*

*INTEGER filenum;*

*LOGICAL modeflags;*

*OPTION EXTERNAL;*

The FSETMODE call parameters are

*filenum*  A word identifier supplying the filenumber of the file to which this call applies, through SPL/3000 conventions.

*modeflags*  A 16-bit value that denotes the access-mode options in effect, as detailed below. If this parameter is omitted, all bits are set to zero by default.

The bits in the *modeflags* parameter are interpreted as follows:

**Bit Nos.**                                    **Access Mode Option**

(14:1)          Critical Output Verification

                1 = All output to the file is to be verified as physically complete before control returns from a write intrinsic to the user's program. For each successful logical write operation, a condition code (CCE) is returned *immediately* to the user's program.

                0 = Output is not verified.

(13:1)          Terminal Control by the User.

                1 = Inhibit normal terminal control by the system; thus, MPE/3000 will *not* issue an automatic carriage-return and line feed at the completion of each terminal input line.

                0 = MPE/3000 will automatically issue the automatic carriage-return and line feed for the terminal. This parameter is ignored if the output device is not a terminal.

(12:1)          Tape Error Recovery

                0 = Report recovered tape error with CCE.

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

The remaining thirteen bits (0:12) and (15:1) in this group are not used and are always set to *zero*.

The condition codes possible are

      CCE          The request was granted.

      CCG          (This condition code is not returned.)

      CCL          The request was not granted because an error occurred.

*EXAMPLE:*

*To establish an access mode where output operations are verified, terminal control is the user's responsibility, and a recovered tape error is reported with the CCE condition code, the programmer desires the following bit settings:*

| Bit No. | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bit Setting | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Octal Equivalent | 0 | | 0 | | 0 | | 0 | | 0 | | 6 | | | | | |

*For the file identified as IDENT, he can establish these settings by entering the octal value 0006, (preceded by the customary % required to denote an octal number) as the modeflags parameter in this FSETMODE call:*

FSETMODE (IDENT,%6);

## LOCKING AND UNLOCKING FILES

Sometimes the user may want to manage the sharing of a file between two or more processes so that no two of these processes can access the file simultaneously. He can do this for files on any devices through the dynamic locking and unlocking of files with the FLOCK and FUNLOCK intrinsic calls. Because such locking and unlocking entails the use of resource identification numbers (RIN's), however, discussion of the FLOCK and FUNLOCK calls is deferred until the discussion of RIN's in Section IX.

## RENAMING A FILE

The user can rename (change the formal designator of) an open disc file, with the FRENAME intrinsic call. This effectively changes the formal designator (including *lockword*) of the file. This file must be either:

1.  A new file, or

2.  An existing file, opened for fully-exclusive access, to which the user has write-access (specified by the security provisions of the file).

The format of FRENAME is

*PROCEDURE*  | *FRENAME (filenum,newfilereference) ;*

*VALUE filenum;*
*INTEGER filenum;*
*BYTE ARRAY newfilereference;*
*OPTION EXTERNAL;*

The parameters are

| | |
|---|---|
| *filenum* | A word identifier supplying the filenumber of the file to be renamed, through SPL/3000 conventions. |
| *newfilereference* | A byte-array containing an ASCII string representing the new name for the file. This parameter may not specify the name of an account other than that to which the file is currently assigned. Additionally, the caller must have *save* access to the group specified or implied by the string. The string must begin with a letter, contain alphanumeric characters, and terminate with a non-alphanumeric character other than a slash or period. |

The condition codes possible are

| | |
|---|---|
| CCE | The request was granted. |
| CCG | (This condition code is not returned.) |
| CCL | The request was not granted because an error occurred. |

*EXAMPLE:*

*To rename a file identified as FNR, with the new designator contained in the byte-array NEWFL, the user can enter:*

```
FRENAME (FNR,NEWFL);
```

## DETERMINING INTERACTIVE AND DUPLICATIVE FILE PAIRS

An input file and a list file are said to be *interactive* if a real-time dialogue can be established between a program and a person using the list file as a channel for programmatic requests, with appropriate responses from a person using the input file. For example, an input file and a list file opened to the same teletype terminal (for a session) would constitute an interactive pair. An input file and a list file are said to be *duplicative* when input from the former is automatically duplicated on the latter. The user can determine whether a pair of files is interactive or duplicative through the FRELATE intrinsic call. (The interactive/duplicative attributes of a file pair do not change between the time it is opened and the time it is closed.) (This intrinsic applies to files on all devices.)

The format of FRELATE is

*LOGICAL PROCEDURE*

| *FRELATE (infilenum,listfilenum) ;* |
|---|

*VALUE infilenum,listfilenum;*

*INTEGER infilenum,listfilenum;*

*OPTION EXTERNAL;*

This intrinsic returns to the calling process a word showing whether the files are interactive and/or duplicative.

The parameters are

infilenum                    A word identifier supplying the filenumber of the input file, through SPL/3000 conventions.

listfilenum                  A word identifier supplying the filenumber of the list file, through SPL/3000 conventions.

The word returned to the user's process contains two significant bits:

Bit (15:1)    =    1, if infilenum and listfilenum form an interactive pair

              =    0, if they do not.

Bit (0:1)     =    1, if infilenum and listfilenum form a duplicative pair.

              =    0, if they do not.

Bits 1-14 are not used, and are always set to zero.

The condition codes possible are

CCE       The request was granted.

CCG       (This condition code is not returned.)

CCL       The request was not granted because an error occurred.


*EXAMPLE:*

*To determine whether the files identified as FABLE and FBAKER are interactive or duplicative, the user can issue this call:*

**INDUP := FRELATE (FABLE,FBAKER);**

*After the intrinsic is executed, if Bit 15 of the word INDUP is set to 0 but Bit 0 of that word is set to 1, the user knows that FABLE and FBAKER are not interactive but are duplicative.*

## FILE INTRINSIC FILE-TYPE SUMMARY

The file intrinsics described in this section and the types of files/devices to which they apply are summarized below:

| All Devices | Disc | Disc and Tape | Printer, Terminal, Tape, and Disc |
|---|---|---|---|
| FOPEN | FUPDATE | FSPACE | FCONTROL |
| FCLOSE | FPOINT | | |
| FREAD | FREADDIR | | |
| FWRITE | FWRITEDIR | | |
| FCHECK | FREADSEEK | | |
| FGETINFO | FRENAME | | |
| FSETMODE | FREADLABEL | | |
| FLOCK | FWRITELABEL | | |
| FUNLOCK | | | |
| FRELATE | | | |

# SECTION VII
# Managing Program Libraries

The user can perform various management functions relating to program libraries. In general, he can

- Modify user subprogram libraries (USL's) by adding, deleting, activating, and de-activating relocatable binary modules (RBM's) stored in them

- Manage libraries used to resolve program references to external procedures

- Dynamically attach and detach a library procedure to a running program

The first two of these functions are performed through the MPE/3000 Segmenter.

## MANAGING USER SUBPROGRAM LIBRARIES (USL'S)

As noted previously, compiled user programs are stored in files called USL's that reside on disc. In any particular USL, each compiled program unit exists as a Relocatable Binary Module (RBM). (In FORTRAN/3000, a program unit is a main program, subroutine, function, or block data unit; in SPL/3000, a program unit is an outer block or procedure. In COBOL/3000, a program unit is a main program or section.)

To prepare a program (and the program units it references) for execution, the MPE/3000 Segmenter selects the appropriate RBM's from the USL and binds them into linked segments written on a program file.

Each RBM is identified by the symbolic name (label) of the primary entry point of the program unit stored in the RBM; the RBM may be associated with a segment name, determined during compilation, that identifies the segment to which it will belong. An activity bit is associated with each entry point (the primary entry point and any secondary entry points). This activity bit determines whether the program unit currently can be entered at the corresponding entry point. When a compiler writes an RBM on a USL, all entry points in the RBM are set to the *active* (entry allowed) state. Through the MPE/3000 Segmenter, they can then be switched from that state to the *inactive* (entry disallowed) state, and back again, by the user.

When a program file is prepared from the USL, all RBM's having at least one active entry point are extracted from the USL for segmentation on the program file. Those having identical segment names are placed in the same segment. To permit the creation of a program file that can be executed properly, only one outer-block or main-program RBM can have active entry points, along with the RBM's for the subprograms or procedures on the USL that it references.

Entry points having the same name are permitted in a USL. Each of these is uniquely identified by its name used in conjunction with an index integer; this integer denotes the particular definition of the entry point in chronological terms, beginning with one to indicate the most recent definition. Since the name of an RBM is equivalent to that of its primary entry-point, the name/index identification method allows RBM's containing different versions of the same program unit to be referenced by the same name used with different index integers. Thus, to identify the most recent version of a program unit in an RBM named START, the user would specify START (1); to indicate the next most recent version, he would specify START (2). A special convention (index = 0) is used to indicate the most recent *active* definition; for example, START (0).

Sometimes, the user may want to alter the structure of a program to incorporate new features or modify existing ones. For example, the user may want to include a new version of a main program or subroutine in place of a previous version. He can do this by adding and deleting RBM's in the pertinent USL, or by activating or deactivating appropriate RBM entry points.

Additionally, the user may want to change the segmentation of a program (and its associated subprograms) to improve its efficiency. In some operating systems, such a modification would require recompilation. Under MPE/3000, however, the user can simply re-arrange the RBM's and then prepare the USL into a new program file.

Before modifying a USL, the user first accesses the MPE/3000 Segmenter by entering:

    *:SEGMENTER  [listfile]*

*listfile*    AN ASCII file (formal designator SEGLIST) from the output set, to which is written any listable output generated by Segmenter subsystem commands. (The designator SEGLIST should *not* be used as the *actual* file designator.) If *listfile* is omitted, the standard job/session list device ($STDLIST) is assigned. (Optional parameter.)

He then enters any of the Segmenter commands defined below to modify the USL. In a session, a dash is issued as the prompt character for each Segmenter command. No prompt character, however, is allowed in a batch job. (The rules covering delimiters, blanks, positional parameters, and continuation characters in Segmenter commands are the same as those rules for all other MPE/3000 commands.) When the user has completed USL modification, he enters the following command to return control from the Segmenter back to the MPE/3000 command interpreter:

    *-EXIT*

Segmenter commands, rather than the conventional MPE/3000 commands, are required to create and change USL's because these files have a special format. However, once a USL has been created, it can be managed by other MPE/3000 commands and intrinsics — for example, it can be purged (:PURGE command), renamed (:RENAME command), or read (FREAD intrinsic).

## Creating New USL's

To create a new (job temporary) USL onto which RBM's containing user program units can be compiled or copied, the programmer enters a command of this format. The USL then exists as a job temporary file.

*–BUILDUSL filereference,records,extents*

*filereference*   The name (file designator) assigned to the new USL file. (This is a fully-qualified file reference that may include file name, group name, and account-name, plus a lockword.) (Required parameter.)

*records*   The length of this file in terms of 128-word logical records. (Required parameter.) NOTE: *All* logical records in files referenced by the Segmenter command parameters described in this section of the manual are 128-words long.

*extents*   The number of disc extents to be allocated to the file. (Required parameter.)

*EXAMPLE:*

*To create a new USL named NEWUSL, the user enters the following command. The USL will contain 300 logical records and will be allocated one extent.*

**-BUILDUSL   NEWUSL,300,1**

## Designating USL's For Management By The User

To enter commands that modify an existing USL in any way, the user must first identify this USL by the –USL command:

*–USL filereference*

*filereference*   The name (and optional group and account names) of the USL file to be manipulated. (Required parameter.)

Then, all subsequent USL-modification commands will refer implicitly to this USL until a new –USL command is issued.

If, however, the user has entered a –BUILDUSL command, this acts as an implicit –USL command for the file referenced. That is, all USL-modification commands between a –BUILDUSL command and the next –USL command or –BUILDUSL command refer to the file specified in the first –BUILDUSL command.

*EXAMPLE:*

*The following sample terminal listing shows how different USL files are designated for management by Segmenter commands:*

| Command | Operation |
|---------|-----------|
| • <br> • <br> • | |
| :SEGMENTER LSTFL | *Accesses the Segmenter.* |
| -USL FILE1 | *Designates the USL file FILE1 (in the user's log-on group and account) for management.* |
| -USE UNIT, PROG(2) <br> -CEASE UNIT, PROG(1) | *These commands modify FILE1.* |
| -BUILDUSL FILE2, 200, 1 | *Creates a new file, FILE2 (in the user's log-on group and account), and designates this file for management.* |
| -AUXUSL FILE1 <br> -COPY UNIT, ORBM | *Copies an RBM from FILE1 to FILE2.* |
| -USL FILE3 | *Designates FILE3 (in the user's log-on group and account) for management.* |
| -LSTUSL <br> • <br> • <br> • | *Lists RBM's residing in FILE3.* |
| -EXIT | *Exits from Segmenter.* |

7-4

## Activating Entry Points

To activate one or more program unit (RBM) entry points in the USL currently designated for management, the user enters the USE command:

_USE  [pointspec,]  name  [(index)]

| | |
|---|---|
| pointspec | Is ENTRY, to activate the entry point indicated by *name (index)*; or |
| | UNIT, to activate all entry points in the RBM indicated by (the primary entry point) *name (index)*; or |
| | SEGMENT, to activate all entry points in all RBM's associated with the segment *name.* |
| | The default value is ENTRY. (Optional parameter.) |
| name | The name of the entry point, RBM (referred to by its primary entry-point name), or segment referenced. (Required parameter.) |
| index | The index integer further specifying the entry point *name.* The index may be used when the USL contains more than one entry point of this name. If *index* is omitted, a default value of 0 is assigned. If *pointspec* is SEGMENT, the *index* parameter does not apply. (Optional parameter.) |

*EXAMPLE:*

*Suppose the user is managing a USL file named FILE2. On this file, he wants to activate all entry points in the most recent active version of the program unit whose primary entry point (RBM name) is RB20.*

```
-USL  FILE2
       •
       •
       •
-USE  UNIT,  RB20
```

7-5

## Deactivating Entry Points

To deactivate one or more entry points in the USL currently designated for management, the user enters:

_CEASE [pointspec,] name [(index)]

pointspec          Is ENTRY, to deactivate the entry point indicated by name (index); or

UNIT, to deactivate all entry points in the RBM indicated by name (index); or

SEGMENT, to deactivate all entry points in all RBM's associated with segment name.

The default parameter is ENTRY. (Optional parameter.)

name          The name of the entry point, RBM, or segment referenced. (Required parameter.)

index          The index integer further specifying the entry point name. The index may be used when the USL contains more than one entry point or RBM of this name. If index is omitted, a default value of 0 is assigned. If pointspec is SEGMENT, the index parameter does not apply. (Optional parameter.)

*EXAMPLE:*

*To deactivate the entrypoint BEGIN2 in the USL named FILE3, the user enters:*

**-USL FILE3**
.
.
.
**-CEASE ENTRY, BEGIN2**
.
.
.

**Deleting RBM's**

To delete one or more RBM's from a USL, the user can enter the -PURGERBM command:

_-PURGERBM_ *[rbmspec,] name [(index)]*

| | |
|---|---|
| *rbmspec* | Is UNIT, to delete the RBM identified by *name (index)*; or |
| | SEGMENT, to delete all RBM's associated with the segment *name.* |
| | The default parameter is UNIT. (Optional parameter.) |
| *name* | If *rbmspec* is UNIT, this is the name of the RBM to be deleted. If *rbmspec* is SEGMENT, this is the name of the segment in which all RBM's are to be deleted. (Required parameter.) |
| *(index)* | If *rbmspec* is UNIT, this is the index integer further specifying the RBM name; the index may be used when more than one RBM of this name exists in the USL. If *index* is omitted, a default value of *0* is assigned. If *rbmspec* is SEGMENT, this parameter does not apply. (Optional parameter.) |

*EXAMPLE:*

*To delete the RBM named XPOINT (on the USL named XFILE), the user enters:*

        **-USL XFILE**
         •
         •
         •
    ▶ **-PURGERBM   UNIT, XPOINT**

## Assigning New Segment Names to RBM's

The user can change the segment name associated with an RBM, thus assigning the RBM to a different code segment the next time it is prepared onto a program file. The segment name is changed by entering:

_NEWSEG  newsegname ,rbmname[(index)]

| | |
|---|---|
| *newsegname* | The new segment name to be associated with the RBM. (Required parameter.) |
| *rbmname* | The name of the RBM whose segment name is to be changed. (Required parameter.) |
| *(index)* | The index integer further specifying the RBM name. It may be used when more than one RBM of the same name exists in the USL. If *index* is omitted, a default value of 0 is assigned. (Optional parameter.) |

*EXAMPLE:*

*To associate the new segment name NEWNAME with an RBM named RB3 in the USL FILE7, the user enters:*

```
-USL  FILE7
    •
    •
    •
-NEWSEG  NEWNAME,  RB3
    •
    •
    •
```

## Transferring RBM's

The user can transfer one or more RBM's from another USL to the currently-managed USL. First, he must designate the USL source (from which the RBM's are transferred), by entering the –AUXUSL command:

> _–AUXUSL_ _filereference_

_filereference_   The name (and optional group and account names) of the USL file from which the RBM's are to be transferred. (Required parameter.)

Next, the user transfers (copies) the RBM's to the destination USL by entering the –COPY command. (All subsequent –COPY commands will refer implicitly to the source USL file designated by the current –AUXUSL command, until a new –AUXUSL command is encountered.)

> _–COPY [rbmspec,]_ _name_ _[(index)]_

_rbmspec_   Is UNIT, to transfer the RBM identified by _name (index)_ from the source USL; or

SEGMENT to transfer _all_ RBM's associated with the segment _name_ from the source USL.

The default parameter is UNIT. (Optional parameter.)

_name_   If _rbmspec_ is UNIT, _name (index)_ identifies the RBM to be transferred. If _rbmspec_ is SEGMENT, _name_ identifies the segment from which all RBM's are transferred. (Required parameter.)

_index_   The index integer further specifying the RBM name; the index may be used when more than one RBM of this name exists in the USL. If _index_ is omitted, a value of 0 is assigned by default. This parameter is ignored if _rbmspec_ is SEGMENT. (Optional parameter.)

When an RBM is transferred, the segment name associated with it (if any) remains the same.

*EXAMPLE:*

*To move all RBM's associated with the segment name SEGNAME from the USL named BASEFL to the USL named DESTFL, the user enters:*

```
-USL  DESTFL
  •
  •
  •
-AUXUSL  BASEFL
  •
  •
  •
-COPY  SEGMENT,  SEGNAME
```

## Listing RBM's

The user can list the names of the RBM's in the currently-managed USL, their segment names, entry points, and other related information, by entering:

*-LISTUSL*

The output is written on the file designated in the *listfile* parameter of the :SEGMENTER command. It shows all program unit entry points. Those entry points related to procedure or block data program units are listed under the segment in which they are contained. Interrupt procedures and block data program units appear in separate lists.

An example of the output produced by the -LISTUSL command is shown in Figure 7-1. On this listing, significant entries are indicated with arrows, keyed to the discussion below. All numbers appearing on this listing are octal values.

| Item No. | Meaning |
|---|---|
| 1 | The name of the USL file (filename.groupname.accountname). |
| 2 | The segment name. |
| 3 | The RBM (entry-point or program unit) name. |
| 4 | The number of words (octal) in the RBM code module. |

5          The type of entry-point named, where:

| | | |
|---|---|---|
| P | = | Procedure primary entry point. |
| SP | = | Secondary entry point for procedure. |
| O | = | Outer block primary entry point. |
| SO | = | Secondary entry point for outer block. |
| BD | = | Block data entry point. |
| In | = | Interrupt procedure entry point (n is the interrupt procedure type number, ranging from 0 through 2). |

6          The active/inactive indicator, where:

| | | |
|---|---|---|
| A | = | Active |
| I | = | Inactive |

7          The callability indicator, where:

| | | |
|---|---|---|
| C | = | The entry point is directly callable by the user. |
| U | = | The entry point is not directly callable by the user. |

8          The privileged/not-privileged indicator:

| | | |
|---|---|---|
| P | = | Privileged. |
| N | = | Not privileged. |

9          The internal flag status:

| | | |
|---|---|---|
| H | = | Internal flag *on* |
| R | = | Internal flag *off*. |

(The internal flag and its purpose are explained later in this section.)

10          The total size of the USL file, in words (octal).

| Item No. | Meaning |
| --- | --- |
| 11 | The number of words (in octal) used in the directory portion of the USL file. |
| 12 | The number of words (in octal) in the directory part of the USL file, filled with meaningless information, as a result of various RBM manipulations. |
| 13 | The number of unused words (in octal) in the directory portion of the USL file. |
| 14 | The number of words (in octal) used in the RBM-information portion of the USL file. |
| 15 | The number of words (in octal) in the RBM-information portion of the USL file, filled with meaningless information, as a result of various RBM manipulations. |
| 16 | The number of unused words (in octal) in the RBM-information portion of the USL file |

## Preparing Program Files

Once the user has modified a USL, he can prepare the active RBM's residing on it into a program file. He can do this by entering the :PREP command, described in Section IV. MPE/3000 also permits him to enter a similar command (-PREPARE) through the Segmenter, in which case the *dash* (rather than the colon) is used as the prompt character (in sessions) and the parameters defined below apply. The currently-managed USL is used as the RBM source.

    *-PREPARE progfile*

        *[;ZERODB]*
        *[;PMAP]*
        *[;MAXDATA=segsize]*
        *[;STACK=stacksize]*
        *[;DL=dlsize]*
        *[;CAP=caplist]*
        *[;RL=filename]*

| | |
| --- | --- |
| *progfile* | The name of the program file onto which the prepared program segments are to be written. This can be any binary file from the output set. (Required parameter.) |

```
USL FILE SCR4.PUB.SYS

SEG40 ←②          ④   ⑤   ⑥   ⑦   ⑧
    LISTRL ←③    351   P   A   C   N   R ←⑨
    CLEANUPRLEXTNBU   13   P   A   C   N   R
    GETNEXTRLEXTN     34   F   A   C   N   R
    SETUPRLEXTNBUF    36   P   A   C   N   R
    ADDEDPROC         27   F   A   C   N   R
    DELETEDPROC            SF  A   C       R
    FIXUPRL          502   P   A   C   N   R
    REMOVERL          72   P   A   C   N   R
    INSERTRL         434   P   A   C   N   R
    CLEANUPRLBUF      11   P   A   C   N   P
    DELETERLENTRY     46   P   A   C   N   R
    FINDRLDIRSPACE    72   P   A   C   N   R
    GETNEXTRLENTRY    44   P   A   C   N   P
    SETUPRLBUF        27   P   A   C   N   R
    RLENTRYPARMS      23   F   A   C   N   P
    GETNEXTRLRECD     20   P   A   C   N   R
    SEARCHRL          76   F   A   C   N   P
    SAVERLMAP         11   P   A   C   N   R
    GETRLMAP          14   P   A   C   N   R
    RETURNRLSPACE     27   F   A   C   N   R
    FINDRLSPACE      126   F   A   C   N   P
SEG33
    REMOVESL         106   P   A   C   N   P
    BINDSEGS         415   P   A   C   N   P
    FIXUPSL          125   P   A   C   N   R
    RETURNSLSPACE     14   F   A   C   N   P
SEG32
    INSERTNEWSEG     202   P   A   C   N   R
    MAKEENTRIES      163   F   A   C   N   P
    INSERTSEG        203   F   A   C   N   R
    FINDSLSPACE      220   F   A   C   N   R
    FINDSLRECORD      35   P   A   C   N   R
SEG31
    CLEANUPRLTBUF     11   F   A   C   N   R
    GETREFTABENTRY    33   F   A   C   N   P
```

```
SEG1
    CB·              606   OB  A   C   N
    CORRECTCLASS      32   P   A   C   N   P


    SETACTIVITY       43   P   A   C   N   R

FILE SIZE    400000 ←⑩
DIR. USED     10001 ←⑪        INFO USED      40737 ←⑭
DIR. GARB.        0 ←⑫        INFO GARB.         0 ←⑮
DIR. AVAIL.   67577 ←           INFO AVAIL.   237041 ←⑯
                   ⑬
```

Figure 7-1. – LISTUSL Command Output

7-13

| ZERODB | An indication that the initially-defined DL-DB area, and uninitialized portions of the DB-Q (initial) area will be initialized to zero. If this parameter is omitted, these areas are not affected. (Optional parameter.) |

*ZERODB*  An indication that the initially-defined DL-DB area, and uninitialized portions of the DB-Q (initial) area will be initialized to zero. If this parameter is omitted, these areas are not affected. (Optional parameter.)

*PMAP*  An indication that a listing describing the prepared program will be produced on the device specified by the :SEGMENTER command parameter *listfile*. (An example of this listing appears under the discussion of the :PREP command in Section IV.) If this parameter is omitted, no listing is produced. (Optional parameter.)

*segsize*  Maximum stack area (Z-DL) size permitted, in *words*. This parameter is included if the user expects to change the size of the DL-DB or Z-DB areas during process execution. If omitted, MPE/3000 assumes that he will not change these areas. (Optional parameter.)

*stacksize*  The size of the user's initial local data area (Z-Q (initial)) in the stack, in *words*. This overrides the *stacksize* estimated by the Segmenter, which applies if the stacksize parameter is omitted. (The default is a function of estimated stack requirements for each program unit in the program. Since it is difficult for the system to predict the behavior of the stack at run time, the user may want to override the default by supplying his own estimate with *stacksize*.) (Optional parameter.)

*dlsize*  The DL-DB area to be initially assigned to the stack. This area is mainly of interest only in programmatic applications, and is discussed in detail in Section II. If the dlsize parameter is omitted, a value of zero is used. (Optional parameter.)

*caplist*  The *capability-class attributes* associated with the user's program; specified as two-character mnemonics. If more than one mnemonic is specified, each must be separated from its neighbor by a comma.

The mnemonics are

    IA  = Interactive access

    BA  = Local batch access

    PH  = Process handling

    DS  = Data segment management

    MR  = Multiple resource management

    PM  = Privileged-mode operation

The user who issues the -PREPARE command can only specify capabilities that he himself possesses (through assignment by the Account Manager). If the user does not specify any capabilities, only IA and BA (if the user possesses them) will be assigned to this program. (Optional parameter.)

*filename*       The name (and optional account and group names) of a relocatable procedure library (RL) to be searched to satisfy external references during preparation. This can be any permanent file of type RL. It need not belong to the log-on group, nor does it have a reserved, local name. This file yields a single segment that is incorporated into the segments of the program file prepared. If *filename* is omitted, no library is searched. (Optional parameter.)

For the *stacksize, dlsize*, and *maxdata* parameters, a value of -1 denotes the default (equivalent to omitting the parameter).

## USING EXTERNAL PROCEDURE LIBRARIES

As part of its file domain, each group within an account may optionally include files comprising procedure *libraries.* These libraries contain external procedures frequently called by programmers accessing this group. They permit several users to concurrently access common routines with efficiency.

Under MPE/3000, there are two types of libraries: Relocatable Libraries and Segmented Libraries. Each group optionally may have one or more Relocatable Libraries and one Segmented Library.

### Relocatable Libraries

Relocatable Libraries (RL's), are specially-formatted files that are searched at *program preparation* time to satisfy references to external procedures required by the user's program. Within such libraries, these procedures exist in RBM form (as they would on a USL). When a program is prepared, these procedures are placed in a single segment and linked to the user's program in the resulting program file. Since a segment containing procedures from an RL is specifically constructed for a given program, any such segment cannot be shared concurrently by different programs. Instead, individual copies of the required procedures must be made and segmented for each program requesting them.

An RL is always a permanent file, but it need not belong to the user's log-on group nor have a reserved local file name.

The RL to be searched during preparation of the user's program is specified by the *filename* parameter of the :PREP, :PREPRUN, or -PREPARE command. For this parameter, the *filereference* format introduced in Section V is used.

Procedures in an RL may not contain TRACE variables.

*EXAMPLE:*

*To specify that an RL named RL in the group GROUP2 of the user's log-on account be searched during preparation of a program from the USL ALPHA1 to the program file ALPHA2, the user enters the following command:*

**:PREP ALPHA1, ALPHA2: RL=RL.GROUP2**

## Segmented Libraries

Segmented Libraries (SL's) are specially-formatted files that are searched at program-allocation time to satisfy references to external procedures. These libraries, like program files, contain procedures in segmented (prepared) form. An individual procedure may be the only procedure in its segment, or may exist in a segment containing many other procedures. When a procedure is referenced, the segment containing it is loaded along with other segments referenced by that procedure. Since the segmentation is not altered when different programs reference procedures in an SL, these procedures may be shared concurrently by many programs.

From one to three SL's may be searched during allocation of the user's program. These are

1. The Group Library (GSL), which is actually the library of the group under which the program file is stored. This library is readable by any user who can access this group.

2. The account's Public Library (PSL), which is the library of the public group of the account under which the program file is stored. This library is readable by any user who can access this account.

3. The System Library (SSL), which is the library of the public group of the system account. This library can be accessed by all users of the system.

The order in which the libraries are searched is specified by the LIB parameter in the :RUN command that requests program allocation/execution, as follows:

| LIB Parameter | Specification |
|---|---|
| LIB = G | The libraries are searched in this order:<br>• Group Library<br>• Public Library<br>• System Library |
| LIB = P | The libraries are searched in this order:<br>• Public Library<br>• System Library |
| LIB = S | Only the System Library is searched. |

When no LIB parameter is specified, the default is LIB = S.

*EXAMPLE:*

*To specify that the public library and then the system library be searched during allocation, the user enters the following command when executing the program ALPHA2.*

**:RUN ALPHA2; LIB=P**

The name of the System Account is always SYS. Within an account, the name of any public group is always PUB; the name of the segmented library is always SL. Thus, when referencing a library file in an MPE/3000 command or intrinsic, the following rules apply.

1. *If the user's program file is a permanent file,* with the name X.Y.Z (in the filereference format), then:

    GSL is referenced as SL.Y.Z

    PSL is referenced as SL.PUB.Z

    SSL is referenced as SL.PUB.SYS

    If Y.Z = PUB.SYS, then only LIB = S is a legal entry in the :PREPRUN or :RUN command.

    If Y.Z = PUB.Z, then LIB = P or LIB = S are legal entries in the :PREPRUN or :RUN command.

    During allocation, the proper libraries are determined by the position of the program file in the file system directory.

2. *If the user's program file is not a permanent file,* but is a job/session temporary or passed file with the name X.Y.Z, where Y is the log-on group and Z is the log-on account, or is nameless (such as the file $OLDPASS) then:

    GSL = SL.Y.Z

    PSL = SL.PUB.Z

    SSL = SL.PUB.SYS

    In the above cases, LIB = G, LIB = P, or LIB = S are all legal entries in the :PREPRUN or :RUN command.

    During allocation, the proper libraries are determined by the log-on group and account.

As an example of how SL's are used, procedures called frequently by many users throughout the system are stored in the SSL. Those used frequently throughout an account can be stored in an PSL. Those used by only a few users in a single group, less frequently, might be stored in a GSL.

Because the global area of the user's stack is already formed at the time a user's program references an external procedure in an SL, the library procedures cannot contain global variables. From the user's standpoint, this means that SPL/3000 procedures in a library cannot contain TRACE, EXTERNAL or OWN variables. Also, FORTRAN procedures in a library cannot contain: DATA, COMMON, LABELED COMMON, or TRACE variables; FORMAT statements; or references to LOGICAL UNITS.


## CREATING AND MAINTAINING RELOCATABLE LIBRARIES (RL'S)

To enter commands to create and maintain relocatable libraries, the user first accesses the Segmenter by entering the :SEGMENTER command. He may then enter any of the following commands, preceded (in sessions) by a dash as the prompt character.

As with USL's, Segmenter commands are required to create and change RL's because these files are written in a special format. But once an RL is created, it can be referenced by other MPE/3000 commands and intrinsics; for example, it is purged with the :PURGE command, renamed with the :RENAME command, or read with the FREAD intrinsic.


### Creating a Relocatable Procedure Library (RL) File

To create a permanent, formatted RL file, the user enters the following command.

_-BUILDRL filereference, records, extents_

| | |
|---|---|
| *filereference* | The filename of the new RL, in the *filereference* format, (optionally including group and account identifiers). (Required Parameter.) |
| *records* | The total maximum file capacity, specified in terms of 128-word, binary logical records. (Required parameter.) |
| *extents* | The total number of disc extents that can be dynamically-allocated to the file as logical records are written to it. The size of each extent is determined by the *records* parameter value divided by the *extents* parameter value. The *extents* value must be an entry from 1 to 16. (Required parameter.) |

## Designating RL's for Management by the User

To enter commands that modify an existing RL in any way, the user must first identify the RL by the  -RL command:

_-RL_  _filereference_

_filereference_   The name (and optional group and account names) of the RL to be modified, in the _filereference_ format. (Required parameter.)

Then, all subsequent RL-modification commands will refer implicitly to this RL until a new -RL command is issued.  If, however, the user has entered a -BUILDRL command, this acts as an implicit -RL command for the file referenced.  That is, all RL-modification commands between a -BUILDRL command and the next -RL command or -BUILDRL command refer to the file specified in the first -BUILDRL command.

## Adding a Procedure to an RL

To add a procedure to the currently-managed RL, the user copies the RBM containing that procedure from the currently-managed USL to the RL. The user must have write-access to the designated RL file. The procedure is added with the following command:

_-ADDRL_  _name[(index)]_

_name_         The name (primary entry point) of the RBM containing the procedure to be added to the RL.  (Required parameter.)

_index_        The index integer further identifying the RBM. The index may be used when the currently-managed USL contains more than one active RBM of this name. If _index_ is omitted, a value of 0 is assigned by default. (Optional parameter.)

## Deleting an Entry-Point or Procedure from an RL

To delete an entry point of a procedure, or an entire procedure, from an RL, the user enters:

_-PURGERL_  _[rlspec,] name_

_rlspec_       Is UNIT, to delete the procedure identified by _name_; or

               ENTRY, to delete the entry point identified by _name_.

               The default parameter is ENTRY.  (Optional parameter.)

_name_         If  _rlspec_ is UNIT, this is the name of the procedure to be deleted.  If _rlspec_ is ENTRY, this is the name of the entry point to be deleted. (Required parameter.)

When the last entry point in a procedure is deleted, the entire body of code representing that procedure is deleted.

Additionally, the entire RL can be deleted by the MPE/3000 command, :PURGE.


## Listing Procedures in an RL

To list all procedure entry-points and external references in the currently-managed RL, the following command is used:

   _-LISTRL

An example of the output produced by the -LISTRL command is shown in Figure 7-2. On this listing, significant entries are indicated with arrows, keyed to the discussion below. (All numbers appearing on this listing are octal values.)

| Item No. | Meaning |
|---|---|
| 1 | The name of the RL file (filename.groupname.accountname). |
| 2 | The procedure entry-point name. |
| 3 | The parameter checking level for the entry point. |
| 4 | The entry-point address (relative displacement within the code module). |
| 5 | The RL file address of the procedure information block. |
| 6 | The number of entry points in the procedure. (If this field is blank, the entry-point name field (Item 2) names a secondary entry point; if this field contains a number, the entry-point name field contains a primary entry-point name.) |
| 7 | The length of the code module, in words. (If this field is blank, the entry-point name field names a secondary entry-point; if this field contains a number, the entry-point name field contains a primary entry-point name.) |
| 8 | The length of the procedure information block, in words. (If this field is blank, the entry-point name field names a secondary entry point; if this field contains a number, the entry-point name field contains a primary entry-point name.) |
| 9 | The name of the external reference. |
| 10 | The parameter checking level for the external reference. |
| 11 | The entry-point address of the external reference. |
| 12 | The RL file address of the external procedure information block. (If a number appears in this field *and* in the entry-point address field (Item 11), the external reference is satisfied within the RL. However, if these fields are blank, the external reference is *not* satisfied within the RL.) |
| 13 | The number of words presently used in the RL file. |
| 14 | The number of words presently available in the RL file. |

RL FILE SCR1.MPE.SYS ①

\* ENTRY POINTS \* ①

| | ② | ③ | ④ | ⑤ | ⑥ | ⑦ | ⑧ |
|---|---|---|---|---|---|---|---|
| FERROR | 0 | 0 | 3600 | 1 | 105 | 174 | |
| FREADMR' | 0 | 20 | 4600 | | | | |
| FEOF | 0 | 0 | 5400 | 1 | 13 | 41 | |
| PRINTWARNING | 0 | 25 | 400 | | | | |
| CLEARLINE | 0 | 0 | 2700 | 1 | 10 | 22 | |
| FREADDIR' | 0 | 0 | 700 | 1 | 12 | 53 | |
| MESSAGE | 0 | 1104 | 1400 | 1 | 1210 | 1222 | |
| FREADMR'' | 0 | 0 | 4600 | 2 | 32 | 105 | |
| PRINTERROR | 0 | 0 | 400 | 2 | 122 | 235 | |
| DNTOA | 0 | 0 | 2640 | 1 | 24 | 36 | |
| ERROR | 0 | 0 | 640 | 1 | 5 | 34 | |

\* EXTERNALS \*

| | ⑨ | ⑩ | ⑪ | ⑫ |
|---|---|---|---|---|
| GETJCW | 0 | | | |
| SETJCW | 0 | | | |
| DNTOA | 0 | 0 | 2640 | |
| MESSAGE | 0 | 1104 | 1400 | |
| PRINT | 0 | | | |
| CLEARLINE | 0 | 0 | 2700 | |
| PRINTERROR | 0 | 0 | 400 | |
| FREADDIR | 0 | | | |
| FERROR | 0 | 0 | 3600 | |
| FCHECK | 0 | | | |
| MESSAGE | 0 | 1104 | 1400 | |
| PRINTERROR | 0 | 0 | 400 | |
| QUIT | 0 | | | |
| FEOF | 0 | 0 | 5400 | |
| FREADDIR | 0 | | | |
| FERROR | 0 | 0 | 3600 | |
| FGETINFO | 0 | | | |

USED        5600 ⑬        AVAILABLE        23200 ⑭

Figure 7-2. -LISTRL Command Output

7-21

## Examples of Relocatable Library Management Commands

In the following listing, several commands that manage relocatable libraries are illustrated:

*EXAMPLE:*

| Command | Operation |
|---|---|
| :SEGMENTER | *Accesses the Segmenter* |
| -BUILDRL R1·G1·A1, 500, 1 | *Creates an RL file, named R1.G1.A1. The RL can contain 500 records, and is allocated one disc extent.* |
| -USL XFILE | *Designates the USL file XFILE for management.* |
| -ADDRL XBM | *Adds an RBM (containing one procedure) named XBM from XFILE to the RL file R1.G1.A1.* |
| -RL R2·G1·A1 | *Designates the RL file R2.G1.A1 for management.* |
| -PURGERL UNIT, PROCA | *Deletes a procedure named PROCA from the RL file R2.G1.A1.* |
| -EXIT | *Exits from Segmenter, returning control to the MPE/3000 Command Interpreter.* |

•
•
•

## CREATING AND MAINTAINING SEGMENTED LIBRARIES (SL'S)

To enter commands to create and maintain segmented libraries, the user first accesses the Segmenter by entering the :SEGMENTER command. He may then enter any of the following commands, preceded by a dash (in sessions) as the prompt character. As with USL's and RL's, Segmenter commands are required to create and change SL's because these files are written in a special format. But once an SL is created, it can be referenced by other MPE/3000 commands and intrinsics.

### Creating a Segmented Procedure Library (SL) File

To create a permanent, formatted SL file, the user enters the following command. The user must have *save-access* to the group to which he assigns an SL. The SL will be created with default file-access security.

> -BUILDSL *filereference, records, extents*

| | |
|---|---|
| *filereference* | Any file whose local name is SL, in the *filereference* format. (The user can create an SL file with a local name other than SL, but such a file cannot be searched by the :RUN command or the CREATE or LOADPROC intrinsics.) (Required parameter.) |
| *records* | The total maximum file capacity, specified in terms of 128-word binary logical records. (Required parameter.) |
| *extents* | The total number of disc extents that can be dynamically-allocated to the file as logical records are written to it. The size of each extent is determined by the *records* parameter value divided by the *extents* parameter value. The *extents* value must be an integer from 1 to 16. (Required parameter.) |

### Designating SL's for Management by the User

To enter commands that modify an existing SL in any way, the user must first identify the SL by the -SL command:

> -SL *filereference*

| | |
|---|---|
| *filereference* | The name of the SL to be modified, in the *filereference* format (including group and account identifiers). (Required parameter.) |

Then, all subsequent SL-modification commands will refer implicitly to this SL until a new -SL command is issued. If, however, the user has entered a -BUILDSL command, this acts as an implicit -SL command for the file referenced. That is, all SL-modification commands between a -BUILDSL command and the next -SL command or -BUILDSL command refer to the file specified in the first -BUILDSL command.

### Adding a Procedure Segment to an SL

To add a procedure to the currently-managed SL, the user actually selects the RBM containing this procedure, and all other RBM's sharing the same segment name, from the currently-managed USL, formats these into a segment, and places this segment in the SL. The user must have write-access to the designated SL file. The procedure is added with the following command:

  _-ADDSL   name[,PMAP]_

name        The name of the segment to be added to the SL.  (Required parameter.)

PMAP       An indication that a listing describing the prepared segment will be produced on the device specified in the :SEGMENTER command parameter *listfile*. If this parameter is omitted, the prepared segment is not listed. (Optional parameter.)

### Deleting an Entry-Point or Segment From an SL

To delete an entry-point from a segment in an SL, or an entire segment from an SL, the user can enter the following command. (The segment code actually remains in the SL, but the entry-point name is withdrawn from the entry-point directory. When the last entry-point in a segment is deleted, the entire segment is automatically removed from the SL.)

  _-PURGESL   [unitspec,] name_

unitspec     Is ENTRY, to delete the entry-point identified by *name;* or

             SEGMENT, to delete the segment identified by *name*.  (Optional parameter.)

             The default parameter is ENTRY.

name        The name of the entry-point or segment to be deleted.  (Required parameter.)

### Listing Procedures in an SL

To list the procedures in the currently-managed SL, the user enters:

  _-LISTSL_

An example of the output produced by the -LISTSL command is shown in Figure 7-3. On this listing, significant entries are indicated with arrows, keyed to the discussion below. (All numbers appearing on this listing are octal values.)

| Item No. | Meaning |
|---|---|
| 1 | The name of the SL file (filename.groupname.accountname). |
| 2 | The logical segment number (relative to this SL). |
| 3 | The segment name. |
| 4 | The segment length, in words. |
| 5 | The entry-point name in the segment. |
| 6 | The parameter checking-level of the entry-point. |
| 7 | The callability of the entry point, where<br>C = Callable<br>U = Uncallable |
| 8 | The Segment Transfer Table (STT) number of the entry point. |
| 9 | The entry-point address (relative displacement within the segment). |
| 10 | The name of the external reference. |
| 11 | The parameter checking level of this external reference. |
| 12 | The STT number of this external reference. |
| 13 | If a number appears in this field, it is a (logical) segment number and indicates that this reference has been bound to an entry point within the SL. If a question mark appears in this field, this indicates that this reference has not been bound to any entry point within the SL. |
| 14 | A bit list of the segments referenced within the SL (with the left-most bit corresponding to the *0th* logical segment number): For each bit,<br>1 = Segment referenced.<br>0 = Segment not referenced. |
| 15 | The number of words presently used in the SL file. |
| 16 | The number of words not used, at present, in the SL file. |

Following the line containing the segment number, name, and length, any of the following segment attributes may appear. (These attributes can only be specified during system configuration. All attributes except PRIVILEGED pertain only to the system SL.)

| Attribute | Meaning |
|---|---|
| RESIDENT | Segment permanently resides in main memory. |
| ALLOCATED | Segment is permanently allocated in virtual memory. |
| PRIVILEGED | Segment is executed in privileged mode. |
| SYSTEM | Segment is part of MPE/3000. |

```
SL FILE SCR2.MPE.SYS                    ④①

SEGMENT  0 SEG3              LENGTH  1464
  ②    ③
     ENTRY POINTS   CHECK CAL STT   ADR    ⑨
     SETBIT      ⑤    0    C   20   1020
     PARMCHECK    ⑥   0  ⑦ C  ⑧ 1     0
     GETRECDDISP       0    C   14    737
     FERROR            0    C   40   1303
     REPAIRRECORD•     0    C   12    710
     TESTBIT           0    C   22   1040
     NTOA              0    C   15    754

     SUMBITS           0    C   17   1006
     DNTOA             0    C   16    762
     FWRITEMR••        0    C   31   1167
     MAKEROOMINDL      0    C    3    163


     EXTERNALS   ⑩ CHECK STT SEG
     QUIT      ⑩     0  ⑫ 52   ?    ⑬
     PRINTERROR ⑪    0    51   1 ⬋
     MESSAGE         0    50   1
     FCHECK          0    47   ?
     FGETINFO        0    46   ?
     FREADDIR        0    45   ?
     FWRITEDIR       0    44   ?
     FWRITE          0    43   ?
     ERROR           0    42   1
     DLSIZE          0    41   ?

110  ⑭


SEGMENT  1 SEG4              LENGTH  1424

     ENTRY POINTS   CHECK CAL STT   ADR
     WARN              0    C   10   1356
     ERRORN            0    C    5   1337
     ERRORS            0    C    4   1332
     PRINTWARNING      0    C    3   1235
     DMPY              0    C   11   1363
     WARNS             0    C    7   1351
     MESSAGE           0    C    1   1104
     PRINTERROR        0    C    2   1210
     ERROR             0    C    6   1344

     EXTERNALS   CHECK STT SEG
     CLEARLINE         0    16   0
     PRINT             0    15   ?
     DNTOA             0    14   0
     SETJCW            0    13   ?
     GETJCW            0    12   ?
110
                              ⑮                    ⑯
USED            17400        AVAILABLE       11400
```

Figure 7-3. -LISTSL Command Output

## SETTING RBM INTERNAL FLAGS

In addition to the activity bit associated with every entry-point on an RBM, there is also an *internal flag* that determines whether or not that entry-point is to be placed in the library directory —and thus made available to users and segments within that SL — when the RBM is segmented and added to an SL. (The internal flag is only interrogated when the segment is actually added to an SL.) As an example of how this facility can be used, suppose that a programmer is writing a complicated utility routine that he wants to add to an SL. He writes this routine as a set of procedures to be placed in the same code segment, but wants only the entry-points of certain procedures made available to other users; he does not want them to use the entry-points of the support procedures for the main routine. He can effectively hide these private entry-points from users by using the OPTION INTERNAL statement (SPL/3000 only) in the declarations of the applicable procedures, or by setting the internal flag *on* with the -HIDE command described below. When these procedures are prepared onto a segment and added to an SL, the entry-points designated as internal are not entered in the directory of the library.

To set an internal flag *on*, enter:

_-HIDE  name[(index)]_

name        The name of the entry-point in the currently-managed USL whose internal
            flag is to be set *on*.   (Required parameter.)

index       The index integer further specifying the entry-point *name*; the index may be
            used when the RBM contains more than one entry-point of this name.  If
            *index* is omitted, 0 is assigned by default. (Optional parameter.)

To set an internal flag to *off*, enter the following command:

_-REVEAL  name[(index)]_

name        The name of the entry-point in the currently-managed USL whose
            internal flag is to be set *off*.  (Required parameter.)

index       The index integer further specifying the entry-point *name*; the index may be
            used when the RBM contains more than one entry-point of this name.  If
            *index* is omitted, a value of 0 is assigned.  (Optional parameter.)


## DYNAMIC LOADING OF LIBRARY PROCEDURES

Normally, segments containing library procedures referenced by a program are attached to that program when the program is allocated in virtual memory. However, the programmer may also dynamically attach and detach such procedures while his program is running. He might, for example, decide to do this for a large procedure used optionally and infrequently by his 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, he could keep the virtual memory space used by his program to a minimum most of the time. The procedures are loaded from SL libraries, not from RL libraries (which are used only at program-preparation time).

Procedures are dynamically loaded and unloaded through the intrinsic calls described below.

The user need not dynamically-load procedures declared as externals to his program, because the loader will automatically load them. Dynamic procedure loading is intended for procedures that are not declared external.

### Dynamic Loading

To dynamically load a library procedure (together with external procedures referenced by it), the user issues the LOADPROC intrinsic call.

*INTEGER PROCEDURE*
         | *LOADPROC (procname,lib,plabel);* |

*VALUE lib;*

*BYTE ARRAY procname;*

*INTEGER lib,plabel;*

*OPTION EXTERNAL;*

This intrinsic returns (as the value of LOADPROC), an identity number required for use in unloading the procedure. (If an error occurs during execution of the intrinsic, an error code number rather than the identity number is returned.)

The intrinsic call parameters are

*procname*            A byte array containing the name of the procedure to be loaded. The name must be terminated by a blank.

*lib*                Is 2, to request library searching for the procedure in this order:

- Group Library
- Account Public Library
- System Library; or

1, to request library searching in this order:

- Account Public Library
- System Library; or

0, to request searching of the system library only.

*plabel*           The word to which the procedure's label (P-label) is returned.

The LOADPROC intrinsic returns either of the following condition codes:

CCE          Request granted.

CCG          (Not returned by this intrinsic.)

CCL          Request denied; the value returned to the user's process is an error-code
             number. (This is one of the Loader run-time errors appearing in Figure 10-5C,
             Section X.)

*EXAMPLE:*

*To dynamically load a procedure named PROC1, the user enters the following call. Only the
system library is searched for this procedure. The label of the procedure is returned to the
word LAB. The procedure identity number is returned to the word PNUM.*

**PNUM := LOADPROC (PROC1,0,LAB);**


## Dynamic Unloading

To dynamically unload a procedure, and its referenced external procedures, the user calls
the following intrinsic:

*PROCEDURE*  | *UNLOADPROC (procid);* |

*VALUE procid;*

*LOGICAL procid;*

*OPTION EXTERNAL;*

The *procid* parameter is a word containing the procedure's identity number, returned to the
user's process through the LOADPROC intrinsic.

The UNLOADPROC intrinsic returns one of the following condition codes:

CCE          Request granted.

CCG          (Not returned by this intrinsic.)

CCL          Request denied because of invalid *procid.*

*EXAMPLE*

*To unload the procedure that was dynamically loaded in the previous example, the user issues the following intrinsic call:*

**UNLOADPROC (PNUM);**

# SECTION VIII
# Requesting Utility Operations

MPE/3000 provides commands and intrinsic calls that allow the user to request various utility operations, such as

- Listing date, time, and accounting information

- Determining job status

- Transmitting messages

- Requesting ASCII/binary number conversion

- Reading Input from Job/Session Input Device

- Writing Output to Job/Session List Device

- Obtaining system timer information

- Determining the user's access mode and attributes

- Searching arrays and formatting parameters

- Executing MPE/3000 commands programmatically

- Enabling and disabling error traps

- Requesting program break, termination, or abort programmatically

- Changing the lengths of the User Managed (DL-DB) Area and Stack (Z-DL) Areas by altering DL-DB and Z-DL register off-sets

- Managing interprocess communication through the job control word

- Verifying assignment of diagnostic devices

- Changing terminal speed and echo mode

If the OPTION VARIABLE notation appears in an intrinsic head, some of the intrinsic parameters are optional. In this manual, these optional parameters are indicated in bold face in the intrinsic head formats.

## LISTING DATE, TIME, AND ACCOUNTING INFORMATION

The current date, time, and accounting charges logged against the user's account and group can be displayed for his convenience on the job/session list device.

### Date and Time

The user can print the current date and time, as recorded by the system clock, by entering:

:SHOWTIME

### Accounting Charges

The user can display the total accounting information logged against his log-on account and group. This information includes usage counts and limits regarding permanent file space (in sectors), central processor time (in seconds), and on-line connect-time (in minutes). The file-space usage count reflects file space used as of the present moment, but the central processor time and connect-time usage counts reflect these counts as they were immediately prior to the inception of the current job/session. The user requests this display by entering:

:REPORT

NOTE: Users with System Manager or Account Manager Capability can issue a more explicit form of this command, referencing particular accounts and groups. Additionally, System Manager Users can re-set resource use counts for an account (and for all of its groups). These functions are described in HP 3000 Multiprogramming Executive System Manager/Supervisor Capabilities (03000-90038).

*EXAMPLE:*

*While logged-on under the account GP and the group COMLIB, the user issues a :REPORT command. In the resulting output (shown below), the specific account and group names, and counts and limits, are indicated under the corresponding headings at the top of the listing. (The double asterisks (\*\*) in the three LIMIT fields indicate that no limits exist.)*

```
▶ :REPORT
  ACCOUNT        FILESPACE-SECTORS      CPU-SECONDS       CONNECT-MINUTES
       /GROUP      COUNT      LIMIT    COUNT      LIMIT    COUNT      LIMIT

  GP             27382         **     34417        **     10637        **
       /COMLIB    2201         **      3073        **       790        **
```

## DETERMINING JOB STATUS

A programmer can obtain status information about a job or session with the following command:

$$
\text{:SHOWJOB} \begin{bmatrix} \# \\ jsnumber \\ jsname \end{bmatrix}
$$

| # | A request to display the number of jobs and sessions accessing the central processor (*executing*) and the total number of jobs and sessions in the system (*defined*), in this format. |
|---|---|

$$SESSIONS = executing/defined \quad JOBS = executing/defined$$

| *jsnumber* | The job or session number assigned by MPE/3000, in the format: |
|---|---|

*#Jnnnnn or #Snnnnn*

This entry requests the display of all status information about the indicated job or session.

| *jsname* | The fully-qualified name of the job or session, as entered in the :JOB command or the :HELLO command, in the format |
|---|---|

*[jsname,] username.actname*

This entry also requests display of all status information about the job or session. If several job/sessions coincidentally have the same name, only one of these will be selected (arbitrarily) for information display; if such a case is likely, the user should reference the job/session by its *jsnumber*.

If the :SHOWJOB parameter entered is *jsnumber* or *jsname*, the job status information is displayed in this format:

*jsnumber [jsname,] username.acctname state[inputdev] [listdev] date time*

| *jsnumber* | The job or session number assigned by MPE/3000 (in the format #Jnnnnn or #Snnnnn). |
|---|---|
| *jsname* | The job or session name, if specified by the user in the :JOB or :HELLO command. |
| *username* | The name of the user as defined in the log-on account. |
| *acctname* | The name of the log-on account. |
| *state* | The current status of the job, which will be one of the following: |

| Message | Meaning |
|---|---|
| NOT READY | The job/session is not ready for initiation. |
| WAITING | The job/session is currently being initiated, or has not been initiated because of lack of necessary resources (such as an unavailable list device). |

| Message | Meaning |
|---|---|
| EXECUTING | The job/session has been initiated and is in process. (This does not necessarily imply that a program is currently running for the job/session.) |
| DONE | The job/session is currently terminating, or has terminated. |

*inputdev*        The logical device number of the job/session input device; appears only if the job/session is not DONE.

*listdev*        The job/session list device indicator. If the job/session is NOT READY or WAITING, this is the logical device number or device class index for the list device required. (Device class indexes are followed by the letter *C*; the actual number preceding *C* cannot be simply related to a device or class.) If the job/session is executing, this is the actual logical device number of the list device assigned.

*date*        The date the job/session was initiated in the format:

month/day/year

*time*        The time the job/session was initiated in the format:

hours:minutes

If no parameter is included with the :SHOWJOB command, then *all* status information for *every* job/session in the system is listed (in the same format described above). This information is then followed by a display showing the number of jobs and sessions *executing* and the total number of jobs and sessions *defined* in the system, in the format:

$$SESSIONS\ =\ executing/defined \quad\quad JOBS\ =\ executing/defined$$

*EXAMPLES:*

*Output resulting from three :SHOWJOB commands is shown below. (The entry TASKS = 0/0 in these examples is not pertinent, and is present only to facilitate implementation of later MPE/3000 features.)*

▶ :SHOWJOB #

  SESSIONS= 3/3      JOBS= 1/1  TASKS= 0/0

▶ :SHOWJOB #S29
  #S29      LAZAR.MPE                EXECUTING  15  15    8/16/73, 4:34PM

▶ :SHOWJOB

| JOBNUM | JOBNAME | STATE | IN | OUT | INTRODUCED |
|---|---|---|---|---|---|
| #J6 | YU.LANG | EXECUTING | 6 | 5 | 8/16/73, 4:12PM |
| #S29 | LAZAR.MPE | EXECUTING | 15 | 15 | 8/16/73, 4:34PM |
| #S20 | RON.GP | EXECUTING | 52 | 52 | 8/16/73, 3:31PM |

  SESSIONS= 2/2      JOBS= 1/1  TASKS= 0/0

## TRANSMITTING MESSAGES

Any user can transmit messages from his job or session to other jobs or sessions currently running, by entering the :TELL command. The message appears on the receiving user's standard list device. If a terminal to which a message is sent is currently receiving program output, this message is queued as high as possible among the current input/output requests, but does not interrupt reading or writing in progress. The format of the :TELL command is

$$:TELL \quad \left\{ \begin{array}{l} jsnumber \\ jsname \end{array} \right\} ; \quad message$$

jsnumber          The job or session number assigned by MPE/3000 to the job/session that is to receive the message. It is entered in this format:

$$\#Jnnnnn \quad or \quad \#Snnnnn$$

jsname          The name of the job or session that is to receive the message. It is written as entered in the :JOB command, or the :HELLO command, in the format:

*[jsname,] username.acctname*

If several jobs or sessions coincidentally have the same name, only one of these will be selected (arbitrarily); in such a case, the user should use the *jsnumber* parameter with the :TELL command.

message          The message, comprised of any string of ASCII characters. The message is terminated by the end of the record on which it appears, or by a carriage return. (Required parameter.)

The message appears on the receiving list device in this format:

*FROM/[jsname,] username .acctname/message*

jsname  ⎫          The names of the transmitting job/session and user,
username  ⎬          and the name of the account under which they are
acctname  ⎭          running.

message          The message, in ASCII characters.

> NOTE:  The :TELL command is rejected if the final destination message (FROM/. . . .) exceeds 72 characters.

If the job/session or user designated to receive the message is not running, the transmitting job/session will receive a message indicating this.

*EXAMPLE*

*A user identified as SMITH, running a session named SMITHSES, wants to send a
message to a user identified as BROWN, running a session named BROWNSES.
(Both users are logged-on under account A.)   SMITH enters:*

```
:TELL BROWNSES,BROWN.A;SYSTEM WILL BE SHUT DOWN AT 4:30 TODAY.
```

*BROWN receives:*

```
FROM/SMITHSES,SMITH.A/SYSTEM WILL BE SHUT DOWN AT 4:30 TODAY.
```

The user can send a message to the MPE/3000 operator's console by entering the :TELLOP command:

> *:TELLOP message*

*message*        The message to be transmitted. The *message* is limited to approximately 57 characters.

## REQUESTING ASCII/BINARY NUMBER CONVERSION

The user can convert decimal or octal numbers represented in ASCII code to their binary equivalents, and vice-versa, with the intrinsics described below.

## Converting Numbers from ASCII to Binary Code

To convert an ASCII-coded number to its binary equivalent, the user issues the BINARY intrinsic call.

*LOGICAL PROCEDURE*     | *BINARY (string, length);* |

*VALUE length;*

*BYTE ARRAY string;*

*INTEGER length;*

*OPTION EXTERNAL;*

This intrinsic returns the converted value to the user's program (as the value of BINARY). In this intrinsic call, the parameters are

| | |
|---|---|
| *string* | A byte array containing the signed octal or decimal number (in ASCII code) to be converted. If the character string in this array begins with a percent sign (%), it is treated as an octal value; if the string begins with a plus sign, minus sign, or digit, it is treated as a decimal value. |
| *length* | An integer representing the length (number of bytes) in the byte array containing the ASCII-coded value. If the value of *length* is 0, the intrinsic returns 0 to the calling process. If the value of *length* is less than 0, the intrinsic aborts. |

The execution of the BINARY intrinsic can result in one of these three condition codes:

| | |
|---|---|
| CCE | Successful conversion; a one-word binary value is returned to the user's program. |
| CCG | A word overflow, possibly resulting from too many characters, occurred in the word returned. |
| CCL | An illegal character was encountered in the byte array specified by *string*. This could include the digits *8* or *9* specified in an octal value. |

*EXAMPLE:*

*Suppose the byte array STRINGA contained the ASCII characters "%16". The user wants to convert this string to its binary equivalent and store the result in the word RESULT. He issues the following call:*

**RESULT := BINARY (STRINGA,3);**

*The binary value 1110 (decimal 14) is stored in RESULT.*

The user can request double-integer ASCII-to-binary code conversion by calling the DBINARY intrinsic. Except for a double-word result, this intrinsic is identical to the BINARY intrinsic just described.

*DOUBLE PROCEDURE*  |  *DBINARY (string, length);*

*VALUE length;*

*BYTE ARRAY string;*

*INTEGER length;*

*OPTION EXTERNAL;*

This intrinsic returns (as the value of DBINARY) the converted double-word value. The parameters and possible condition codes are the same as those of the BINARY intrinsic.

## Converting Numbers from Binary to ASCII Code

The user can convert any 16-bit binary number to its ASCII equivalent by entering the ASCII intrinsic call.

*INTEGER PROCEDURE*  |  *ASCII (word, base, string);*

*VALUE word, base;*

*LOGICAL word;*

*INTEGER base;*

*BYTE ARRAY string;*

*OPTIONAL EXTERNAL;*

This intrinsic returns to the caller (as the value of ASCII) the number of characters in the final conversion. The parameters are

word            The number to be converted to ASCII code.

base            An integer indicating octal or decimal conversion.

                8 = octal
                10 = decimal

                If neither 8 or 10 are entered in this parameter, the intrinsic aborts.

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

                For octal conversion (base = 8), six characters (including leading zeros) are always returned in string, showing the octal representation of word. For octal conversions, the result of ASCII is the number of significant (right-justified) characters in string. (If word = 0, the result of the intrinsic is 1.)

                For decimal conversions (base = 10), word is considered as a 16-bit, 2's-complement integer (ranging from −32768 to +32767). Leading zeros are removed and the ASCII result is left-justified. In decimal conversions, if the value is negative, the first byte of string contains a minus sign. If word contains a single zero, only 0 is returned to string. For decimal conversions, the result of ASCII is the total number of characters in string (including the sign). (If word equals 0, the result of ASCII is 1.)

With this intrinsic, the condition code remains unchanged.


*EXAMPLE:*

*Suppose the word BWORD contained %177666. The user wants to convert this value to its decimal equivalent, returning it to the byte-array RSTRING and returning the number of characters in the conversion to the word LEN. He issues this call:*

**LEN := ASCII (BWORD,10,RSTRING);**

*Then, RSTRING contains "−74" and LEN contains 3.*

The user can convert a 32-bit (double-word) binary number to its ASCII equivalent by entering the DASCII intrinsic call.

*INTEGER PROCEDURE*      | *DASCII (dword, base, string);* |

*VALUE dword, base;*

*DOUBLE dword;*

*INTEGER base;*

*BYTE ARRAY string;*

*OPTION EXTERNAL;*

The DASCII intrinsic returns to the caller the number of characters in the final conversion. The parameters are

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

*base*         The same as the corresponding parameters for the ASCII intrinsic, except that *string* always receives 11 characters for octal conversions, and may

*string*      receive up to 11 characters for decimal conversions.

With this intrinsic, the condition code remains unchanged.


## TRANSMITTING PROGRAM INPUT/OUTPUT (FROM JOB/SESSION INPUT/LIST DEVICES)

In addition to the FREAD and FWRITE intrinsics discussed in Section VI, MPE/3000 provides two other intrinsics that permit the user to read information from the job/session input device or write information to the job/session listing device. (The user does not issue FOPEN against these devices prior to issuing these intrinsics.) These commands are primarily used by programmers designing their own subsystems or command executors, to read commands or write error messages. Other programmers typically use the file system commands for general input, and listing output.


### Reading Input

The job/session input device acts as the source of all MPE/3000 commands relating to a job or session, and the primary source of all ASCII information input to the job or session. This is normally a card reader for jobs and a terminal for sessions.

The programmer can read a string of ASCII characters from a job/session input device into an array in his program by calling the READ intrinsic. (This intrinsic call is equivalent to issuing an FREAD call against the file $STDIN.) Basic line editing such as cancellation of lines and backspacing are performed automatically by the input/output driver. If the input device is a terminal and it is in full-duplex mode and the echo facility is on, or if the terminal is in half-duplex mode, the characters read are printed.

*INTEGER PROCEDURE*        | *READ (message,expectedl);* |

*VALUE expectedl;*

*ARRAY message;*

*INTEGER expectedl;*

*OPTION EXTERNAL;*

The READ intrinsic returns to the caller the positive length of the input actually read, in bytes (if *expectedl* is negative) or words (if *expectedl* is positive), as the value of READ. The parameters for this intrinsic call are

*message*      The array into which the ASCII characters are read.

*expectedl*      An integer denoting the maximum length of the array *message*. If *expectedl* is negative, this specifies the length in bytes; if *expectedl* is positive, it indicates the length in words. 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.

Device mode settings specified through the FCONTROL or FSETMODE intrinsics on the file $STDIN appropriately affect transactions using the READ intrinsic.

The READ command can result in the following condition codes:

CCE      The record was read into the area specified by *message*.

CCG      A record with a colon in the first column was encountered, signalling the end of data.

CCL      A physical input/output error occurred, and *message* is not valid; further error analysis, through the FCHECK intrinsic, is not possible.

*EXAMPLE:*

*To read information from a record input through the job input device into an array named INBUFF whose size is 72 bytes, the user issues the following call:*

```
LEN := READ (INBUFF,-72);
```

*LEN will contain the (positive) number of characters actually transmitted.*

## Writing Output to the Listing Device

All MPE/3000 commands issued and any system messages for the user are copied to the job/session listing device; this device is the primary destination of all ASCII output. Normally, this device is a line printer for jobs and a terminal for sessions.

The programmer can write a string of ASCII characters from an array in his program to the job/session listing device by calling the PRINT intrinsic. This intrinsic call is equivalent to issuing a FWRITE call against the file $STDLIST.

PROCEDURE

PRINT (message,length,control);

VALUE length,control;

ARRAY message;

INTEGER length,control;

OPTION EXTERNAL;

In this intrinsic call, the parameters are

message       The array from which the characters are output.

length        An integer denoting the length of the output string to be written. If
              length is negative, this specifies the length in bytes; if it is positive,
              it indicates the length in words.

control       An integer representing a carriage control code (as described under the
              discussion of the FWRITE intrinsic in Section VI).

Device control settings specified through the FCONTROL or FSETMODE intrinsics on the file $STDLIST appropriately affect transactions using the PRINT intrinsic.

A secondary entry-point to the PRINT intrinsic, PRINT', coincides with the main entry point and is provided primarily for use in compilers that require a reserved intrinsic inaccessible to their users.

The condition codes returned are

CCE     The intrinsic was successfully executed.

CCG     End-of-data was encountered.

CCL     Physical input/output error. The character string specified was not successfully written. Further error analysis, through the FCHECK intrinsic, is not possible.

*EXAMPLE:*

*The following intrinsic call prints the number of characters specified by XCHARS from the array labeled OUTBUFF onto the job listing device. The first character of the string is used for carriage control.*

**PRINT (OUTBUFF,XCHARS,1);**

## Writing Output to the Operator's Console

The programmer can also transmit information from an array to the operator's console. He does this with the PRINTOP intrinsic call:

*PROCEDURE*      | *PRINTOP (message,length,control);* |

*VALUE length;*

*ARRAY message;*

*INTEGER length, control;*

*OPTION EXTERNAL;*

The parameters and condition codes for this intrinsic are identical to those for the PRINT intrinsic, except that *message* is limited to approximately 57 characters.

## OBTAINING SYSTEM TIMER INFORMATION

The programmer can request the return of system timer information to his program with the intrinsic calls described below.

### System Timer Bit-Count

The user can return to his program a 31-bit logical quantity representing the current system timer and timer overflow count. This count can be used in routines that generate random numbers, or in measuring the real-time (wall-clock time) elapsed between two events. The resolution of the system timer is one millisecond; in other words, readings taken within a one-millisecond period may be identical. (The user should bear in mind that this timer-count is a hardware representation that is meaningless except for its role in random-number initialization.) The count is obtained with this intrinsic:

*DOUBLE PROCEDURE*  | *TIMER;* |

*OPTION EXTERNAL;*

The timer count is returned as the value of TIMER. The condition code is not changed by this intrinsic.


## Current Time

The user can request that the current time, as monitored by the system timer, be returned to his process as a triple word. The first word contains the year (last two digits) and day-of-the-year; the second word contains the hour (24-hour clock) and minute; the third word contains the second and tenth-of-a-second. The time is requested with the CHRONOS intrinsic.

*LONG PROCEDURE*  | *CHRONOS;* |

*OPTION EXTERNAL;*

The format of the returned information is

| 0 | 6/7 | 15 |

| Last 2 digits of year | Day of year 7/8 |
|---|---|
| Hour | Minute |
| Second | Tenth-of-Second |

The time is returned as the value of CHRONOS. The condition code is not changed by this intrinsic.


## OBTAINING PROCESS RUN-TIME (USE OF THE CENTRAL PROCESSOR)

The user can obtain a double integer showing the duration, in milliseconds, that the process has been running. He does this by issuing the PROCTIME intrinsic call.

*DOUBLE PROCEDURE*  | *PROCTIME;* |

*OPTION EXTERNAL;*

The process run-time is returned as the value of PROCTIME. The condition code is not changed by this intrinsic.

## 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 describing the user. This information is requested with the WHO intrinsic call:

PROCEDURE      WHO *(*mode,capability,lattr,usern,groupn,acctn,homen,termn*)*;

*LOGICAL* **mode,termn**;

*DOUBLE* **capability, lattr;**

*BYTE ARRAY* **usern,groupn,acctn,homen**;

*OPTION VARIABLE,EXTERNAL*;

The parameters in this intrinsic call are as follows. All are optional parameters specifying locations to which information is to be returned. If any parameter is omitted, the corresponding information is not returned, by default.

**mode**        A word to which the current user's mode of access will be returned. In this word, the bits will have the following meaning. (Bit positions are numbered from left to right, 0 through 15.)

| Bits | | | Access Mode |
|------|---|---|-------------|
| (15:1) | 1 | = | The job/session input file and job/session list file form an *interactive* pair. (A dialogue can be established between a program (displaying information on the job list device) and a person (responding through the job input device).) |
| | 0 | = | The job/session input file and job/session list file are *not* interactive. |
| (14:1) | 1 | = | The job/session input file and job/session list file form a *duplicative* pair. (Images on the job input device are automatically duplicated on the job list device.) |
| | 0 | = | The job/session input file and job/session list file are *not* duplicative. |
| (12:2) | 01 | = | The user is accessing the system through a *session*. |
| | 10 | = | The user is accessing the system through a *job*. |
| (0:12) | | | (These bits are not used; the intrinsic always sets them to zero.) |

**capability**          A double-word where the user's file access, user, and capability class attributes will be returned. In the first word, possession of the following file-access and user attributes is indicated by the corresponding bit set *on* (equal to 1).

| | | |
|---|---|---|
| Bit (15:1) | = | Ability to save files (declare them permanent). |
| Bit (14:1) | = | Ability to acquire non-sharable devices. |

> File Access Attributes

| | | |
|---|---|---|
| Bit (7:7) | = | (Not used.) |
| Bit (6:1) | = | (Not used.) |
| Bit (5:1) | = | System supervisor. |
| Bit (4:1) | = | Diagnostician |
| Bit (3:1) | = | Group librarian. |
| Bit (2:1) | = | Account librarian. |
| Bit (1:1) | = | Account manager. |
| Bit (0:1) | = | System manager |

> User Attributes

In the second word, possession of the user's capability-class attributes is indicated by the corresponding bit set on:

| | | |
|---|---|---|
| Bit (15:1) | = | Process handling. |
| Bit (14:1) | = | Extra-Data segments. |
| Bit (13:1) | = | (Not used; the intrinsic sets this bit to zero.) |
| Bit (12:1) | = | Exclusive simultaneous use of more than one system resource. (Multiple RIN's) |
| Bit (11:1) | = | (Not used; the intrinsic sets this bit to zero.) |
| Bit (10:1) | = | (Not used; the intrinsic sets this bit to zero.) |
| Bit (9:1) | = | Privileged-mode operation. |
| Bit (8:1) | = | Interactive (session) access. |
| Bit (7:1) | = | Batch (job) access. |
| Bit (6:1) | = | (Not used.) |
| Bit (0:6) | = | (Not used; the intrinsic sets these bits to zero.) |

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

usern                 An 8-byte array where the user's name is returned.

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

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

homen                 An 8-byte array where the name of the user's home group is returned.

termn                 A word where the logical device number of the job/session input
                      device is returned.

The names in *usern*, *groupn*, *acctn*, and *homen* are are returned left-justified, padded at
the right with blanks.

The condition code is not changed by this intrinsic.


*EXAMPLE*

*Suppose the programmer wants to return to his program the attributes assigned to the user by
the system.  He wants this information stored in the areas noted:*

| Characteristic | Location Where Stored |
|---|---|
| *Mode* | *AMODE* |
| *Capability* | *CAPLIST* |
| *User Name* | *UNAME* |
| *Account Name* | *ANAME* |
| *Terminal Address* | *TADDR* |

*He can accomplish this by entering:*


**WHO (ADMODE,CAPLIST,,UNAME,,ANAME,,TADDR);**

## SEARCHING ARRAYS

Occasionally, a user constructs byte arrays whose contents he may later want to search for specified entries or names. A dictionary of commands designed by the user is one such example. The SEARCH intrinsic can assist the user with specially-formatted arrays consisting of sequential entries, each including:

- One byte specifying the length (in bytes) of the entire entry — the length includes this byte plus all the information in the following byte areas

- One byte specifying the *length* of the "name" (in bytes) for which the search is performed

- A byte string forming the name for which the search is performed — in a command dictionary, this is the command name

- An optional byte string containing a definition — in a command dictionary, this would be command-relevant information

The entry-number of the first entry in such a dictionary is *one*. The last entry is indicated by an entry-length of *zero*.

The user can request the search of such an array for a specified name with the SEARCH intrinsic call. A simple linear search is then performed, with the name (specified as a byte-array by the user) compared against the byte-array forming the name in each entry. (Because the search is linear, the most frequently-used name 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 user's program. If the name is not found, a zero is returned. At his option, the user can also request the return of a pointer to the definition information for the name.

The search is requested with the SEARCH intrinsic call:

*INTEGER PROCEDURE*  | *SEARCH (target,length,dict,***defn***);*

*VALUE length;*

*BYTE ARRAY target,dict;*

*INTEGER length;*

*BYTE POINTER* **defn***;*

*OPTION VARIABLE,EXTERNAL;*

This intrinsic returns (as the value of SEARCH) the entry number of the definition. In this intrinsic call, the parameters are

*target*        The byte array containing the name for which the search is performed.

*length*        An integer specifying the length, in bytes, of the byte-array *target*.

*dict*        The specially-formatted byte-array in which *target* is sought.

**defn**        A word to which is returned the relative byte address of the definition portion of the entry sought in the array. If omitted, this address is not returned.

The intrinsic searches *dict* for a word matching *target*, and returns the corresponding entry number to the user's process. If the matching word is not found, an entry number of zero is returned.

The condition code is not changed by this intrinsic.


*EXAMPLE:*

*Consider a byte array, wherein the relationship of each entry to its user-relevant definition is:*

| Entry | User-Relevant Definition |
|---|---|
| *THIS* | *200* |
| *IS* | |
| *AN* | *"XYZ"* |
| *EXAMPLE* | *"X"* |

*The byte-array itself, named DEFF, is structured as follows:*

| | |
|---|---|
| 7 | 4 |
| T | H |
| I | S |
| 200 | 4 |
| 2 | I |
| S | 7 |
| 2 | A |
| N | X |
| Y | Z |
| 10 | 7 |
| E | X |
| A | M |
| P | L |
| E | X |
| 0 | |

*Suppose that the user wants to search the byte-array DEFF for the byte-array NAME (containing "AN"). Suppose, also, that NLENGTH is the length of the array NAME (2 bytes). The byte-address of the definition sought is to be returned to the word DEFADR. The entry-number corresponding to the definition is to be returned (as the value of SEARCH) to the word ENUM. The user issues the following call:*

**ENUM := SEARCH (NAME,NLENGTH,DEFF,DEFADR);**

*When this intrinsic is executed, ENUM contains 3 and DEFADR contains the byte pointer to "XYZ."*

## FORMATTING COMMAND PARAMETERS

The programmer can, within his program, extract and format for execution the parameters of a command (that is *not* an MPE/3000 command) through the MYCOMMAND intrinsic call. This intrinsic also allows the programmer, at his option, to request searching of a byte array (serving as a command dictionary) for a specified command.

*INTEGER PROCEDURE*   *MYCOMMAND (comimage,*delimiters,*maxparms, numparms,parms,*dict,defn*);*

*VALUE maxparms;*

*BYTE ARRAY comimage,*delimiters,dict;

*INTEGER maxparms,numparms;*

*DOUBLE ARRAY parms;*

*BYTE POINTER* defn;

*OPTION VARIABLE,EXTERNAL;*

If *dict* is specified, the command entry number is returned as the value of MYCOMMAND. In this call, the following parameters apply:

*comimage*          A byte array that contains either:

- A command name (expected if the *dict* parameter is specified), followed by parameters, followed by a carriage return. The command name is delimited from the parameters by the first special character. The parameters will be formatted and the byte array specified by *dict* will be searched for a name matching the command.

- Only command parameters (expected if the *dict* parameter is not specified), followed by a carriage return. These parameters will be formatted.

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

delimiters          A word containing a string of up to 32 legal delimiters (each of which is an ASCII special byte). The last of these must be a carriage return. Each delimiter is later identified by its position in this string. If this parameter is omitted, the delimiter array "comma, equal, semicolon, carriage-return" is used.

| | |
|---|---|
| *maxparms* | An integer specifying the maximum number of parameters expected in comimage. |
| *numparms* | A word to which is returned the number of parameters actually found in comimage. |
| *parms* | A double array of *maxparms* double words that, on return, delineates the parameters. When the intrinsic is executed, the first *numparms* double words are returned to the user's process in this array, with the first double word corresponding to the first parameter, the second double word corresponding to the next parameter, and so forth. The parameter fields of *comimage* are delimited by the delimiters specified in *delimiters*. In formatting, leading and trailing blanks around each delimiter are removed and lower-case letters are upshifted. Each double word in the array named by *parms* contains the following information: |

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

Word 2: Contains bits that describe the parameter:

Bit (11:5) denotes the delimiter number in *delimiters* (starting at zero).

Bit (10:1) if *on*, indicates that the parameter contains special characters.

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

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

Bit (0:8) indicates the length of the parameter, in bytes. This value is zero if the parameter is omitted.

| | |
|---|---|
| **dict** | 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 SEARCH. (Actually, the command, delimited by a blank, is extracted from *comimage*, and the SEARCH intrinsic is called with the command used as the *target* parameter in SEARCH.) If the command 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 is not found in *dict*, the parameters specified in *comimage* are *not* formatted. |
| **defn** | A word to which is returned the relative address of the definition portion of the command entry in *dict*. |

The MYCOMMAND intrinsic call can result in the following condition code settings:

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

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

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

*EXAMPLE:*

*The user wants to parse PARMARRAY, allowing comma, semicolon, period, and carriage-return as delimiters. These delimiters are found in the array DELS, as follows:*

| , | ; |
|---|---|
| . | (CR) |

*The user wants to specify that no more than 10 parameters are expected in PARMARRAY, and that the number of parameters actually found in PARMARRAY be returned to NUM. The delineation information for the parameters is to be returned to the double-array PARMS. He issues the following call:*

**MYCOMMAND (PARMARRY,DELS,10,NUM,PARMS);**

*Suppose PARMARRAY starts at DB-relative Address %1200 and contains the following ASCII image. (The character "Δ" represents a blank and* (CR) *is a carriage-return.)*

ΔP1,ΔPPΔ.;P3*; 12     (CR)

*When the intrinsic is executed, NUM will contain the value 5, and the double-array PARMS will contain the following:*

| | | |
|---|---|---|
| %1201 | | |
| 2 | 6 | 0 |
| %1205 | | |
| 2 | 4 | 2 |
| %1211 | | |
| 0 | 0 | 1 |
| %1212 | | |
| 3 | 7 | 1 |
| %1216 | | |
| 2 | 2 | 3 |

*(Byte address.)*

*(Length = 2; Alphanumeric characters; 0th delimiter in DELS.)*

*(Omitted Parameter.)*

Bits     0          8  11   15

## EXECUTING MPE/3000 COMMANDS PROGRAMMATICALLY

The COMMAND intrinsic call allows the user to programmatically request the execution of an MPE/3000 command. The command image is passed to the intrinsic, which searches the system command dictionary for a command of the same name, and executes it. When command execution is completed, or when an error is detected during this execution, control returns to the calling process. Commands that cannot be executed programmatically are indicated in Appendix B.

> *NOTE:   Warning messages issued by the command executor are transmitted to $STDLIST, with no programmatic indication to the calling process.*

PROCEDURE    | COMMAND (comimage,error,parm); |

BYTE ARRAY comimage;

INTEGER error;

INTEGER parm;

OPTION EXTERNAL;

The parameters for the COMMAND intrinsic call are

comimage    A byte array that contains an ASCII string making up a command and
parameters terminated by a carriage return. No prompt character,
however, should be included in this string. The *comimage* array may
be altered by the COMMAND intrinsic (for example, in upshifting),
but will be returned in a form that can be re-submitted to this
intrinsic without adjustment.

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

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

The following condition codes can result from the COMMAND intrinsic:

CCE    The command was successfully executed.

CCL    The command was an undefined command.

CCG    An executor-dependent error, such as an erroneous parameter, prevented
execution of the command. The location specified by the *error* parameter
contains the numeric error code. The location specified by *parm* (if not
zero) contains the erroneous parameter element.

*EXAMPLE:*

*Suppose the user wants to programmatically execute the command    :SHOWTIME.
All characters for this command except the prompting colon are contained in the
byte-array COMD.   Any error code is to be returned to ECODE.   Since the
:SHOWTIME command has no parameters, no information will be returned to
EPARM.   The user issues the following call:*

**COMMAND (COMD,ECODE,EPARM);**

*The date and time are printed on the job/session list device.*

## ENABLING AND DISABLING TRAPS

Whenever a major error occurs during the execution of a hardware instruction, a procedure
from the System Library, or an intrinsic called by the user, normally the user's program is
aborted and an error message is output.  The user can, however, avoid immediate abort by
arming any of three software traps provided by MPE/3000:

- 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 callable system
  intrinsic

When an error occurs, the corresponding trap, if armed, suppresses output of the normal error
message, transfers control to a *trap procedure* defined by the user, 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 desired by the user.
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, the user can specify that his
process be aborted when control exits from the trap procedure.)

Traps can be invoked from within trap procedures.

### Arithmetic Trap

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 armed, receives an internal
interrupt signal from a hardware trap when an error is encountered, and transfers control to
the user's trap procedure.

When the user's process begins execution, all hardware trap set interrupt signals are automatically enabled, but the software trap is disarmed, permitting any hardware error to abort the process. Through intrinsic calls, however, the user 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 the user's trap procedure.

To enable or disable the internal interrupt signals from all hardware arithmetic traps, the user enters the ARITRAP intrinsic call. (The overflow bit in the status register is cleared, for the calling process.)

*PROCEDURE*  | *ARITRAP (state);*

*VALUE state;*

*LOGICAL state;*

*OPTION EXTERNAL;*

In this intrinsic call, *state* is TRUE (Bit 15 = 1) to enable the signals from all traps, and FALSE (Bit 15 = 0) to disable them.

The following error conditions can result from the ARITRAP intrinsic:

CCE    Request is granted; the trap set signal was originally disabled.

CCG    Request is granted; the trap set signal was originally   enabled.

CCL    (Not returned by this intrinsic.)

To arm the software arithmetic trap, the user enters the XARITRAP intrinsic call:

*PROCEDURE*  | *XARITRAP (mask,plabel,oldmask,oldplabel)  ;*

*VALUE mask,plabel;*

*INTEGER mask,plabel,oldmask,oldplabel;*

*OPTION EXTERNAL;*

The parameters of the XARITRAP call are

*mask*
A word-mask that selects which hardware traps will invoke the software trap, and which will not. Only the five rightmost bits of the word forming the mask are used. (The setting of the other bits is not significant, but it is recommended that they be set to zero. Thus, octal values up to %37 are suggested for this parameter.) If a bit is on, the corresponding hardware trap activates the software trap; otherwise, it does not. If all bits are set to zero, the software trap is *disarmed.*

| Bit | Hardware Error Trap |
|-----|---------------------|
| 15 | Floating-point divide by zero. |
| 14 | Integer divide by zero. |
| 13 | Floating-point underflow. |
| 12 | Floating-point overflow. |
| 11 | Integer overflow. |

*plabel*
The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is *disarmed.* (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/3000) by placing a @ before the procedure name, as shown in the example below.)

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

*oldlabel*
A word in which the previous *plabel* is returned to the user's program.

The following condition codes can result from the XARITRAP intrinsic call:

CCE     Request granted, software trap armed.

CCG     Request granted; software trap disarmed.

CCL     Illegal *plabel* (for example, not an external type label); no action taken.

*EXAMPLE:*

*To arm the software arithmetic trap so that only floating-point division by zero alone causes a branch to the post-trap procedure RETRY, the user can enter the following call. The word-mask previously specified is returned to the location OMASK, and the previous plabel is returned to the location OLABEL.*

**XARITRAP (%1,@RETRY,OMASK,OLABEL);**

## Library Trap

The software library trap reacts to major errors that occur during execution of procedures from the System Library. When the user's program begins execution, this trap is automatically disarmed. The user can arm (or disarm) it with the XLIBTRAP intrinsic call. When armed, the library trap passes control to a trap procedure in the event of an error. This procedure, in turn, returns to the user's program four words containing the stack marker created when the library procedure was called by the user's 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 job/session at the user's option. The trap procedure is defined by the user, but it must conform to the special format discussed in the manual *HP 3000 Compiler Library (03000-90009)*.

The format of the XLIBTRAP intrinsic call is

*PROCEDURE*   XLIBTRAP (plabel,oldplabel)  ;

*VALUE plabel;*

*INTEGER plabel,oldplabel;*

*OPTION EXTERNAL;*

*plabel*        The external-type label of the user's trap procedure. If the value of this entry is 0, the trap is *disarmed*.

*oldlabel*      A word in which the previous *plabel* is returned to the user's program.

These condition codes can result from the XLIBTRAP intrinsic call:

CCE    Request granted, trap armed.

CCG    Request granted, trap disarmed.

CCL    Illegal *plabel*, no action taken.

*EXAMPLE:*

*To arm a disarmed library trap so that, if a library procedure error occurs, control is trans-ferred to the post-trap procedure RTNX, the user enters the following call. (The old plabel of the trap procedure is returned to OLDLAB.)*

**XLIBTRAP (●RTNX,OLDLAB);**

## System Trap

The software system trap reacts to errors occurring in intrinsics called by the user's program. Typical errors are

- Illegal access (an attempt by the 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 the user is either illegal or outside the constraints imposed by MPE/3000).

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

The system trap is armed or disarmed by the XSYSTRAP intrinsic call:

*PROCEDURE*  | *XSYSTRAP (plabel,oldlabel)  ;*

*VALUE plabel;*

*INTEGER plabel,oldlabel;*

*OPTION EXTERNAL;*

*plabel*      The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disarmed.

*oldlabel*    A word in which the previous *plabel* is returned to the user's process.

These condition codes can result from the XSYSTRAP intrinsic call:

CCE      Request granted, trap armed.

CCG      Request granted, trap disarmed.

CCL      Illegal plabel, no action taken.

## CONTROL-Y Traps

In addition to the error traps just discussed, a user running an interactive process can arm a special trap that transfers control from the currently-executing program or procedure to a trap procedure whenever a CONTROL-Y subsystem break signal is entered from the terminal during a *session*. (On most terminals, the signal is transmitted by striking the *Y*-key while depressing the *control* key.) When more than one process is currently running within the user's process tree structure, the CONTROL-Y signal interrupts the last process to arm the trap. (The trap is armed by issuing the XCONTRAP intrinsic call.)

When a process is interrupted by a CONTROL-Y signal, the following occurs:

1. The input/output transactions pending between the process and the terminal are halted and flagged as though all were completed successfully.

2. Control is transferred to the trap procedure defined by the user, with which he can now interact. (The trap procedure executes in the same mode (privileged or non-privileged) as the user program that was interrupted.)

3. Control returns from the trap procedure to the interrupted program or procedure. If the interrupted program or procedure was awaiting completion of input/output (reading from or writing to the terminal) when the CONTROL-Y signal was received, the FREAD or FWRITE intrinsic that was being executed is flagged as successfully completed when control returns from the trap procedure. If the CONTROL-Y signal was received during reading, the number of characters typed in *before* this signal is returned to the user as the value of FREAD. (The carriage position is unchanged.)

If the user sends another CONTROL-Y signal, it will be ignored unless a call to the RESETCONTROL intrinsic was issued at some point prior to the signal.

If the user sends a CONTROL-Y signal while MPE/3000 system code is executing on his behalf, no interrupt occurs until the process resumes execution of the user's code; then control transfers to the trap procedure. This protects the MPE/3000 System operation.

When a session is initiated, the CONTROL-Y trap is disarmed. The user arms this trap by issuing the XCONTRAP intrinsic call. This call takes effect on the file $STDIN, when this filename indicates a terminal.

PROCEDURE

XCONTRAP (plabel, oldplabel);

VALUE plabel;

INTEGER plabel, oldplabel;

OPTION EXTERNAL;

The parameters are

plabel            The external-type label of the user's trap procedure. If the value of this entry is 0, the software trap is disarmed.

oldplabel         A word in which the previous plabel is returned to the user's process.

These condition codes can result from the XCONTRAP intrinsic:

CCE               Request granted, trap armed.

CCG               Request granted, trap disarmed.

CCL               Illegal plabel (for example, the plabel is not an external-type label), or $STDIN is not a terminal; no action taken.

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

PROCEDURE

RESETCONTROL;

OPTION EXTERNAL;

The following condition codes are returned by this intrinsic:

CCE               Request granted.

CCG               (Not returned by this intrinsic.)

CCL               The request was not granted because the trap procedure was not invoked.

## Trap Procedure Execution

When writing trap procedures, the user should bear the following information in mind.

**ARITHMETIC TRAPS.** When a software arithmetic trap procedure is executed, a one-word parameter that denotes the type of hardware trap invoked is obtainable in the address (Q-4).

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

|  | User Program |
|---|---|
| Q-4 | Hardware Trap Type Parameter |
| Q | (Stack Marker) |
| S | Arithmetic Trap Procedure |
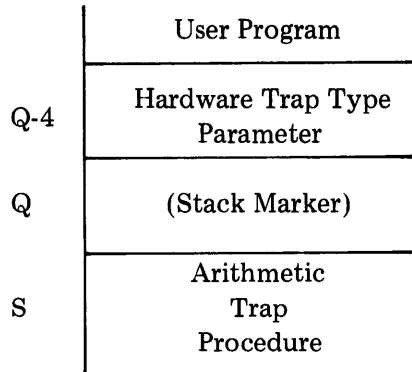
The parameter in Q-4 has the same bit configuration as the mask for the XARITRAP intrinsic.

A suitable declaration for a user's arithmetic trap procedure might be

*PROCEDURE*   ┌─────────────────────────────────┐
              │ *ARITHMETICTRAP (parameter);* │
              └─────────────────────────────────┘

*VALUE parameter;*

*LOGICAL parameter;*

**LIBRARY TRAPS.** When a library trap procedure is invoked, the condition of the stack is

|  | User's Program |
|---|---|
| Q' -3 |  |
| Q' -2 |  |
| Q' -1 |  |
| Q' |  |
|  | Compiler Library Routines |
| Q-6 | *USERSTACK* |
| Q-5 | *ERRORCODE* |
| Q-4 | *ABORTFLAG* |
|  | User's Trap Procedure |
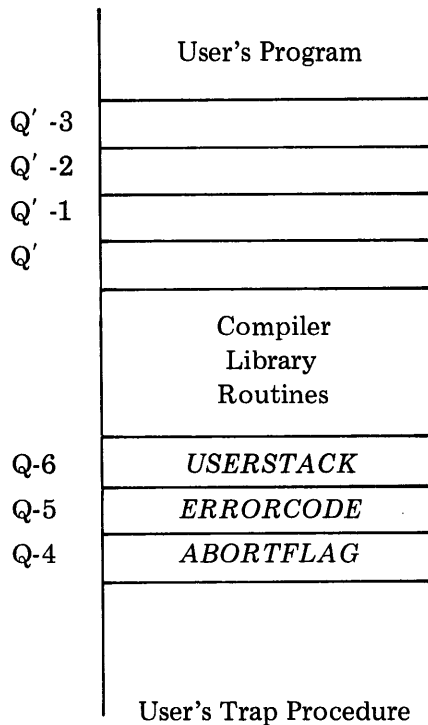
USERSTACK            A word pointer to the base of the stack marker placed on the stack
                     when the user's program called the compiler library.

ERRORCODE            A number indicating the type of compiler library error, described in
                     *HP 3000 Compiler Library (03000-90009)*.

ABORTFLAG            A value set before the user exists from the trap procedure.  If TRUE,
                     the compiler library aborts the program with the standard error
                     message (just as if no trap procedure had been executed).  If FALSE,
                     the compiler library does not abort the program and no error message
                     is printed;  in this case, the compiler library attempts error-recovery.

A suitable declaration for a user's library trap procedure is

*PROCEDURE*      *LIBRARYTRAP (USERSTACK,ERRORCODE,ABORTFLAG);*

*ARRAY USERSTACK;*

*INTEGER ERRORCODE;*

*LOGICAL ABORTFLAG;*

When execution of the library trap procedure is complete, control returns to the library
routine that was interrupted.

## System Traps

When a system trap procedure is executed because of an abort condition arising in a callable system intrinsic, the stack is readjusted to provide an eight-word parameter group between the intrinsic parameters and the stack marker.

| | | |
|---|---|---|
| | User Program | |
| | *N* Words for the Callable Intrinsic | } *N* Words |
| Q-11 | *Parameter 1* | |
| Q-10 | *Parameter 2* | |
| Q-9 | *Parameter 3* | |
| Q-8 | *Parameter 4* | |
| Q-7 | *Parameter 5* | Eight-Word Parameter Group |
| Q-6 | *Parameter 6* | |
| Q-5 | *Parameter 7* | |
| Q-4 | *Parameter 8* | |
| Q-3 | | |
| Q-2 | Stack Marker | |
| Q-1 | | |
| Q | | |
| | System Trap Procedure | |
| S | | |
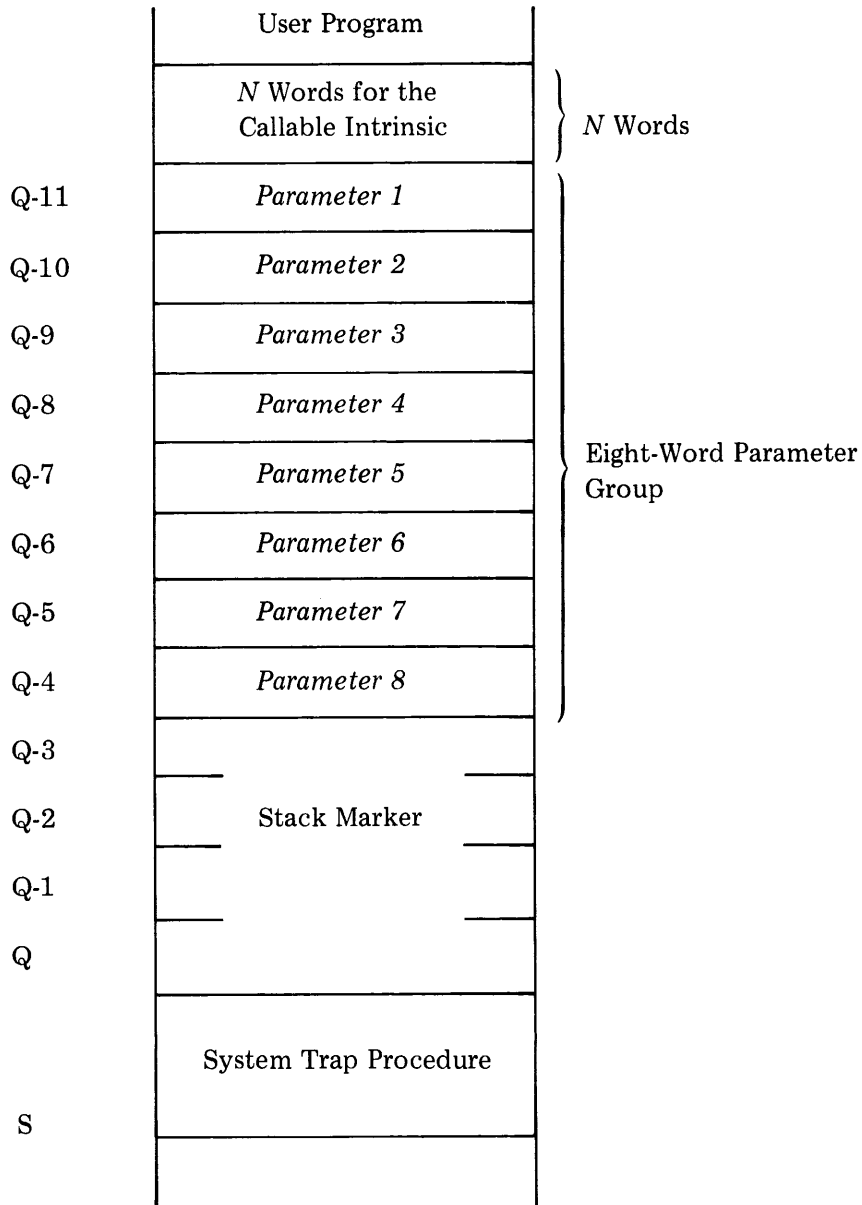
The format of the eight-word parameter group in Q-4 through Q-11 is

| 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

| | |
|---|---|
| I | Intrinsic number, as defined in Figure 10-4, Section X. |
| N | Number of callable intrinsic parameters. (To resume execution in the user code domain, an EXIT N+8 instruction should be executed.) |
| P | Additional parameter information. |
| P-1 through P-6 | Parameters modifying the error bytes, described below. If no modifying parameter is present, the corresponding parameter byte is set to zero. |
| E-1 through E-6 | Error bytes, containing the error codes noted in Section X. The last error code present is delimited by the value of zero in the following error byte. |

With these parameters, the trap procedure may take any recovery action necessary--write messages, produce selective dumps, set error-indication flags, or allow interactive debugging. Finally, the procedure may either call the TERMINATE intrinsic or issue an (EXIT N+8) instruction to return to the user's 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 one might issue an EXIT N+8 instruction follow:

PROCEDURE  SYSTEMTRAP  (PARAMETER1,PARAMETER2,PARAMETER3,
             PARAMETER4,PARAMETER5,PARAMETER6,
             PARAMETER7,PARAMETER8);

VALUE    PARAMETER1,PARAMETER2,PARAMETER3,PARAMETER4,
       PARAMETER5,PARAMETER6,PARAMETER7,PARAMETER8;

LOGICAL   PARAMETER1,PARAMETER2,PARAMETER3,PARAMETER4,
       PARAMETER5,PARAMETER6,PARAMETER7,PARAMETER8;

BEGIN

    INTEGER N;
      .
      .
      .
    << USER MAY OUTPUT MESSAGES >>
      .
      .
      .
    N:=PARAMETER1  LAND%37; <<N=NUMBER OF PARAMETERS
                  PASSED TO CALLABLE
                  INTRINSIC >>

    TOS:=N+%31410;     <<PUT "EXIT N+8" ON TOP
                  OF STACK >>

    ASSEMBLE (XEQ 0);    <<EXECUTE "EXIT N+8" ON
                  TOP OF STACK >>

END;

**CONTROL-Y Traps.**  A CONTROL-Y trap procedure assumes control when a CONTROL-Y entry interrupts the user's program.   An (EXIT N) instruction later returns control to the instruction in the user's code domain following the last instruction executed before the CONTROL-Y trap procedure was invoked.   When the trap procedure is invoked, the condition of the stack is as follows:
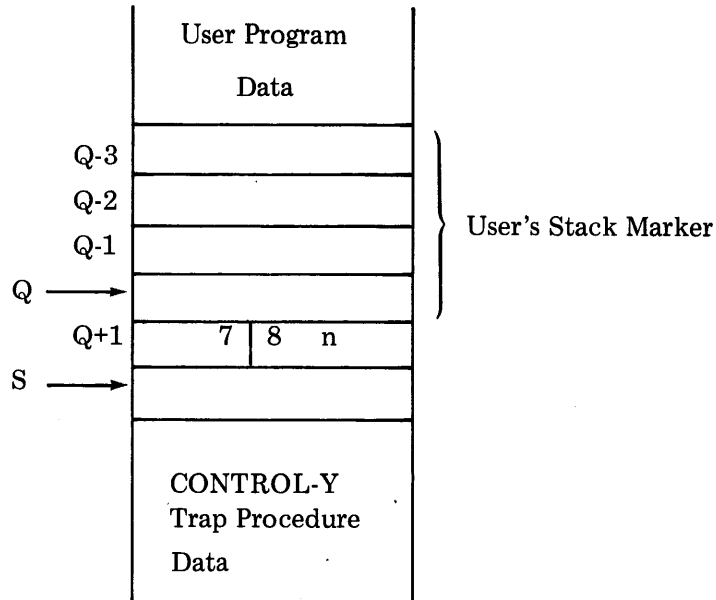


When the first instruction in the trap procedure is executed, the Q-Register points to the user's stack marker and the S-Register points to Q+2.   The trap procedure should not write data in the rightmost byte of the word Q+1, because it is used to exit back to the interrupted code. (The value $\underline{n}$ is the $\underline{N}$ parameter in the (EXIT N) instruction, which must be placed on the stack as follows.)

        TOS  :=%31400 + N

The instruction is then executed (by an XEQ instruction).

> *NOTE:*    *Users with the Privileged-Mode Optional Capability should be aware of the following:*
>
> 1.   *If the user's interrupted code was executing in privileged mode, his trap procedure must also be executed in privileged mode (and must thus have the privileged mode capability).*
>
> 2.   *When a user's 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's stack when a CONTROL-Y signal is received, it will continue to point to that extra data segment when the trap procedure is entered.*

A suitable declaration for a CONTROL-Y trap procedure is

        *PROCEDURE    CONTROLYTRAP;*

## CHANGING STACK SIZES

When a user prepares or executes a process, he specifies (or allows MPE/3000 to assign by default) the size of the stack (Z-DB) area and the user-managed (DL-DB) area within the stack segment. Once the process begins execution, the user can programmatically change the size of these areas to meet new requirements as they arise, by altering the register off-sets Z-DB or DL-DB. For example, the user typically expands the size of these areas when he finds, during process execution, that the sizes initially specified were not sufficient for his data requirements. Conversely, he might contract the size of either of these areas should his process no longer require large amounts of space for data. These changes are requested through the intrinsics described below.

If the user intends to dynamically expand or contract these areas, he *must* specify, at the time the stack is created, the anticipated maximum size of the stack segment; this value is used by MPE/3000 in allocating disc storage. The maximum stack size value can be specified in the *segsize* parameter of the :PREP, :PREPRUN, and :RUN commands, or in the *maxdata* parameter of the CREATE intrinsic (used by programmers with the *Process-Handling Optional Capability* to create processes).

> *NOTE:*  *When the stack segment belonging to a process running in privileged mode is frozen in main memory, either implicitly (when a user's process interfaces directly with the input/output system), or explicitly (by a direct call to appropriate uncallable system intrinsics), the intrinsics to change the register offsets DL-DB or Z-DB cannot be executed. When these intrinsics are called under such circumstances, a special FROZEN STACK error code is returned to the calling process, which may then attempt recovery. In general, this error code implies that the user should wait until the stack is unfrozen before re-issuing the intrinsic call.*

### Changing the DL-DB Area Size

The user can expand or contract the size of the current DL-DB area by altering the register off-set of the DL-address from the DB address (DL-DB). He does this by calling the DLSIZE intrinsic. This moves the current DB address and all following data forward (during expansion) or backward (during contraction) within the stack segment. All current information in the DL-DB area is saved. If the DL-DB size requested exceeds the maximum size permitted by the Z-DL (stack) area, only the maximum size permitted is granted. The intrinsic format is

*INTEGER PROCEDURE*    | *DLSIZE (size);* |

*VALUE size;*

*INTEGER size;*

*OPTION EXTERNAL;*

The size actually granted is returned to the calling process as the value of DLSIZE.

The *size* parameter specifies the new value of DL-DB. It is *an integer less than or equal to zero*. All increments or decrements to the DL-DB area are made in groups of 128 words. Thus, the size actually granted will be the *absolute value* of the size specified by the user, rounded upward accordingly.

The following condition codes may result:

CCE       Request granted.

CCG       The requested size exceeded the maximum limits allowed for the Z-DL area (Z-DL offset). This maximum limit is granted, and its value is returned to the calling process.

CCL       An illegal *size* parameter was specified; the original size assigned when the stack segment was created is granted. The CCL condition code also results when a FROZEN STACK exists, but, in addition, DLSIZE is replaced by a positive value as a special error indicator.

*EXAMPLE:*

*The user issues this intrinsic call to expand his DL-DB area to 300 words.*

**DLSIZE (-300);**

*Because DL-DB space is allocated in increments of 128 words, the amount of space actually granted is 384 words.*

## Changing the Z-DB Area Size

The user alters the size of the current Z-DB area by altering the register off-set of the Z address from the DB-address (Z-DB). He does this by calling the ZSIZE intrinsic. This moves the Z address forward (expansion) or backward (contraction). If the Z-DB area size requested exceeds the maximum size permitted for the Z-DL (stack area), only the maximum size allowed is granted. The format of the intrinsic is

*INTEGER PROCEDURE*       | *ZSIZE (size);* |

*VALUE size;*

*INTEGER size;*

*OPTION EXTERNAL;*

The size actually granted is returned to the calling process as the value of ZSIZE.

The size parameter specifies the new register offset (in words) for Z–DB. This must be an integer value greater than or equal to zero. All changes to the Z–DB area are made in increments or decrements of 128 words, and hence the size actually granted may differ from the size requested.

The following condition codes may be returned:

CCE     Request granted.

CCG     The requested size exceeded the maximum limits of the Z–DL (stack) area. This maximum limit is granted, and its value is returned to the calling process.

CCL     An illegal *size* parameter, less than (S–DB)+64 words, was specified. This minimum value is assigned by default. The CCL condition code also results when a FROZEN STACK exists, but, in addition, ZSIZE is replaced by a negative value as a special error indicator.


## REQUESTING A PROCESS BREAK

During a session, the user can interrupt an executing process, transferring control to the MPE/3000 Command Interpreter. This operation is called a *process break*. It allows the user to enter certain MPE/3000 commands, perhaps to create a file, transmit an informal message, or terminate a process. (The commands permitted during a break are designated in Appendix B.) The user initiates a break by pressing the *break* key on the terminal.


During a session, the user can also initiate a break programmatically, by entering the following intrinsic call (which requires no parameters). (This intrinsic is not valid in a job.)

*PROCEDURE*     | *CAUSEBREAK;* |

*OPTION EXTERNAL;*

A secondary entry-point to the CAUSEBREAK intrinsic, CAUSEBREAK', coincides with the main entry-point.

This intrinsic returns the following condition codes:

CCE         Request granted.

CCG         (Not returned by this intrinsic.)

CCL         Request not granted because the intrinsic was not called from a *session*.

The user can resume execution of a process at the point where it was interrupted, with this command:

*:RESUME*

## TERMINATING A PROCESS

The user can programmatically terminate or abort a process as follows.


### Termination

Process termination (the type occurring after successful execution of a process) is requested with the following intrinsic call:

PROCEDURE | TERMINATE;

OPTION EXTERNAL;

The process and all of its descendents (and any extra data segments belonging to them) are deleted. (Programmers using the CREATE intrinsic call to initiate the process can specify that the father of the terminating process be automatically activated.) All files still open by the process are closed and assigned the same disposition they possessed when opened. A secondary entry-point to the TERMINATE intrinsic, TERMINATE', coincides with the main entry-point.

### Abort

From within any process in a user process structure, the user can call an intrinsic that aborts this process. This intrinsic is the QUIT intrinsic, which also transmits an abort message to the calling process output device. This intrinsic is identical to TERMINATE, except that it is used for abnormal termination and consequently sets the job/session in an error state. (In batch jobs not containing the :CONTINUE command, this results in job termination when the entire program finishes.) The format of the QUIT intrinsic is

PROCEDURE | QUIT (num);

VALUE num;

INTEGER num;

OPTIONAL EXTERNAL;

In this call, *num* is an arbitrary user-specified number. When the QUIT intrinsic is executed, *num* is also output as part of the resulting abort message. The format of the abort message is described in Section X.

The user may abort the entire user-process structure (program) by calling the QUITPROG intrinsic. This destroys all processes up to, but not including, 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.) An abort message, as described in Section X, is transmitted to the job/session listing device. The QUITPROG intrinsic format is:

PROCEDURE        | QUITPROG (num); |

VALUE num;

INTEGER num;

OPTION EXTERNAL;

In this intrinsic, *num* is an aribtrary user-specified number. When the QUITPROG intrinsic is executed, *num* is output as part of the abort message.


## SETTING BREAKPOINTS AND DISPLAYING/MODIFYING STACK OR REGISTER DATA

The user can establish breakpoints (halts) in his program or display and modify data in his stack or in various registers by invoking the DEBUG intrinsic. This intrinsic is intended primarily for processes running in non-privileged mode. *Privileged programs can also invoke this intrinsic, but they should not do so while the DB-Register is not pointing to the stack.* The intrinsic format is:

PROCEDURE        | DEBUG; |

OPTION EXTERNAL;

### Invoking DEBUG

The DEBUG intrinsic can be invoked in two ways:

1.  By a direct call, in which case the following message is printed on the user's list device:

    *DEBUG segment.offset*

    (*segment* specifies the logical program segment being executed; *offset* is the current relative offset of the call to DEBUG within that segment.) (Logical program segments are discussed in Sections IV and VII; the *segment* parameter is actually the logical segment number of the desired segment, as reflected in the listing requested through the *PMAP* parameter of the :*PREP* or :*PREPRUN* command.)

2.  By reaching a previously-established breakpoint within the user's program, in which case the following message is printed on the user's list device:

    *BREAK segment.offset*

    (*segment* specifies the logical program segment being executed; *offset* is the relative offset of the breakpoint within that segment.

8-43

Following the DEBUG or BREAK message, the action taken depends upon whether the user is programming in *batch job* or *session* mode:

1. In *batch job* mode, the DEBUG procedure returns to the calling (or interrupted) process.

2. In *session* mode, the DEBUG procedure prompts the user for a DEBUG command by printing a question mark (?) as a prompt character. Such commands, discussed below, can be used to establish breakpoints, request displays, and change stack or register contents.

*NOTE:* *Although breakpoints are effective in programs running in either batch job or session mode, they can only be established in session mode because this is the only mode in which DEBUG commands can be entered.*

The condition code is not changed by the DEBUG intrinsic.

## DEBUG Command Format

A DEBUG command consists of:

- *A question mark* (used as a command identifier), printed automatically by DEBUG.
- *A command name* (mnemonic) consisting of a single letter.
- *A parameter list* (in most cases).

Any number of optional spaces can be embedded *anywhere* in the command, even between digits in a parameter list, to provide a free and flexible format.

The entire command is terminated by a carriage return.

*NOTE:* *Whenever a number appears as a parameter in a DEBUG command, it can be written as an* expression. *An expression is a group of octal values (terms) joined by plus or minus signs (operators) in the following format:*

*expression : := [+ | –] octaldigit. . . [ {+ | –} octaldigit. . .]. . .*

If a command results in an error, an error message is printed. The message consists of a four-letter *code* followed by a number (*nn*) indicating the position of the erroneous character in the command, in this format:

*codenn*

The following error codes are possible:

| Code | Meaning |
|------|---------|
| NONO | The user entered an illegal command — for example, a command containing a format error or requesting an illegal change to a register. |
| XTRA | Too much data was input by a command to modify memory (the M command.) |
| FULL | The system breakpoint table is full. |

## Setting Breakpoints

The user establishes a breakpoint within his program by entering the B command:

_?B [segment.] offset[,[segment.] offset] ...

segment          The logical program segment to contain the breakpoint. If omitted, the current segment applies.

offset           The relative offset (displacement from the start of the segment) of the breakpoint. (In this and other DEBUG commands, *offset* is an octal value.)

This command sets a breakpoint at each location specified by a *[segment.] offset* parameter.

When a breakpoint is reached during execution of the user's program, it is also cleared. But if , at that point, it is *immediately* respecified (through the B command), it will be reached again — with no instructions executed — when the user returns control to his program (through the R command, discussed below). Thus, to reach a breakpoint at the same place each time through a program loop, the user should alternate between two breakpoints.

> *NOTE:  Setting a breakpoint temporarily modifies the instruction at which the breakpoint occurs; thus, users are cautioned to ensure that this will not have any adverse side-effect upon their programs.*

*EXAMPLE:*

*The following B command establishes a breakpoint at Location 75 of Segment 3, Location 144 of Segment 2, and Location 76 of the current segment.*

_?B 3.75, 2.144, 76

## Clearing Breakpoints

A program breakpoint can be cleared by entering the C command:

_?C [segment.] offset[,[segment.] offset] ...

segment             The logical program segment containing the breakpoint. If omitted, the current segment applies.

offset              The relative offset of the breakpoint from the start of the segment.

When the parameter list is omitted, all breakpoints are cleared.


*EXAMPLES:*

*To clear the breakpoint at Location 25 of Segment 10, the user enters:*

_?C 10.25

*To clear all breakpoints in his program, the user enters:*

_?C


## Resuming Program Execution

To resume execution of the program (and optionally establish another breakpoint), the user enters the R command:

_?R [[segment.] offset]

segment             The logical program segment to contain the new breakpoint. If omitted, the current segment applies.

offset              The relative offset, from the beginning of the segment, of the new breakpoint.


The program always resumes execution at the instruction following the last breakpoint encountered.

If the optional parameter list is included in the R command, the new breakpoint is established before the user's program resumes execution. If the parameter list is omitted, execution resumes without establishment of a new breakpoint.

*EXAMPLE:*

*To resume program execution and run until Location 100 of the current segment is encountered, the user enters:*

?R 100

## Displaying Register Contents

To display the current contents of the Q, S, X, ST (status), P, Z, and/or DL registers, the user enters the D command in this format:

*D [(filereference)] [register]...*

| | |
|---|---|
| *filereference* | The name (and optional group and account) of the file on which the register display is to appear. If this parameter is included, DEV=LP is assumed. This file specification can be overridden by a :FILE command. If no *filereference* parameter is included, the session list device is used. |
| *register* | The register whose contents is to be displayed. This can be Q, S, X, ST, P, Z, or DL. If omitted, all registers are displayed. |

*EXAMPLE:*

*To display (on the session list device) all registers, the user enters:*

?D

*To display (on the session list device) the S and X registers, the user enters:*

?D S X

*To display (on the file named LP) the status and Z registers, the user enters:*

?D (LP) ST Z

## Displaying Stack Contents

To display the contents of particular areas of the user's stack, the *D* command is used in the following format:

*?D [(filereference)] stackaddr[,count]*

| | |
|---|---|
| *filereference* | The name (and optional group and account) of the file on which the stack display is to appear. If this parameter is included, DEV=LP is assumed. This file specification can be overridden by a :FILE command. If no *filereference* parameter is included, the session list device is used. |

*stackaddr*          The location (in the user's stack) at which the area to be displayed begins, relative to the contents of a particular register. It is written in this format:

$$\begin{bmatrix} Q \\ S \\ DL \end{bmatrix} \text{expression}[I[\text{expression}]]\ldots$$

Q, S, or DL specifies the register to which the *first* expression is relative; if none of these is specified, the DB register is assumed. The letter *I* indicates indirect addressing. The subparameter *expression* is an octal expression (as defined previously).

*count*          An *expression* (in the same format noted previously), showing the number of words to be displayed. If omitted, *count* is assigned a default value of 1.

The D command displays *count* words, starting with location *stackaddr.*

*EXAMPLES*

*Several D commands and their resultant displays are shown below:*

| D Command | Item Displayed |
|---|---|
| ?D 100 | *Location 100 (Relative to DB).* |
| ?D Q-5 | *Location Q-5. (If Q = 100, then this specifies Location 73.)* |
| ?D Q-5I | *Location pointed to by Location Q-5. (If Q = 100 and Location 73 contains 50, then this specifies Location 50.)* |
| ?D Q-5I, 100 | *100 words pointed to by Location Q-5.* |
| ?D (LP) Q-5I,100 | *100 words pointed to by Location Q-5, displayed on file LP.* |
| ?D Q-5II-1 + 3 | *Location specified by this double indirect address through Q-5 with an offset of 2. (If Q = 100, Location 73 contains 50, and Location 50 contains –30, then this specifies Location –26.)* |

## Modifying Register Contents

To modify the contents of the Q, S, X, ST, P, Z, and/or DL registers, the user enters the M command in this format:

_?M register=expression[,register=expression]...

register | The register whose contents are to be modified. This can be Q, S, X, ST, P, Z, or DL.

expression | An *expression* (as defined previously), specifying the new contents of the register. If the *register* parameter is *P*, then *expression* must be:

[[segment.]offset]

NOTE: *The following restrictions apply to the M command:*

1. *The register contents must be such that*

$$DL \leqslant 0 \leqslant Q \leqslant S \leqslant Z$$

2. *If the S-register is specified, it must be the last register in the parameter list.*

3. *Only Bits 2 through 7 of the ST register can be changed.*

EXAMPLE:

*Three M commands and their results are shown below:*

Command | Result

?M X = -5 | *Sets X Register to –5.*

?M Q = 100, S = 200 | *Sets Q Register to 100, and S Register to 200.*

?M P = 15.75 | *Sets P Register to Location 75 of Segment 15.*

## Modifying Stack Contents

To modify the contents of the user's stack, the user enters the M command in the following format:

?M stackaddr[,count] [=[expression] [,expression] ]...[.]

stackaddr | The location (in the stack) at which the area to be modified begins, written in the same format as *stackaddr* for the D command.

8-49

count                       An *expression* (in the same format noted previously for *expressions*)
                            specifying the number of words to be changed.  If omitted, *count* is
                            assigned a default value of 1.

*expression*                An *expression* (as previously defined) specifying the new contents
                            of the stack area.  When *expression* is omitted, the user is interactively
                            requested to supply new data as illustrated in the examples below.


*EXAMPLES:*

*To set Location Q + 3 in the user's stack to 100, the user enters:*

    ?M Q+3 = 100

*To set the three locations Q + 5 through Q + 7 to 300, 301, and 302 respectively, the user
enters:*

    ?M Q+5,3 = 300, 301, 302

*Suppose the user wants to modify some of 50 contiguous words of memory, pointed to by
the contents of Location Q + 1.  He would enter:*

    ?M Q11, 50

*The DEBUG intrinsic then would print the first location pointed to by Q + 1 (underscored
below), plus the contents of that location.  If Q + 1 pointed to Location 101, and
Location 101 contained 10, the output would show:*

                    ?M Q11, 50
▶ +000101 = 10,

*This output implicitly asks the user to supply new data for Location 101 and any following
locations, specified positionally.  Thus, if the user wanted to set locations 101, 102, 103,
and 105 to 1, 2, 3, and 5 respectively, he would respond as shown below.  Notice that
commas are used as delimiters between the data, and that an unchanged word (Location 104)
is denoted by two adjacent commas.*

                    ?M Q11, 50
▶ +000101 = 10,1,2,3,,5

*Since the user originally specified that he wanted to modify 50 words, he is prompted for
additional input (beginning with Location 106, present contents 0).  Suppose that the user
wanted to leave Locations 106 and 107 at their present setting, but set Location 110 to 10
and then terminate his input.  He would enter the following, using the period to denote
termination.*

                    ?M Q11, 50
   +000101 = 10,1,2,3,,5
▶ +000106 = 0 ,,,10.

**Requesting Trace of Stack Markers**

The user can request a trace of the current stack marker contents by entering:

<u>?</u>T

*EXAMPLE:*

*The T command below results in the following information:*

<u>?</u>T

    <u>10.100</u>    *(Current location.)*

    <u>5.175</u>    *(Location from which 10.100 was called.)*

    <u>0.65</u>    *(Location from which 5.175 was called.)*

    <u>0.3</u>    *(Location from which 0.65 was called. Since this is the last element listed, 0.3 should be in the outer block — if it is not, the stack markers have been destroyed.)*

## INTERPROCESS COMMUNICATION

The user can arrange for two processes belonging to the same job to communicate with each other through a *job control word.* This word is used primarily by systems programmers to enable a subsystem process to return information to the command executor that initiated that process. (Such a communication mechanism is used by the command executors for :RUN and various subsystem commands.) The general user may find this control word helpful in other applications.

Within the 16-bit control word, bit 0 is a sign bit used to notify the command executor (or other receiving process) of the normal termination (bit 0 = 0) or abort (bit 0 = 1) of the process. (This is the only HP convention governing the passing and interpretation of job control words between steps within a job.) Once bit 0 is set to 1, it can only be re-set by the actions of the :CONTINUE command. The remaining 15 bits can be used for whatever purpose desired, and can be set and re-set as necessary.

The following intrinsic call is used to set the bits in the job control word:

PROCEDURE     | *SETJCW (word);* |

*VALUE word;*

*LOGICAL word;*

*OPTION EXTERNAL;*

In this call, *word* is a word whose bit contents, established by the user, are placed in the job control word. Bit 0 in *word* cannot be set to 0.

The SETJCW intrinsic does not change the condition code.

The next intrinsic call returns the complete job control word to the calling process:

*LOGICAL PROCEDURE*     | *GETJCW;* |

*OPTION EXTERNAL;*

The job control word is returned as the value of GETJCW.

The GETJCW intrinsic does not change the condition code.


*EXAMPLE:*

*Suppose that a user is writing a job where several processes pass information to each other through the job control word. In one process, the user transmits the contents of the word PROCLNK to the job control word. By HP convention, Bit 0 of this word signifies the successful execution or abort of the process setting the job control word, but the meaning of the remaining 15 bits must be established by the user's conventions. Process A sets the job control word to PROCLNK, as follows:*

```
SETJCW (PROCLNK);
```

*When Process B is executed, it obtains this current job control word through the GETJCW intrinsic. In this case, the contents of the job control word are returned to the word STORELNK.*

```
STORELNK := GETJCW;
```

# VERIFYING DIAGNOSTIC DEVICE ASSIGNMENT

Users running diagnostic programs must frequently acquire sole access to one or more devices normally managed by MPE/3000 in order to perform field maintenance tests on those devices while interfacing with them at the start input/output (SIO) level. The user can verify the assignment of such devices programmatically through the CHECKDEV intrinsic:

*LOGICAL PROCEDURE*  $\boxed{\textit{CHECKDEV (ldev,hdwradr);}}$

*VALUE ldev;*

*INTEGER ldev,hdwradr;*

*OPTION EXTERNAL;*

This intrinsic returns to the user's process a word containing one of these logical values (as the value of CHECKDEV).

TRUE, (Bit 15 = 1) if all units referenced are assigned to diagnostic programs.

FALSE, (Bit 15 = 0) if one or more units referenced are not yet so assigned.

The parameters are

*ldev*　　　　　An integer indicating the logical unit number of the device to be isolated.

*hdwradr*　　　An integer indicating the following:

> 0, to return the DRT and unit numbers of the device specified by *ldev*, and check the diagnostic assignment of this unit. When the intrinsic is executed, Bits 0-7 of the word *hdwradr* will contain the DRT number of the device reference, and Bits 8-15 will contain its unit number.

> 1, to return the DRT number only of the device, and check the diagnostic assignment of *all* devices assigned this DRT number. When the intrinsic is executed, Bits 0-7 of the word *hdwradr* will contain the DRT number, and Bits 8-15 will be set to zero.

The following condition codes apply:

CCE　　　The requested logical unit is defined in MPE/3000.

CCG　　　The logical unit number specified is invalid.

CCL　　　(Not returned by this intrinsic.)

*EXAMPLE:*

*To check the diagnostic assignment of a device whose logical unit number is 23, and to return its DRT and unit numbers to the word INFOLNO, the user issues this call. If the unit requested is assigned, the value of the word ASGT will be TRUE.*

**ASGT := CHECKDEV (23,INFOLNO);**

## INTRINSICS FOR COMPILER WRITERS

Programmers writing compilers may need to programmatically request certain operations on USL files. For these programmers, the following utilities are provided.

### Initializing Buffers for USL Files

The user can initialize the first record (Record 0) of a USL file to the empty state by calling the INITUSLF intrinsic.

*INTEGER PROCEDURE*     | *INITUSLF (uslfnum,rec0);* |

*VALUE uslfnum;*

*INTEGER uslfnum;*

*INTEGER ARRAY rec0;*

*OPTION EXTERNAL;*

This intrinsic returns (as the value of INITUSLF) an error number (if an error occurs); if no error occurs, no value is assigned to INITUSLF.

The parameters are

*uslfnum*     A word identifier supplying the filenumber of the USL file.

*rec0*     A 128-word buffer to be initialized to the empty state, corresponding to the first record of the USL file (record 0).

The condition codes possible are

CCE     Request granted.

CCG     (Not returned by this intrinsic.)

CCL     Request denied; an error number is returned as the value of INITUSLF. (The error numbers possible are listed at the end of this sub-section.)

## Changing the Directory Block/Information Block Size on a USL File

The user can move the information block forward or backward on a USL file, thereby increasing or decreasing, respectively, the space available for the file directory block. (Notice that this does not change the overall length of the file.) This is done with the ADJUSTUSLF intrinsic.

*INTEGER PROCEDURE*          $\boxed{\textit{ADJUSTUSLF (uslfnum,records);}}$

*VALUE uslfnum,records;*

*INTEGER uslfnum,records;*

*OPTION EXTERNAL;*

This intrinsic returns (as the value of ADJUSTUSLF) an error number (if an error occurs); if no error occurs, no value is assigned to INITUSLF.

The parameters are

*uslfnum*        A word identifier supplying the filenumber of the USL file.

*records*        A signed record count. If *records* is greater than 0, the information block is moved forward in the USL file, increasing the space available for the directory block and decreasing that available for the information block. If *records* is less than 0, the information block is moved backward in the file, decreasing directory-block space and increasing information-block space.

The condition codes possible are

CCE        Request granted.

CCG        (Not returned by this intrinsic.)

CCL        Request denied; an error number is returned as the value of ADJUSTUSLF. (The error numbers possible are listed at the end of this sub-section.)


## Changing the Size of a USL File

The user can increase or decrease the length of a USL file by calling the EXPANDUSLF intrinsic:

*INTEGER PROCEDURE*          $\boxed{\textit{EXPANDUSLF (uslfnum,records);}}$

*VALUE uslfnum,records;*

*INTEGER uslfnum,records;*

*OPTION EXTERNAL;*

When this intrinsic is executed, a new USL file is created whose length is *records* longer (or shorter) than the USL specified by *uslfnum*. (The *records* parameter is a signed value; if it is positive, the new USL is longer than the old USL; if it is negative, the new USL is shorter than the old USL.) The old USL file is copied to the new file with the same file name, and the old file is then deleted.

This intrinsic returns (as the value of EXPANDUSLF) the new file number. If an error occurs during the execution of EXPANDUSLF, the intrinsic instead returns an error number as its value.

The parameters are

*uslfnum*       A word identifier supplying the filenumber of the USL file.

*records*       An integer specifying the number of records by which the length of the USL file is to be changed. If less than 0, the file is contracted; if greater than 0, the file is expanded.

The condition codes possible are

CCE       Request granted; the new file number is returned.

CCG       (Not returned by this intrinsic.)

CCL       Request not granted; an error number is returned. (The error numbers possible are discussed below.)


**USL File Intrinsic Error Numbers**

The INITUSLF, ADJUSTUSLF, and EXPANDUSLF intrinsics may return any of the following error numbers:

| Error Number | Meaning |
|---|---|
| 0 | Unexpected end-of-file was encountered. |
| 1 | Unexpected input/output error occurred. |
| 2 | An invalid *filecode* was specified. |
| 3 | An illegal file length was specified. |
| 4 | The user's request attempted to exceed the maximum file directory size (32,768 words). |
| 5 | Insufficient space was available in the USL file directory block. |
| 6 | Insufficient space was available in the USL file information block. |
| 7 | The utility was unable to open a new USL file. |
| 8 | The utility was unable to close (purge) an old USL file. |

| Error Number | Meaning |
|---|---|
| 9 | The utility was unable to close (purge) a new USL file. |
| 10 | The utility was unable to close $NEWPASS. |
| 11 | The utility was unable to close $OLDPASS. |

## CHANGING TERMINAL CHARACTERISTICS

Various commands and intrinsics can be issued against terminals to alter certain aspects of their operation. Before any of these intrinsics is issued against a terminal, the terminal/file must be opened with the FOPEN intrinsic.

On the HP 3000 Computer, terminals may be supported through either of the following controllers:

- Clock-Teletype Interface (Each controls one terminal.)

- Terminal Controller (Each controls up to 16 terminals.)

The Terminal Controller supports 103A and 202A modems, and hardwired terminals. The Clock-Teletype Interface supports only hardwired terminals.

### Types of Terminals

The following user terminals can be connected to an HP 3000 Computer running under MPE/3000:

- HP 2600A or DATAPOINT 3300 Keyboard — Display Terminal (10-240 cps). Connectable through clock-teletype interface.

- ASR-33 EIA-compatible (HP 2749B) Terminal (10 cps). Connectable through clock-teletype interface.

- ASR-35 EIA-compatible Terminal (10 cps). Connectable through clock-teletype interface.

- General Electric TermiNet 300 Data Communications Terminal, Model B (10/15/30 cps) with Paper Tape Reader/Punch, Option 2. Connectable through clock-teletype interface.

  *NOTE:    This terminal must be equipped for "ECHO PLEX."*

- Memorex 1240 Communication Terminal (10/15/30/60 cps). *Not* connectable through clock-teletype interface.

  *NOTE:* *This terminal must be equipped with the even parity checking option.*

- Execuport 300 Data Communications Transceiver Terminal (10/15/30 cps). *Not* connectable through clock-teletype interface.

- ASR-37 Teleprinter Terminal with Paper Tape Reader/Punch (15 cps). *Not* connectable through clock-teletype interface.

- IBM 2741 Communication Terminal (14.8 cps). (Call 360 Correspondence Code or PTTC/EBCD Code.) *Not* connectable through clock-teletype interface.

  *NOTE:* *This terminal must be equipped with the following features:*

  1. *Interrupt, Receive (IBM #4708) and Transmit (IBM #7900) associated with the terminal's ATTN key. (Terminals without Transmit (IBM #7900) are disconnected when log-on is attempted.)*

  2. *Dial-Up (IBM #3255) to enable system connection through a 103A modem or acoustic coupler.*

Terminals equipped with the automatic linefeed feature (operator selectable) must be operated with this feature OFF.

## Special Keys

The following keys have special significance to MPE/3000:

| KEY | MEANING (MOST TERMINALS) | EXCEPTIONS |
|---|---|---|
| $X^c$ | Deletes (ignores) the line being typed and then reads any following characters. The system responds with a triple exclamation point (!!!) followed by a carriage-return and line-feed. (The superscript $^c$ denotes a control character. Thus, "$X^c$" means "CONTROL-X.") | On IBM 2741, the user enters a two-character code followed by the character X. (Several ASCII characters and functions must be simulated by special two-character codes. These codes are described in Figure 8-1.) |
| $H^c$ | Deletes the previous character. (*Physical* backspacing does not occur.) The system echoes a reverse slash (\) unless in tape mode. (To delete $n$ characters, enter $n$ $H^c$'s.) | |
| $Q^c$ | Places terminal in tape mode; input is from paper tape containing $H^c$ or $X^c$'s. (Tape mode is not required to read tapes without these characters.) | |
| $Y^c$ | If the terminal is not in tape mode, $Y^c$ requests subsystem break (terminating program or command execution.) If the terminal is in tape mode, $Y^c$ returns it to the keyboard mode. | On an IBM 2741, a subsystem break is requested by the character sequence "*ATTN Y*" when not in input mode. (In input mode, this function is requested by "¢CY.") |
| *BREAK* | Requests a system break. | On an IBM 2741, a system break is requested by the sequence "*ATTN ATTN.*" |
| *ESC:* | Places the terminal in the echo-on mode so that characters input are echoed to the terminal printer by MPE/3000. | All terminals except the IBM 2741 are assumed to be in *echo-on* mode when the user logs on. |
| *ESC;* | Places the terminal in echo-off mode so that characters input are not echoed by MPE/3000. | |

The defined control characters $X^c$, $H^c$, $Q^c$, and $Y^c$ are recognized even when following an ESC-Key entry. However, entry of ESC followed by any character (other than one of these control characters or a colon or semi-colon) is read as a two-character string in the user's input stream.

## IBM 2741 Communication Terminal Interface

Because the IBM 2741 terminal generates non-ASCII code, special consideration must be given to the representation of several ASCII characters and functions which are not available in the 2741 character set.

For input from a 2741 terminal, these characters (and some of the functions) are simulated by entry of a two-character code. The first character of this code is the cent symbol (¢). The cent symbol is followed by one of several alphanumeric or special characters (shown in Figure 8-1) to compose a unique code representing one ASCII character or function.

On input from a 2741 terminal, the two-character code is translated into the internal ASCII code. On output to a 2741 terminal, ASCII code is translated into the appropriate two-character representation.

The IBM 2741 Communication Terminal must be equipped with the interrupt feature associated with the *ATTN* key. This key represents the *break* function; it is used to terminate program or command execution.

Any CALL/360 or PTTC/EBCD characters that do not have an equivalent ASCII character are ignored on input.

Figure 8-1 shows 2741 terminal representation of ASCII characters and functions.

## Changing Terminal Speed

MPE/3000 supports interactive terminals that run at speeds ranging from 10 to 240 characters per second (CPS). Once the user has logged on, he can change these speeds without logging-off by using the :SPEED command or the FCONTROL intrinsic (introduced for other functions in Section VI). This ability also allows a user running a mark-sense card-reader coupled to a terminal to operate the two devices at different speeds (for example, the card reader at 240 cps for input and the terminal at 10 cps for output). The :SPEED command and FCONTROL intrinsic are not valid for IBM 2741 Terminals, or other terminals that operate at only one speed, and do not apply to terminals connected through clock-teletype interfaces.

The :SPEED command can change both the input and output speed of the terminal. Its format is

$$\underline{:SPEED} \quad \left\{ \begin{matrix} [inspeed], \; outspeed \\ inspeed \end{matrix} \right\}$$

The following character sequences are defined to enable users of 2741 terminals to both input and output characters not available on the 2741 terminal.

| ASCII Terminal | | 2741 |
|---|---|---|
| <[1] | Less Than | ¢L |
| >[1] | Greater Than | ¢G |
| [ | Left Bracket | ¢( |
| \ | Reverse Slant | ¢/ |
| ] | Right Bracket | ¢) |
| ˆ | Circumflex | ¢A |
| ` | Grave Accent | ¢' |
| { | Opening Brace | ¢0 |
| \|[1] | Vertical Line | ¢V |
| } | Closing Brace | ¢S |
| ~ | Tilde | ¢T |
| *DEL* | Delete | ¢D |
| *ESC* | Escape | ¢E |
| *BREAK* | System break | *ATTN* |
| *CONTROL* | Control character | ¢C |
| —[2] | Underline | *UPSHIFT—* |

[1] Available on PTTC/EBCD terminal.

[2] Exists on some type balls.

**Figure 8-1. ASCII vs 2741 Character Representation**

*inspeed*    The new input speed desired.  (Required parameter if *outspeed* is omitted.)

*outspeed*    The new output speed desired.  (Optional parameter.)

Both of these values can be any of the following integers, specifying speed in characters per second.

   10, 14, 15, 30, 60, 120, 240.

If the *input* speed is changed, the following message is output (at the old speed):

   *CHANGE SPEED AND INPUT "MPE":*

In response, the user manually changes the speed control on the terminal and verifies the change by entering the message:

   *MPE*

If the characters "MPE" cannot be verified, the system assumes the terminal is to continue at the old speed.

Through the FCONTROL intrinsic, the user may change the terminal input or output speed programmatically.  The format of this intrinsic is

*PROCEDURE*    | *FCONTROL (filenum,controlcode,speed);* |

   *VALUE filenum,controlcode;*

   *INTEGER filenum,controlcode;*

   *LOGICAL speed;*

   *OPTION EXTERNAL;*

The FCONTROL parameters are

*filenum*    A word identifier supplying the filenumber of the terminal for which the speed is changed.

*controlcode*    The integer 10 (decimal) to change the input speed or the integer 11 to change the output speed.

*speed*    A reference parameter or word identifier that specifies the new speed desired:  10, 14, 15, 30, 60, 120, or 240 characters/second.  When the FCONTROL intrinsic is executed, the previous input or output speed is returned to the calling process through the *speed* parameter.

The condition codes are

    CCE      The request was granted.

    CCG      The request was not granted; the speed entered is not acceptable.

    CCL      The request was not granted; the process does not own the logical device, or this device is not a terminal.


*EXAMPLE:*

*To change the current input speed of the terminal identified by the filenumber stored in the word TERMFN from 60 to 120 characters per second, the user issues the following call. (The word SPD contains the value 120.)*

**FCONTROL (TERMFN, 10, SPD);**


*After the intrinsic is executed, the word SPD contains the integer 60 (the previous speed).*


**Changing Input Echo Facility**

Users can programmatically determine whether MPE/3000 transmits (echoes) input from the terminal keyboard back to the terminal printer, by calling the FCONTROL intrinsic to turn the echo facility on or off.

When the echo facility is *on*, input read from the terminal is echoed to the terminal's printer by MPE/3000. If the terminal is operating in full-duplex mode, the echoed information appears as normal printed lines. If the terminal is in half-duplex mode, however, the echoed printing is illegible — as the user enters input on such terminals, it is simultaneously printed by the terminal itself and subsequently overwritten by the echoed information. (Where a terminal can operate in either full- or half-duplex mode, the mode is selected by a switch on the terminal. When the user logs on, all terminals except 2741 terminals are assumed to have the echo facility *on*.)

When the echo facility is *off*, input read from the terminal is not echoed to the terminal's printer by MPE/3000. If the terminal is operating in full-duplex mode, no printing appears. If the terminal is in half-duplex mode, the input is copied by the terminal itself, and appears as normal, printed lines. (Bear in mind that the only way printing can be suppressed is with the echo facility *off* and the terminal in full-duplex mode, as illustrated in Figure 8-2.)

In addition to the FCONTROL intrinsic, the echo facility also can be switched on and off by entering the characters:

    *ESC:*   to turn echo facility *on*

    *ESC;*   to turn echo facility *off.*
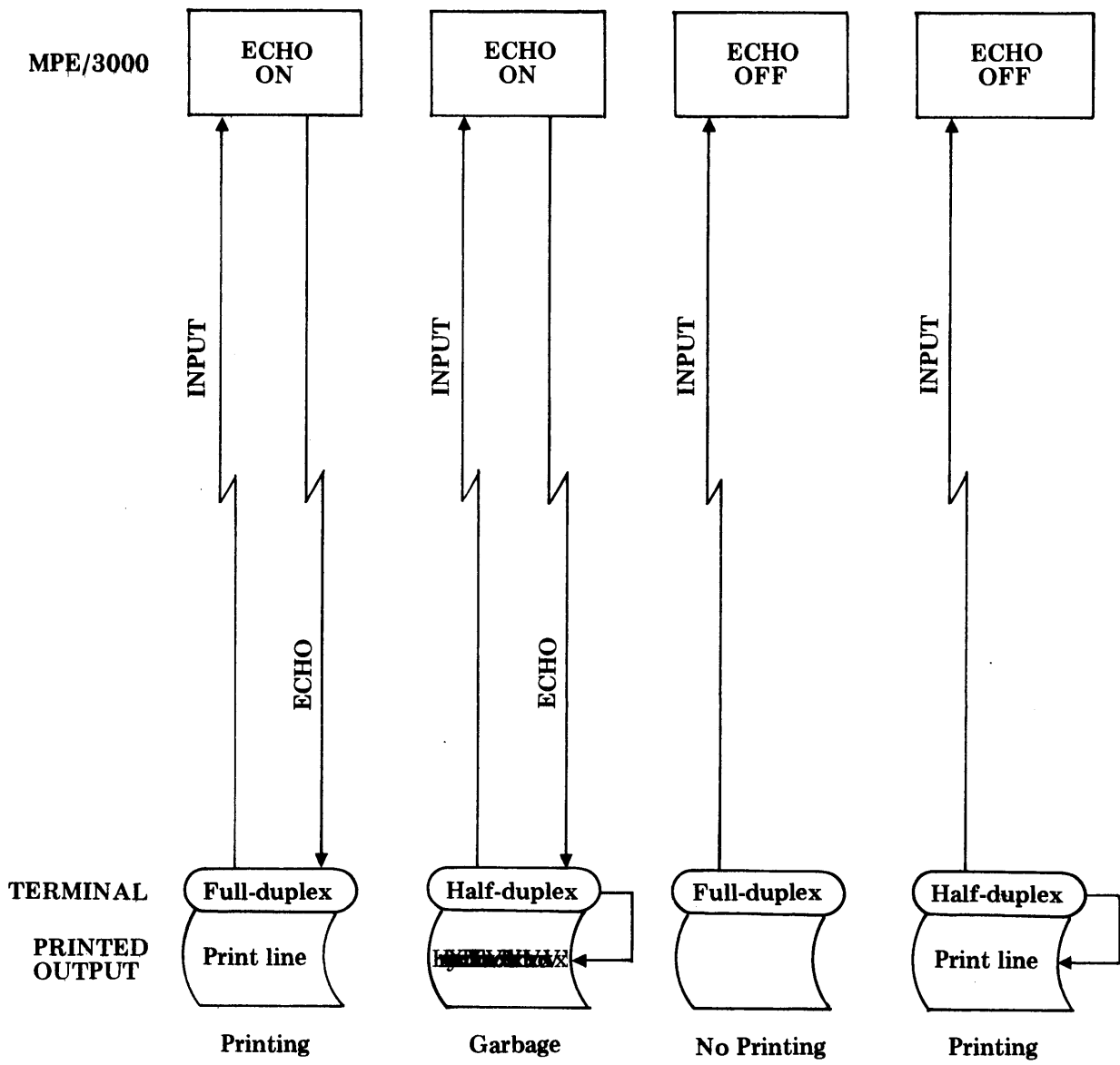
Figure 8-2. Echo Facility vs Duplex Mode

The format for this application of the FCONTROL intrinsic is

PROCEDURE      | FCONTROL *(filenum,controlcode,last);* |

*VALUE filenum,controlcode;*

*INTEGER filenum,controlcode;*

*LOGICAL last;*

*OPTION EXTERNAL;*

The FCONTROL parameters are

| *filenum* | A word identifier supplying the filenumber of the user's terminal. |
| *controlcode* | The integer 12 to turn the echo facility *on*, or the integer 13 to turn it *off*. |
| *last* | A reference parameter to which the *previous* echo facility is returned, where: |

0 = Echo on.

1 = Echo off.

The condition codes are

| CCE | The request was granted. |
| CCG | (Not returned by this intrinsic.) |
| CCL | The request was not granted; the file specified did not belong to this process or was not a terminal. |

## Enabling and Disabling System Break Function

The user can programmatically suspend or enable a terminal's ability to react to a system break request by calling the FCONTROL intrinsic. (System break requests are initialized by pressing the *break* key or by calling the CAUSEBREAK intrinsic.)

PROCEDURE    | *FCONTROL (filenum,controlcode,anyinfo);* |

*VALUE filenum,controlcode;*

*INTEGER filenum,controlcode;*

*LOGICAL anyinfo;*

*OPTION EXTERNAL;*

The FCONTROL parameters are

| | |
|---|---|
| *filenum* | A word identifier supplying the filenumber of the terminal for which the break function is enabled or disabled. |
| *controlcode* | The integer 15 to enable the break function, or the integer 14 to disable the break function. |
| *anyinfo* | Any variable or word identifier. This parameter is needed by FCONTROL to satisfy the internal requirements of this intrinsic. However, this parameter serves no other purpose and is not modified by the intrinsic. |

The condition codes are

| | |
|---|---|
| CCE | The request was granted. |
| CCG | (Not returned by this intrinsic.) |
| CCL | The request was not granted, because the file number specified does not denote a terminal or does not belong to the calling process. |

### Enabling and Disabling Subsystem Break Function

All terminals are initially set to disable (not accept) subsystem break requests, generated by pressing the *CONTROL-Y* (Y$^c$) key during a session. But the user can programmatically enable and again disable a terminal's ability to react to subsystem break requests by calling the FCONTROL intrinsic.

PROCEDURE    | *FCONTROL (filenum,controlcode,anyinfo);* |

*VALUE filenum,controlcode;*

*INTEGER filenum,controlcode;*

*LOGICAL anyinfo;*

*OPTION EXTERNAL;*

The FCONTROL parameters are

filenum              A word identifier supplying the filenumber of the terminal for which the
                     subsystem break function is enabled or disabled.

controlcode          The integer 17 to enable the escape function, or the integer 16 to disable
                     the subsystem break function.

anyinfo              Any variable or word identifier. This parameter is needed to satisfy
                     the internal requirements of the intrinsic. However, this parameter
                     serves no other purpose and is not modified by the intrinsic.

The condition codes are

    CCE     The request was granted.

    CCG     (Not returned by this intrinsic.)

    CCL     The request was not granted, because the file number specified does not
            denote a terminal or does not belong to the calling process.


## Enabling and Disabling Parity-Checking

All terminals and mark-sense card readers are initially set to disable parity-checking during
read operations. They may, however, be programmatically enabled for parity-checking by
calling the FCONTROL intrinsic. Then, the parity of the data received is checked against the
parity computed by the HP 3000 Asynchronous Channel multiplexor. If a parity error is
detected, an error code is made available to the user, who obtains it through the FCHECK
intrinsic. When a user is running a card reader coupled to a terminal, the ability to enable/
disable parity-checking allows him to obtain optimum utilization of the card reader by running
it at 240 characters/second without printing characters, while still outputting characters for
the terminal.

PROCEDURE        FCONTROL (filenum,controlcode,anyinfo);

VALUE filenum,controlcode;

INTEGER filenum,controlcode;

LOGICAL anyinfo;

OPTION EXTERNAL;

The FCONTROL parameters are

*filenum*  A word identifier supplying the filenumber of the terminal for which the parity check function is enabled or disabled.

*controlcode*  The integer 24 to enable parity-checking, or the integer 23 to disable parity-checking.

*anyinfo*  Any variable or word identifier. This parameter is needed to satisfy the internal requirements of the intrinsic. However, this parameter serves no other purpose and is not modified by the intrinsic.

The condition codes are

CCE  The request was granted.

CCG  (Not returned by this intrinsic.)

CCL  The request was not granted because the file number specified does not denote a terminal, or does not belong to the calling process.

## Enabling and Disabling Tape-Mode Option

The user can programmatically enable or disable the tape-mode option for a terminal. When enabled, the tape-mode option inhibits the implicit line-feed normally issued by MPE/3000 each time a carriage-return is entered. The tape-mode option also inhibits responses to $H^c$ and $X^c$ entries. (Thus, when $H^c$ is received and tape-mode is enabled, no reverse slash (\) is sent to the terminal; when $X^c$ is received and tape-mode is in effect, no exclamation points (!!!) are sent to the terminal.) The tape-mode option is enabled or disabled with the FCONTROL intrinsic.

*PROCEDURE*   FCONTROL (filenum,controlcode,anyinfo);

*VALUE filenum,controlcode;*

*INTEGER filenum,controlcode;*

*LOGICAL anyinfo;*

*OPTION EXTERNAL;*

The FCONTROL parameters are

*filenum*                  A word identifier supplying the filenumber of the terminal for which
                           the tape-mode is enabled or disabled.

*controlcode*              The integer 19 to enable tape-mode, or the integer 18 to disable
                           tape-mode.

*anyinfo*                  Any variable or word identifier. This parameter is needed to satisfy
                           the internal requirements of the intrinsic. However, this parameter
                           serves no other purpose and is not modified by the intrinsic.

The condition codes are

    CCE     The request was granted.

    CCG     (Not returned by this intrinsic.)

    CCL     The request was not granted because the filenumber specified does not
            denote a terminal, or the terminal does not belong to the calling process.


## Reading Paper Tapes Without X-OFF Control

The *X-OFF* control character (written by pressing the *X-OFF* key on a teletype) is used to
delimit data input on paper tape. When a TTY Tape Reader encounters this character while
reading a tape, reading halts until the program requests more input data.

If a user is working at a terminal with a paper tape reader that does not recognize the *X-OFF*
character, he can input data from paper tape by using the :PTAPE command. (This command
strips all *X-OFF* characters from the input read.) The input terminates when the $Y^c$
(*CONTROL-Y*) character is encountered, returning control to the user at the terminal
keyboard. The :PTAPE command is written as follows:

    *:PTAPE filename*

*filename*                 The name of an existing ASCII file on disc, to which the input data is
                           written. (This is normally a file with variable-length records; the record
                           size specified must be great enough to contain the longest paper tape
                           record.) (Required parameter.)

The user can programmatically read data from paper tapes not containing the *X-OFF* control
character, or from tapes input through terminals not recognizing this character, by calling the
PTAPE intrinsic. (The *X-OFF* characters are stripped from the tape.) Tape input terminates
when $Y^c$ is encountered, returning control to the user at the terminal keyboard. (Prior to
calling this intrinsic, the user must be sure to position the end-of-file pointer to the proper
position in the file. If he is inputting more than one tape, he should specify, in the FOPEN
call that opens the file, the *append-only* access type, and a *variable-length* record format,
before the first PTAPE call; furthermore, he should set the end-of-file pointer to zero, if
necessary, before issuing the first PTAPE call.)

The PTAPE intrinsic format is

*PROCEDURE*    | *PTAPE (filenum1,filenum2);* |

*VALUE filenum1,filenum2;*

*INTEGER filenum1,filenum2;*

*OPTION EXTERNAL;*

The intrinsic parameters are

*filenum1*    A word identifier supplying the filenumber of the user's terminal. (This is the value returned by FOPEN when this terminal/file is opened.).

*filenum2*    A word identifier supplying the filenumber of the disc file to which the data is to be written. (This is the value returned by FOPEN when this disc file is opened.)

The PTAPE intrinsic can return the following condition codes:

CCE    Request granted.

CCG    Request not granted because an error occurred while writing to the specified disc file.

CCL    Request not granted 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.

**Enabling and Disabling the Terminal Input Timer**

The terminal input timer records the time required to satisfy an input request on the terminal, from the time the input is requested, until it is completed. (This applies only to unbuffered, serial terminal input requests.) The user can programmatically enable or disable the terminal timer by calling the FCONTROL intrinsic.

*PROCEDURE*    | *FCONTROL (filenum,controlcode,anyinfo);* |

*VALUE filenum,controlcode;*

*INTEGER filenum,controlcode;*

*LOGICAL anyinfo;*

*OPTION EXTERNAL;*

The FCONTROL parameters are

*filenum*          A word identifier supplying the filenumber of the terminal for which the timer is enabled or disabled.

*controlcode*      The integer 21 to enable the timer, or the integer 20 to disable the timer.

*anyinfo*          Any variable or word identifier. This parameter is needed to satisfy the internal requirements of the intrinsic. However, this parameter serves no other purpose and is not modified by the intrinsic.

The condition codes are

    CCE      The request was granted.

    CCG      (Not returned by this intrinsic.)

    CCL      The request was not granted because the filenumber specified does not denote a terminal, or the terminal does not belong to the calling process.

## Reading the Terminal Input Timer

The user can read the result from the terminal input timer through the FCONTROL intrinsic. This result will be valid only if the terminal input was preceded by a call to enable the terminal timer. If valid, the result is the time (in hundredths of seconds) required for the last direct, unbuffered serial input on the terminal specified.

*PROCEDURE*      | *FCONTROL (filenum,controlcode,inputtime);* |

*VALUE filenum,controlcode;*

*INTEGER filenum,controlcode;*

*LOGICAL inputtime;*

*OPTION EXTERNAL;*

The FCONTROL parameters are

*filenum*          A word identifier supplying the filenumber of the terminal for which the timer is read.

controlcode          The integer 22.

inputtime            A word to which is returned the input time (in seconds/100).

The condition codes are

    CCE      The request was granted.

    CCG      The request was granted, but the result overflow 16-bits (*inputtime* was greater
             than 655.35 seconds).

    CCL      The request was not granted because the filenumber specified does not denote
             a terminal or the terminal does not belong to the user's process.


## Defining Line-Termination Characters For Terminal Input

Normally, the user at a terminal indicates the end of a line of input by entering a carriage
return (through the *return* key on most terminals).  He can, however, specify that a
different character (such as an equal sign, a period, or exclamation point) be recognized
as the standard line terminator during his process.  On subsequent read operations, the
input line will be terminated upon receipt of the specified character.  That character will
be returned to the requester's buffer.  No line-feed or carriage-return will be generated.

The line-terminator character change is requested through the FCONTROL intrinsic.

    *PROCEDURE*          | *FCONTROL  (filenum,controlcode,character);* |

    *VALUE*              *filenum,controlcode;*

    *INTEGER*            *filenum,controlcode;*

    *LOGICAL*            *character;*

    *OPTION EXTERNAL;*

The FCONTROL parameters are

    *filenum*            A word identifier supplying the filenumber of the terminal to
                         which the new terminating character applies.  (This parameter
                         *must* specify a terminal.)

    *controlcode*        The integer 25.

    *character*          A word identifier supplying (in the right byte) the character to
                         be used as a line terminator.  (The left byte of this word can
                         contain any information--it is ignored by the intrinsic.)

The condition codes are

CCE     The request was granted.

CCG     (Not returned by this intrinsic.)

CCL     The request was not granted; the character specified is disallowed.

The following characters are *not* allowed as line-terminating characters in the *character* parameter.

| ASCII Character | Octal Code |
|---|---|
| Control Y ($Y^c$) | 31 |
| Carriage-return | 15 |
| Line-feed | 12 |
| ESC | 33 |
| Control X ($X^c$) | 30 |
| Control H ($H^c$) | 10 |
| Control Q ($Q^c$) | 21 |
| X-OFF | 23 |

If the character supplied equals zero, the terminal will revert to normal line-control operation.

# SECTION IX
# Resource Management

Within MPE/3000, any element that can be accessed by a user's program is regarded as a *resource*. A resource thus can be an input/output device, file, program, subroutine, procedure, code segment, or the data stack. Occasionally, the user may want to manage a specific resource shared by a particular set of jobs or processes, so that no two of these jobs or processes can use the resource at the same time. To accomplish this type of resource management on either the inter-job (or session) or inter-process level, the jobs or processes involved must mutually cooperate. For example, if JOBB must not access a particular file when JOBA is using it, both jobs should include provisions for a hand-shaking arrangement overseen by MPE/3000 when these jobs are being executed concurrently. Under this arrangement, when JOBA has exclusive access to the file and JOBB attempts to access the same file, this access will be denied. JOBB will be suspended until JOBA releases its exclusive access. Then, JOBB can resume execution and access the file. (It is important to realize that as long as JOBB is suspended, it not only cannot access the file—it cannot perform *any* operations.)

On either the inter-job or inter-process level, the hand-shaking arrangement is based upon an arbitrary *resource identification number (RIN)* made available to users (at the inter-job level) or assigned to the job (at the inter-process level). Within their jobs (or processes), the cooperating programmers relate a RIN to a particular resource through the structure of the statements making up each job (or process). When a job (or process) seeks exclusive access to a resource, it requests MPE/3000 to lock the related RIN. (This request is granted only if no other job or process has already locked the RIN; otherwise, the requesting process is suspended until the RIN is released.) When it is finished with the resource, the job (or process) requests MPE/3000 to unlock the RIN so that other jobs (or processes) can lock it.

A RIN is *not* a *physical entity*. Furthermore, it is *not* logically *assigned* to any resource. The association between a RIN and a resource is accomplished only by the context of the statements within the job or process using the RIN. (This technique is illustrated later in this section.) The RIN is always known to MPE/3000 but its meaning (the resource with which it is associated) is not. For this reason, all cooperating programs must agree on what RIN is associated with what resource.

Processes run by users having only the *Standard MPE/3000 Capabilities* can lock only one RIN at a time. But processes run by users having the *Multiple RIN Optional Capability* discussed in Section XIII can lock more than one at a time. In doing so, however, the users must be careful to avoid deadlocking, where two suspended processes cannot be resumed because they are mutually blocked.

## INTER-JOB LEVEL

The RIN's used at the inter-job level are called *global RIN's*. They are used to exclude simultaneous access of a resource by two or more jobs. Each global RIN is a positive integer unique within MPE/3000. Global RIN's are acquired and released through commands, and locked and unlocked through intrinsics.

### Acquiring Global RIN's

Before any users can mutually engage in resource management through a RIN, one of these users must request the RIN and assign it a RIN password that enables all who know the password to lock the RIN. He does this by entering the :GETRIN command:

> *:GETRIN  rinpassword*

> *rinpassword*          A password that will be required in the intrinsic that locks the RIN. This is a string of up to eight alphameric characters beginning with a letter.  (Required parameter.)

The :GETRIN command is typically entered during a session. As a result of this command, MPE/3000 makes a RIN available for use, and displays the RIN number in this format:

> <u>*RIN: rin*</u>

> *rin*     The RIN number.

The user issuing :GETRIN can use this rin number to lock and unlock the RIN in the current session, or in future jobs and sessions. He also tells the RIN number (and password) to other users to permit them to lock and unlock the RIN in their jobs and sessions. All users pass the RIN number to the intrinsics that lock and unlock the RIN, as a reference parameter in the intrinsic calls. These users can continue using the RIN until the user who issued :GETRIN releases it. Thus, a user may use the same RIN for several days or weeks. (MPE/3000 regards the user issuing :GETRIN as the owner of the RIN assigned. This means that only this user may release the RIN.)

The total number of RINS that MPE/3000 can allocate is specified when the system is configured, and in no case can exceed 1024.

### Locking and Unlocking Global RIN's

Any global RIN assigned to a group of users can be locked (by one job at a time) by issuing the LOCKGLORIN intrinsic call. When this is done, any other jobs that attempt to lock this RIN are suspended. The requestor must know both the RIN number and password. Users with only *Standard MPE/3000 capabilities* cannot lock more than one global RIN simultaneously; an attempt to do so aborts the job. The intrinsic used to lock a global RIN is

| *PROCEDURE* | *LOCKGLORIN  (rinnum,lockcond,rinpassword) ;* |
|---|---|

*VALUE rinnum;*

*LOGICAL rinnum,lockcond;*

*BYTE ARRAY rinpassword;*

*OPTION EXTERNAL;*

The parameters are

| | | |
|---|---|---|
| *rinnum* | . | A word supplying the RIN number of the resource to be locked. (This is the RIN number furnished in the :GETRIN command.) |
| *lockcond* | | A word specifying conditional or unconditional RIN locking, as follows: |

TRUE    = Locking will take place unconditionally; if the RIN is not available, the calling process suspends until it is. (The TRUE condition is passed as a word whose 15th bit is *on*; all other bits are ignored.)

FALSE   = Locking will take place only if the RIN is immediately available; if it is not, control returns to the calling process immediately, with the condition code CCG. (The FALSE condition is passed as a word whose 15th bit is *off*; all other bits are ignored.)

*rinpassword*      A byte array containing the RIN password assigned through the :GETRIN command. In this array, the password must be terminated by an ASCII special character.

The condition codes possible if *lockcond* = TRUE are

| | |
|---|---|
| CCE | The request was granted. If the calling process had already locked the RIN, FALSE is returned to the word *lockcond*; if the RIN was free, TRUE is returned to *lockcond*. |
| CCG | (Not returned.) |
| CCL | (Not returned.) |

This condition codes possible if *lockcond* = FALSE are

| | |
|---|---|
| CCE | The request was granted. If the calling process had already locked the RIN, FALSE is returned to the word *lockcond*; if the RIN was free, TRUE is returned to *lockcond*. |
| CCG | The request was not granted because the RIN was locked by another job. |
| CCL | (Not returned.) |

The LOCKGLORIN intrinsic is aborted if the process already has another global RIN locked (and the user does not have the Multiple RIN Optional Capability); if an incorrect password is used; or if the specified RIN is not assigned to the job.

To unlock the same RIN, the user calls the following intrinsic:

*PROCEDURE*      | *UNLOCKGLORIN*     *(rinnum) ;* |

*VALUE rinnum;*

*LOGICAL rinnum;*

*OPTION EXTERNAL;*

In this call, the rinnum parameter must be a word supplying the number of the last RIN locked by the job, since only one RIN per job can be locked at one time. If *rinnum* does *not* specify this RIN, no action is taken. If no UNLOCKGLORIN intrinsic is issued for a locked RIN during a job, that RIN remains locked until :EOJ is encountered, then, it is unlocked.

The condition codes possible are

CCE      The request was granted.

CCG      The request was *not* granted because this RIN was not locked for this process.

CCL      The RIN was not allocated.

To illustrate how the above two intrinsic calls are used, consider two jobs run by two users who cooperate in managing a common disc file named RES3. Although both jobs may alter the contents of RES3, the users want to avoid the problems that would occur if both jobs attempted to write on the file simultaneously. Assuming that the users are utilizing RIN 42, allocated through a :GETRIN command, they might manage the file RES3 as illustrated. (RIN 42 is passed to the RIN-management intrinsics in the word RINX. The password associated with RIN 42 is stored in the array RINPASS. The 15th bit of the word TRU is *on.)*

When the first FWRITE intrinsic call in JOBA references the file RES3, no LOCKGLORIN intrinsic call has yet been issued by this job or by JOBB, running concurrently. Thus, this FWRITE call can access RES3. Subsequently, JOBA issues a LOCKGLORIN call that locks RES3 (and any other resources referenced between this LOCKGLORIN call and the subsequent UNLOCKGLORIN call). While RES3 is locked, *any* job can access it in the normal fashion, but *none* can lock it for exclusive use. Because the programmer who wrote JOBB does not want to access RES3 while this file is locked by another job, he specifies a lock call before he references this file in a FWRITE call. (In other words, he tries to lock the file for JOBB.) When the LOCKGLORIN call in JOBB is encountered and JOBA still has RES3 locked, JOBB is suspended. As soon as JOBA unlocks RES3, however, JOBB resumes execution and writes on this file.

*EXAMPLE:*

```
:JOB,JOBA,ACCT.JOE                          :JOB,JOBB,ACCT.JOHN
 •                                           •
 •                                           •
 •                                           •
                                            :SPL
                                             •
:SPL                                         •
 •                                           •
 •
 •
FWRITE (RES3,...);
 •
 •
 •
LOCKGLORIN (RINX,TRU,RINPASS);
 •
 •                                          LOCKGLORIN (RINX,TRU,RINPASS);
 •
                                               (JOBB IS SUSPENDED)

FWRITE (RES3,...);
 •
 •
 •

                                               (JOBB IS RESUMED)

UNLOCKGLORIN (RINX);
 •                                          FWRITE (RES3,...);
 •                                           •
 •                                           •
                                             •
:EOD
:EOJ
                                            UNLOCKGLORIN (RINX);
                                             •
                                             •
                                             •
                                            :EOD
                                            :EOJ
```

TIME

9-5

### Freeing Global RIN's

The owner of a RIN (the user who issued the :GETRIN command to acquire it) can de-allocate the RIN, returning it to the RIN pool managed by MPE/3000. Only the owner can de-allocate a RIN in this fashion. He does this through the :FREERIN command:

*:FREERIN rin*

*rin*  The number of the RIN to be de-allocated. (Required parameter.)

## INTER-PROCESS LEVEL

The RIN's used at the inter-process level are called *local RIN's*. They are used to exclude simultaneous access of a resource by two or more processes within the same job. Each is a positive integer that is significant only with respect to different processes within that job. Local RIN's are assigned, managed, and released through intrinsic calls.

### Acquiring Local RIN's

Just as global RIN's must be acquired by users before they can be used in jobs, local RIN's must be acquired by a job before they can be used by processes within the job. This is done with the following intrinsic call:

*PROCEDURE*  | *GETLOCRIN (rincount) ;* |

*VALUE rincount;*

*LOGICAL rincount;*

*OPTION EXTERNAL;*

In this intrinsic call, *rincount* is the number of local RIN's to be acquired by the job. The following condition codes can result from the GETLOCRIN call:

  CCE    The request was granted.

  CCL    Not enough RIN's are available to satisfy this call; none are allocated to the job.

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

## Locking and Unlocking Local RIN's

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

*PROCEDURE* | LOCKLOCRIN (rinnum,lockcond) ;

*VALUE rinnum;*

*LOGICAL rinnum, lockcond;*

*OPTION EXTERNAL;*

The parameters are

*rinnum*    One of the previously-allocated local RIN's, designated by an integer from 1 to the value specified in the *rincount* parameter of the GETLOCRIN call.

*lockcond*   A word specifying conditional or unconditional locking:

         TRUE  = Locking will take place unconditionally; if the RIN is not available, the calling process suspends until it is available. (The TRUE condition is passed as a word whose 15th bit is *on*; all other bits are ignored.)

         FALSE = Locking will take place only if the RIN is immediately available. If it is not, control returns to the calling process immediately with the condition code CCG. (The FALSE condition is passed as a word whose 15th bit is *off*; all other bits are ignored.)

The condition codes possible if *lockcond* = TRUE are

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

CCG   (Not returned.)

CCL   The request was not granted because the RIN was invalid.

The condition codes possible if *lockcond* = FALSE are

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

CCG   The request was not granted because the RIN was locked by another process.

CCL   The RIN was invalid.

To unlock this same RIN, the user issues the UNLOCKLOCRIN intrinsic call:

*PROCEDURE*       | *UNLOCKLOCRIN   (rinnum) ;* |

*VALUE rinnum;*

*LOGICAL rinnum;*

*OPTION EXTERNAL;*

This call makes the RIN indicated by *rinnum* available for locking by other processes in the job. The highest-priority process suspended because this RIN was locked is now activated.

The condition codes possible are

| CCE | The request was granted. |
| CCG | The request was not granted because the RIN specified is not locked by this process. |
| CCL | The request was not granted because the specified RIN is not allocated to the job. |

To illustrate how the LOCKLOCRIN and UNLOCKLOCRIN intrinsic calls are used, consider two processes (a father process and its son) within a job:

**FATHER PROCESS**

.
.
.

LP:=FOPEN (LIN, . . .);
.
.
.

GETLOCRIN (3);

FWRITE (LP, . . .);

LOCKLORIN (1, TRUEVAL);
CREATE (DESCEND, . . .).

FWRITE (LP, . . .);
.
.
.

UNLOCKLOCRIN (1);
.
.
.

**SON PROCESS**

.
.
.

LP:=FOPEN (LIN, . . .);
.
.
.

LOCKLORCIN (1, TRUEVAL);

FWRITE (LP, . . .);
.
.
.

FWRITE (LP, . . .);
.
.
.

UNLOCKLOCRIN (1);
.
.
.

Suppose that the father process and its son wanted to use RIN (1) to manage a line printer (designated by LP) so that the son process could not use the printer at any time that it was being used by the father process. This could be done as shown in the above coding. When the father process first references LP, the son process is not yet created and the printer need not be locked. However, just prior to creating the son, the father process locks the RIN covering the printer. The father issues all of its print requests before unlocking the printer. Before the son process accesses the printer, it tries to lock it, fails, and is suspended. As soon as the father unlocks the printer, the son process locks it, and issues print requests.

## Freeing Local RIN's

The user can free all RIN's currently reserved for his job by issuing this intrinsic call:

PROCEDURE | FREELOCRIN ;

OPTION EXTERNAL;

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.

The condition codes are

CCE       Request granted.

CCG       No RIN's are currently reserved for the job.

CCL       The request was not granted because at least one RIN is currently locked by a process.

## LOCKING AND UNLOCKING FILES

When an FOPEN intrinsic specifying the *dynamic locking aoption* is issued against a disc file, a RIN is established for that file. The user's process can call intrinsics that dynamically lock and unlock the file by alternately acquiring and releasing exclusive use of this RIN. When a file resides on a unit-record device, locking the file (RIN) is equivalent to locking the device itself.

Because the RIN's used in dynamic file locking are global RIN's, the user employing the file-locking intrinsics must follow the rules governing global RIN's. Specific capability-class rules governing file locking are:

1. *Standard Capabilities.* A user's running process (program) can lock only one file at a time.

2. *Process-Handling Optional Capability.* Within the job process structure, only one file can be locked at any one time.

3. *Multiple-RIN Optional Capability.* No restrictions are imposed.

If several processes of the same job are allowed access to a file in an exclusive mode, *local* (as opposed to global) RIN's are available without limitation.

Misuse of the file-locking intrinsics that violate the above rules can abort the job or session.

To dynamically lock a file, the user issues the FLOCK intrinsic call:

*PROCEDURE*        | *FLOCK (filenum,lockcond);* |

*VALUE filenum, lockcond;*

*INTEGER filenum;*

*LOGICAL lockcond;*

*OPTION EXTERNAL;*

The parameters are

*filenum*          A word supplying the filenumber of the file to be locked, through SPL/3000 conventions.

*lockcond*         A word specifying conditional or unconditional locking:

                 TRUE    = Locking will take place unconditionally; if the file cannot be locked immediately, the calling process suspends until it can. (Bit 15 = 1.)

                 FALSE   = Locking will take place only if the file's RIN is not currently locked; if the RIN is locked, control returns immediately to the calling process, with condition code CCG.(Bit 15 = 0.)

The condition codes possible if *lockcond* = TRUE are

     CCE       The request was granted.

     CCG       (Not returned.)

     CCL       The request was not granted because this file was not opened with the *dynamic locking aoption.*

The condition codes possible if *lockcond* = FALSE are

     CCE       The request was granted.

     CCG       The request was not granted because the file was locked by another process.

     CCL       The request was not granted because this file was not opened with the *dynamic locking aoption.*

*EXAMPLE:*

*To dynamically lock (unconditionally), the file whose filenumber is stored in the word FILEX the user enters the following call. (TRU is a word whose 15th bit is on.)*

**FLOCK (FILEX,TRU);**

To dynamically unlock a previously-locked file (RIN), the user issues the FUNLOCK intrinsic call:

*PROCEDURE*    | *FUNLOCK  (filenum) ;* |

*VALUE filenum;*

*INTEGER filenum;*

*OPTION EXTERNAL;*

The condition codes possible are

CCE     The request was granted.

CCG     The request was not granted because the file had not been locked by the calling process.

CCL     The request was not granted because the file was not opened with the *dynamic locking aoption.*

# SECTION X
# MPE/3000 Messages

Users running programs under MPE/3000 at any batch input device or terminal may encounter the following types of error messages:

*Command Interpreter Error Messages*, reporting fatal errors that occur during the interpretation or execution of an MPE/3000 command.

*Command Interpreter Warning Message*, reporting unusual conditions that occur during command interpretation or execution but that may not necessarily be deterimental to the processing of the user's job or session.

*Run-Time Messages*, denoting conditions that abort the user's running program (provided that an appropriate error trap has not been armed).

*User Messages*, which are messages sent to this user by other users currently running jobs or sessions.

*Operator Messages*, which are messages sent to this user by the console operator.

*System Messages*, that denote miscellaneous conditions that terminate or otherwise affect the user's job/session, such as an abort requested by the system supervisor or console operator.

These messages are all discussed in this section.

Other messages may be received only at the MPE/3000 Console. These are:

*Console Operator Messages*, including:

- Status Messages that indicate the current status of jobs/sessions or input/output devices.

- Input/Output Messages that request service for, and report errors on, input/output devices.

- User Messages, sent by users to the console operator.

*System Failure Messages*, including:

- System Halt Messages, reporting the aborting of an uncallable MPE/3000 intrinsic.

- Cold Load Errors, reporting errors encountered while cold-loading the system.

These console operator and system failure messages are discussed in *HP 3000 Multiprogramming Executive Console Operator's Guide (03000-90006)*.

## COMMAND INTERPRETER ERROR MESSAGES

When MPE/3000 detects an error during interpretation or execution of an MPE/3000 command, it suppresses execution of that command and prints an error message on the job/session listing device, in the format shown below.

*ERR errnum [,detail] [message]*

errnum
A number, ranging from 0 to 250, identifying one of the errors described in Figure 10-1. These error numbers are grouped as follows:

|  |  |
|---|---|
| 0-19: | General errors not related to specific command parameters. |
| 20-47: | Errors relating to command parameter syntax. |
| 48-99: | Errors relating to specific commands. |
| 100-199: | Errors encountered by the File Management System, or within the file directory. |
| 200-249: | Errors encountered by the CREATE intrinsic, or by the MPE/3000 Loader |
| 250: | Error encountered by the MPE/3000 Segmenter. |

detail
A number that (unless otherwise stated in Figure 10-1) refers to the erroneous parameter element in the command. When counting parameter elements, count from the left, beginning with the element immediately following the command name counted as 1; consider every delimiter, including equal signs and omitted items.

message
A message describing the error, also shown in Figure 10-1. In batch jobs, this message appears automatically; in sessions, it may be requested optionally by entering any character other than a carriage return following

*ERR errnum [,detail]*

When an error occurs while the user is trying to initiate a job or session, the error number is printed but no *message* appears (jobs) or can be requested (sessions).

When an error occurs in a batch job (and no :CONTINUE command precedes the erroneous command, as noted in Section III), all information between the erroneous command and the end-of-job is ignored, and the job is aborted. When an error occurs during an interactive session, control returns to the user, who may then request the *message* display (as described above) or may enter a carriage return to request prompting for a new MPE/3000 command.

| ERRNUM | MESSAGE |
|--------|---------|
| 0 | JOB NOT YET DEFINED<br><br>The first command entered was not :JOB, :HELLO, or :DATA, required to initialize an input stream. Enter such a command. |
| 1 | UNKNOWN COMMAND<br><br>The command entered was not recognized as a legal MPE/3000 command. Enter a correct command. |
| 2 | ABNORMAL PROGRAM TERMINATION<br><br>The user's program terminated with an error condition (the high-order bit of the job control word set). (Calling the QUIT or QUITPROG intrinsic, setting Bit 0 of the job control word to 1, issuing an :ABORT command during a break, or killing a process does this; HP 3000 subsystems set this state when detecting serious errors.) Debug and re-run program. |
| 3 | WRONG (JOB/SESSION) MODE<br><br>This command is not permitted in the defined job or session mode.<br>Enter a correct command. |
| 4 | INSUFFICIENT CAPABILITY<br><br>The user does not have the capability required to execute this command or one of its particular options. Check user's capability against command requirements. |
| 5 | TOO MANY PARAMETERS<br><br>The command image contained too many parameters or exceeded 255 characters. Check the parameter list and re-enter the command correctly. |
| 6 | INSUFFICIENT PARAMETERS<br><br>Required parameters were omitted from the command parameter list. Check the parameter list and re-enter the command correctly. |
| 7 | MISSING COLON<br><br>This command, encountered in a batch job, did not contain a colon as the first character of the command image. (Continuation lines must also have such colons.) Correct the command and re-run the job. |

Figure 10-1. Command Interpreter Error Messages

| ERRNUM | MESSAGE |
|--------|---------|
| 8 | **ILLEGITIMATE ACCESS** <br><br> Access to MPE/3000 was denied because one of the following conditions occurred: <br><br> 1. The user did not specify a log-on group and also had no home group. Specify a log-on group when re-entering the log-on command. <br><br> 2. The account, user, specific group, or home group does not exist. See system manager, supervisor, or operator. <br><br> 3. A password is required by the account, user, or group (other than home group), and the user either supplied no password or supplied an incorrect one. Check to ensure that the correct password is supplied; if this fails, contact the system manager or account manager user. |
| 9 | **UNACCEPTABLE DEVICE** <br><br> A :HELLO or :JOB command was entered on a device not currently configured to accept jobs/sessions; a :DATA command was entered on a device that does not presently accept data; or :HELLO was entered on a non-interactive terminal. Initialize input stream on an appropriate device. |
| 10 | **INSUFFICIENT RESOURCES** <br><br> The operation requested was rejected because sufficient system resources (data segments, code segment table entries, process control blocks, and so forth) were not available for it. Request operation when it is more likely that such resources will be available, or see system supervisor user. |
| 11 | **COMMAND NOT YET IMPLEMENTED** <br><br> This command entered is not legal in this version of MPE/3000. |
| 12 | **NOT ALLOWED PROGRAMMATICALLY** <br><br> This command cannot be entered with the COMMAND intrinsic. |
| 13 | **VDD FULL** <br><br> A :DATA command was entered, but the virtual device directory was full and could not accomodate more :DATA files. Re-enter command later, or see system supervisor user. |
| 14 | **JOB OVERLOAD** <br><br> A request for job scheduling was not accepted because the Job Master Table is full. Request scheduling later, or see system supervisor user. |
| 15 | **SUBSYSTEM NOT FOUND** <br><br> An MPE/3000 subsystem was invoked, but does not exist in the permanent file directory. See system manager user. |

**Figure 10-1. Command Interpreter Error Messages (Continued)**

| ERRNUM | MESSAGE |
|--------|---------|
| 16 | **DISC I/O ERROR**<br><br>A disc input/output error occurred during processing of a command. Depending on the nature of the error, the command may succeed when re-issued. |
| 20 | **SYNTAX ERROR**<br><br>A delimiter in the parameter list is not permitted in the command, or is not permitted at the point where it was detected. When a qualifying number is shown (*detail*), the bad delimiter is adjacent to the indicated parameter element (usually following it). Re-enter the command correctly. |
| 21 | **PARAMETER NOT OPTIONAL**<br><br>A parameter was omitted in a position where the parameter is required. Re-enter the command correctly. |
| 22 | **ILLEGAL PARAMETER**<br><br>A parameter was not recognized as legal. Check the parameter list, and re-enter the command correctly. |
| 23 | **PARAMETER OUT OF BOUNDS**<br><br>The value specified for the indicated parameter was not within the allowable range. Refer to the command description for the permissible range, and re-enter the command correctly. |
| 24 | **ILLEGAL PARAMETER IN THIS CONTEXT**<br><br>An otherwise legitimate parameter is not permitted in the specified format of the command, or it conflicts with some previous specification in the command. Check the command format, and re-enter the command correctly. |
| 25 | **DUPLICATE PARAMETER**<br><br>The command was rejected because the same parameter appeared twice in the parameter list; re-enter the command without the duplicate parameter. |
| 26 | **ILLEGAL KEYWORD**<br><br>A keyword in a command was not recognized as legal. Check the keyword, and re-enter the command correctly. |
| 27 | **DUPLICATE KEYWORD**<br><br>The command was rejected because the same keyword appeared twice in the parameter list; re-enter the command without the duplicate keyword. |
| 28 | **ILLEGAL KEYWORD IN THIS CONTEXT**<br><br>An otherwise legitimate keyword is not permitted in the specified format of the command, or it conflicts with some previous specification in the command. Check the command format, and re-enter the command correctly. |

**Figure 10-1. Command Interpreter Error Messages (Continued)**

10-5

| ERRNUM | MESSAGE |
|---|---|
| 29 | ILLEGAL NAME<br><br>A syntactically-invalid name was included in the command; most names are limited to eight alphanumeric characters or less, beginning with a letter. Check the format of the name. |
| 30 | INVALID NUMBER<br><br>A syntactically-invalid number appeared in the command. This could be caused by a non-numeric character (other than a leading "+", "=", or "%"); an "8" or "9" for an octal number; or a number that is too large. Check the parameter list, and re-enter the command correctly. |
| 48 | TERMINAL/TYPE INCOMPATIBILITY<br><br>For a :HELLO or :JOB command, the *TERM=termtype* parameter is not legitimate for the terminal being used. Check the terminal type versus the *termtype* specified. |
| 49 | BAD OUTCLASS<br><br>For a :JOB command, the *OUTCLASS=outclass* parameter is not a legitimate device specification, or the device (device class) does not support serial output. Check the device requested against the *outclass* parameter. |
| 50 | MINIMUM CAPABILITIES OMITTED<br><br>One of the following violations occurred:<br><br>1. SM was not included for the SYS account.<br><br>2. Neither IA nor BA was included.<br><br>3. A system manager or account manager attempted to cancel his own manager capability.<br><br>Check the caplist parameter, and re-enter the command properly. |
| 51 | FILESPACE LIMIT BELOW CURRENT COUNT<br><br>A filespace limit was specified that was less than the current amount allocated. Re-specify the limit correctly. |
| 52 | CAPABILITY EXCEEDS ACCOUNT'S<br><br>For an account manager command to create or alter a group or user, the capability specified for group, user, or local attributes is not a subset of the account's capability or local attributes. Determine account's capability and respecify user's capability within these limits. |
| 53 | MAXPRI EXCEEDS ACCOUNT'S<br><br>For an account manager command to create or alter a user, specification of the user's maximum priority is higher than the account's maximum priority. Determine account's maximum priority, and respecify user's within these limits. |

**Figure 10-1. Command Interpreter Error Messages (Continued)**

| ERRNUM | MESSAGE |
|--------|---------|
| 54 | **LIMIT(S) EXCEEDS ACCOUNT'S**<br><br>Filespace, central processor time, and/or connect limit greater than the corresponding limits for the account were specified. Determine these values for the account, and re-specify user's limits accordingly. |
| 55 | **ILLEGAL FILE NAME**<br><br>In a command reference for a file, an illegal file name appeared. The syntax for file names is:<br><br>    filename [/lockword] [.gname[.aname] ]<br><br>Each name or lockword consists of eight characters or less, beginning with a letter. Check this syntax against the file name specified, and re-enter the command. |
| 56 | **FILE EQUATION TABLE FULL**<br><br>The last :FILE command was rejected because no more space existed in the file equation table. The :RESET command, however, can be used to provide space for additional entries by cancelling existing :FILE commands. |
| 57 | **BACK FILE REFERENCE NOT FOUND**<br><br>A previous :FILE command for this job/session was referenced, in the *formaldesignator* format, but could not be found. Check :FILE name. |
| 58 | **TOO MANY BACK FILE REFERENCES**<br><br>A :FILE command had more than 155 following :FILE commands referring to it via the *formaldesignator* back-file reference construct. |
| 59 | **INVALID FILE DESIGNATOR**<br><br>The formal file designator in a :FILE or :RESET command was syntactically incorrect. Re-enter the command with correct syntax. |
| 61 | **DEVICE OF WRONG TYPE**<br><br>A device of the wrong type was referenced in a command—for example, output was directed to a card reader or the :STORE command was applied to a non-tape file. Re-specify, with correct device. |
| 62 | **DUPLICATE ACCESS SPECIFIED**<br><br>A *modelist:userlist* pair appears twice in the same :ALTSEC command (or system manager command) parameter list. Eliminate this redundancy and re-enter command. |
| 63 | **ILLEGAL LIBRARY SPECIFIED**<br><br>In the :PREPRUN and :RUN command, the LIB=library parameter was not specified correctly—only S (system), P (public), and G (group) are permitted entries for this parameter. Re-enter command with proper *library* requested. |

Figure 10-1. Command Interpreter Error Messages (Continued)

| ERRNUM | MESSAGE |
|--------|---------|
| 64 | UNABLE TO ACCESS TERMINAL FILE<br><br>The user's job/session could not open $STDIN.<br>See system supervisor, |
| 66 | ILLEGAL DEVICE OR INVALID INPUT SPEED<br><br>For the :SPEED command, an illegal *inspeed* parameter was specified for this device. Check this parameter against characteristics of device used. |
| 67 | ILLEGAL DEVICE OR INVALID OUTPUT SPEED<br><br>For the :SPEED command, an illegal *outspeed* parameter was specified for this device. Check this parameter against the characteristics of the device used. |
| 68 | SUBQUEUE IS LINEAR AND HAS NO QUANTUM<br><br>This command attempted to set a time-quantum for a linear subqueue, but quanta can only be changed for circular subqueues. |
| 69 | UNKNOWN SUBQUEUE<br><br>The subqueue referenced by this command could not be identified by the system. Check names of subqueues available. |
| 70 | RIN CURRENTLY IN USE<br><br>The :FREERIN command did not succeed because the RIN requested was in use. Wait until the RIN no longer is in use, and re-enter command. |
| 71 | RIN NOT ALLOCATED TO THIS USER<br><br>The :FREERIN command attempted to de-allocate a RIN not belonging to the user; only the owner of the RIN can free it. |
| 72 | RIN TABLE FULL<br><br>The :GETRIN command was issued, but no RIN's are presently available. Re-enter command when a RIN is available or see system supervisor. |
| 73 | DIRECTORY ERROR<br><br>A miscellaneous error was detected in accessing the file directory; re-enter the command. |
| 74 | INVALID LIST FILE<br><br>An improper list file was specified in the command—re-specify, naming another file. |
| 75 | UNDEFINED JOB NAME<br><br>The specified job does not exist, or is in an improper mode for the command (for example, when the :TELL command is applied to a job not in execution). |
| 76 | MESSAGE ROUTING ERROR<br><br>An internal routing error occurred; *detail* indicates the type of error. Inform system supervisor. |

Figure 10-1. Command Interpreter Error Messages (Continued)

| ERRNUM | MESSAGE |
|--------|---------|
| 77 | TOO MANY FILESETS<br><br>Too many *filesets* were specified in a :RESTORE command. The :RESTORE command requires the following limits for specifying *filesets*: 10 by account name; and 15 by account and group name; and 20 by account, group, and file name. (More than one :RESTORE can be applied against the same tape, using fewer filesets than the limit in each command.) |
| 78 | IMPROPER TAPE FORMAT<br><br>A tape being restored (:RESTORE command) has a valid STORE/RESTORE label, but the information on the tape does not conform to the format of a tape written by the :STORE or :SYSDUMP commands. Ensure that a proper tape is used. |
| 79 | NON-EXISTENT USER<br><br>The command specified a user not defined in the system. Check to ensure that the spelling of the user name, as supposedly recorded in the system, is correct. If it is and the user still cannot be referenced, see the system manager user. |
| 100 | FILE SYSTEM ERROR<br><br>An error was returned from the File System; *detail* shows the actual File System Error Number, interpreted by referring to Error Codes 20 through 110 in Figure 10-5B. |
| 101 | UNIT NOT READY<br><br>The command specified an input/output unit that was not ready. Request the operator to ready the unit. |
| 102 | NO WRITE RING<br><br>The command specified, as an output device, a tape unit with no write-enable ring on the tape. Request the operator to insert the ring. |
| 103 | INCONSISTENT FILE OPERATION<br><br>The command was inconsistent with the access type, record type, or device type for the referenced file. Check for proper consistency. |
| 104 | PRIVILEGED FILE VIOLATION<br><br>The command attempted unauthorized access to a privileged file. |
| 105 | INSUFFICIENT DISC SPACE<br><br>The command requested more disc space than that available. If possible, resubmit command with less space requested or purge existent files to make available additional space. |
| 106 | NON-EXISTENT ACCOUNT<br><br>The command referenced an account not defined in the system. See system manager user. |

Figure 10-1. Command Interpreter Error Messages (Continued)

| ERRNUM | MESSAGE |
| --- | --- |
| 107 | NON-EXISTENT GROUP<br><br>The command referenced a group not defined in the system. See account manager user. |
| 108 | NON-EXISTENT FILE<br><br>The command referenced a file not defined in the job temporary or system file domain. Ensure that the file name was entered correctly. |
| 109 | INVALID FILE NAME<br><br>The command requested a file with an invalid name. Check the file name spelling and syntax. |
| 110 | DEVICE UNAVAILABLE<br><br>The command referenced a device that is not available. Request availability from the operator, or re-specify the device. |
| 111 | INVALID DEVICE SPECIFICATION<br><br>The command requested a device with an invalid device specification. Check the specification used, and re-enter command. |
| 112 | NO PASSED FILE<br><br>The command required a passed file, but no file was passed. |
| 113 | EXCLUSIVE VIOLATION<br><br>The command required exclusive access to a file already being accessed, or required access to a file to which another process had exclusive access. |
| 114 | LOCKWORD VIOLATION<br><br>The command requested a file protected by a lockword, but no lockword (or an incorrect lockword) was supplied. |
| 115 | SECURITY VIOLATION<br><br>The command requested a file protected against this access by the MPE/3000 File Security System. |
| 116 | DUPLICATE NAME<br><br>The attempted creation of an account, group, user, permanent file or temporary file encountered a duplicate name. For files, this could be a duplicate name in the system or job temporary file directory. |
| 117 | DIRECTORY OVERFLOW<br><br>The command caused the job temporary file directory or system directory to overflow. |

**Figure 10-1. Command Interpreter Error Messages (Continued)**

| ERRNUM | MESSAGE |
|---|---|
| 118 | ATTEMPT TO SAVE SYSTEM FILE AS JOB TEMPORARY<br><br>The command attempted to save a system file in the job temporary file directory. Re-enter the command correctly. |
| 119 | IN USE: CAN'T BE PURGED<br><br>The command attempted to purge a file, group, user, or account that was in use. Wait until the item referenced is free, and re-enter the command. |
| 120 | CREATOR CONFLICT<br><br>The :RENAME, :SECURE, :RELEASE, or :ALTSEC command was entered for a file by someone other than the creator of the file; use of these commands is restricted to creator of the file. |
| 121 | GROUP FILESPACE EXCEEDED<br><br>The maximum disc space allocatable for the group's files was exceeded. |
| 122 | ACCOUNT FILESPACE EXCEEDED<br><br>The maximum disc space allocatable for the account's files was exceeded. |
| 200 | CREATE ERROR<br><br>An error has been detected by the CREATE intrinsic; *detail* indicates the actual CREATE intrinsic error number, which can be interpreted by reference to Message Block D in Figure 10-5D. |
| 201 | LOAD ERROR<br><br>An error has been detected by the LOAD intrinsic; detail shows more information about the actual error, and can be interpreted by reference to Message Block C in Figure 10-5C. |
| 202 | ILLEGAL LIBRARY SEARCH<br><br>The command requested an illegal library search. |
| 203 | UNKNOWN ENTRY POINT<br><br>The command specified a program entry point that could not be located. |
| 204 | DATA SEGMENT TOO LARGE<br><br>The command required a data segment such that either:<br><br>1. The data segment exceeded the system-defined maximum size allowed; or<br><br>2. The data segment required, or the *maxdata* parameter specified, exceeds 32767 words. |
| 206 | DATA SEGMENT LARGER THAN MAXDATA SPECIFICATION<br><br>The command required a data segment larger than the maximum size specified by the user in this case. |

Figure 10-1. Command Interpreter Error Messages (Continued)

| ERRNUM | MESSAGE |
|---|---|
| 207 | ILLEGAL NUMBER OF CODE SEGMENTS<br><br>The command required more than 152 code segments or more than the maximum allowed by the system. |
| 208 | INVALID PROGRAM FILE<br><br>The command referenced a program file whose filecode or format is incorrect. |
| 209 | CODE SEGMENT TOO LARGE<br><br>The command required a code segment larger than the maximum size permitted by the system. |
| 210 | MORE THAN ONE EXTENT IN PROGRAM<br><br>The command referenced a program containing more than one disc extent. |
| 214 | SYSTEM SL ACCESS ERROR<br><br>A system segmented library access error occurred; *detail* specifies a File System Error Code shown in Figure 10-5B (Error Codes 20 through 110). |
| 215 | PUBLIC SL ACCESS ERROR<br><br>A public segmented library access error occurred; *detail* specifies a File System Error Code shown in Figure 10-5B (Error Codes 20 through 110). |
| 216 | GROUP SL ACCESS ERROR<br><br>A group segmented library access error occurred; *detail* specifies a File System Error Code shown in Figure 10-5B (Codes 20 through 110). |
| 217 | PROGRAM FILE ACCESS ERROR<br><br>A program file access error occurred; *detail* specifies a File System Error Code shown in Figure 10-5B (Codes 20 through 110). |
| 218 | LIST FILE ACCESS ERROR<br><br>A list file access error occurred; *detail* specifies a File System Error Code shown in Figure 10-5B (Codes 20 through 110). |
| 219 | INVALID SYSTEM SL FILE<br><br>The command referenced the system segmented library file, whose filecode or format is incorrect. |
| 220 | INVALID PUBLIC SL FILE<br><br>The command referenced a public segmented library file whose filecode or format was incorrect. |

Figure 10-1. Command Interpreter Error Messages (Continued)

| ERRNUM | MESSAGE |
|---|---|
| 221 | INVALID GROUP SL FILE<br><br>The command referenced a group segmented library file whose filecode or format was incorrect. |
| 222 | INVALID LIST FILE<br><br>The command referenced an invalid list file. |
| 223 | ILLEGAL PROGRAM DEALLOCATION<br><br>The command illegally attempted to deallocate a program. |
| 224 | ILLEGAL PROGRAM ALLOCATION<br><br>The command illegally attempted to allocate a program. |
| 225 | ILLEGAL PROCEDURE DEALLOCATION<br><br>The command illegally attempted to deallocate a procedure. |
| 226 | ILLEGAL PROCEDURE ALLOCATION<br><br>The command illegally attempted to allocate a procedure. |
| 227 | ILLEGAL CAPABILITY<br><br>The command referenced a file or program such that: permanent program file capability exceeds its groups; temporary file capability exceeds user's; or program requires privileged mode but program's capability does not include it. |
| 228 | UNABLE TO OBTAIN CST ENTRIES<br><br>The command required code segment table (CST) entries that could not be obtained. |
| 229 | UNABLE TO OBTAIN PROCESS DST ENTRY<br><br>The command required data segment table (DST) entries that could not be obtained. |
| 230 | UNABLE TO OBTAIN VIRTUAL MEMORY<br><br>The command required virtual memory space that could not be obtained. |
| 231 | TRACE SUBSYSTEM NOT PRESENT<br><br>The command requested loading of a program to be traced, but the TRACE/3000 subsystem is not in the system segmented library as required. |
| 232 | PROGRAM LOADED IN OPPOSITE MODE<br><br>The command requested that a program be simultaneously loaded in NORMAL and NORPRIV mode. |

Figure 10-1. Command Interpreter Error Messages (Continued)

| ERRNUM | MESSAGE |
|--------|---------|
| 250 | SEGMENTER ERROR<br><br>A Segmenter error was detected in attempting to prepare a program; *detail* may show additional information, as follows:<br><br>3 = Unable to create Segmenter process.<br><br>4 = Unable to access Segmenter process.<br><br>5 = Unable to send mail to Segmenter process.<br><br>6 = Unable to receive mail from Segmenter process.<br><br>7 = Segmenter process aborted. |

**Figure 10-1. Command Interpreter Error Messages (Continued)**

## COMMAND INTERPRETER WARNING MESSAGES

When certain unusual but generally non-fatal conditions arise during interpretation or execution of an MPE/3000 command, various warning messages may appear on the job/session list device. Most such messages do not terminate batch jobs nor interrupt sessions; a job/session generally continues immediately following the warning message. The message itself appears simply as:

*message*

The Command Interpreter Warning Messages are summarized in Figure 10-2.

> *NOTE:* In addition to the messages listed in Figure 10-2, other command Interpreter Warning Messages are also output. Figure 10-2 shows only those messages that require explanation; the other messages are self-explanatory.

CAPABILITY EXCEEDS ACCOUNT'S

The user's capability is greater than his account's capability. He is given the maximum capability possible—
the intersection of the two sets—for the job/session.

COMMAND IGNORED—NOT ALLOWED IN BREAK

This command cannot be executed during a break, without aborting the job.

DEFAULT VALUES TAKEN

Some default values were taken during loading the user's program (during execution of commands such
as :RUN, :SPLGO, and :PREPRUN, for example). These conditions are described under the discussion
of the CREATE intrinsic (CCG condition code) in Section XI.

EXECUTION PRIORITY REQUESTED EXCEEDS CAPABILITY

The scheduling priority requested at log-on time (through the ;PRI=*priority* parameter of the :HELLO or
:JOB command, or its default value (CS)), is greater than the user's maximum priority as defined by the
account manager user. The user is given the highest-priority standard subqueue below the BS subqueue
that satisfies the user's maximum priority restriction.

INVALID RESPONSE

The user gave an invalid response to a query from MPE/3000 requiring only a YES or NO response; the
user is given a second chance to respond.

MAXPRI EXCEEDS ACCOUNT'S

The user's maximum priority value (as defined by the account manager) is greater than that of the
account. The user's job/session is given, as its maximum priority, the smaller of the two values.

MESSAGE TOO LONG

The *message* parameter for the :TELL or :TELLOP command is too long to permit the destination mes-
sage (including its prefix) to fit within a single 72-character line. (The total length is a function of the
prefix, which in turn is a function of the fully-qualified job name of the sender.) The message is not sent.

PREMATURE JOB TERMINATION

An error during a command in a batch job (without a preceding :CONTINUE command) caused the job
to fail without further command interpretation.

READ PENDING

A *break* request was initiated while a read was pending for the user's program. The read must be satisfied
following entry of the :RESUME command. All characters entered for a partial record before the *break*
are retained.

USER NOT ACCEPTING MESSAGES

The job to receive a user's message, though defined, is not in EXECUTION state.

Figure 10-2. Command Interpreter Warning Messages

# RUN-TIME MESSAGES

A user's program can be aborted as a result of any of the following general types of run-time errors:

| Type | Description |
|------|-------------|
| 1 | Special violations—those detected through the internal interrupt structure (such as arithmetic trap errors, parity errors, bounds violations, and so forth) and other violations detected by MPE/3000 (such as stack overflows or invalid stack markers). |
| 2 | Explicit calls to the QUIT intrinsic (Section VIII). |
| 3 | Explicit calls to the QUITROG intrinsic (Section VIII). |
| 4 | Violations of other callable intrinsics, such as passing of illegal parameters or invoking of an intrinsic without having the required capability class or a valid register environment. |

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/3000 terminates the user's process and transmits a *run-time (abort) message* to the user's output device. In a multi-process structure, Error Type 2 aborts only the violating process but Error Types 1, 3, and 4 abort the entire program.

If the aborted program was running in a batch job, the job is removed from the system (if no :CONTINUE command overrides termination).

If it was running in a session, control of the session is returned to the user 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 (is indicating 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.*

## Type 1 Error Messages

For Type 1 Errors, the format for a run-time (abort) error message is:

$$ABORT{:}pname.segment.location{:}sname.segment.location{:}message$$

$$\underbrace{\hspace{4cm}}_{p\text{-field}} \quad \underbrace{\hspace{4cm}}_{s\text{-field}} \quad \underbrace{\hspace{2cm}}_{m\text{-field}}$$

The *p-field* indicates the location of the last instruction executed in the user program prior to the abort.

The *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.

The *m-field* contains the error message text, as described below.

Within each field, the parameters are:

| | |
|---|---|
| *pname* | The name of the program file containing the user's program, and optionally, the group and account name. |
| | In the special case of a process having been PROCREATED from a segment in a segmented library (SL) (for example, the Command Interpreter), an asterisk (*) is output followed by the SL name in symbolic form (sname, below). |
| *sname* | The symbolic name of the SL in which the segment exists |

            SYSL  -    System SL

            PUSL  -    Public SL

            GRSL  -    Group SL

| | |
|---|---|
| *segment* | The logical number of the code segment with respect to either the program or SL, whichever is appropriate. |
| *location* | The location in the code segment. This is expressed in terms of the displacement (P-PB), where P is the absolute address of the instruction and PB the absolute address of the base of the code segment. |
| *message* | Any of the *messages* defined in Figure 10-3. |

> *NOTE:*   *Octal numbers are indicated by a percent sign (%) preceding the number.*
>
>            *If the stack is completely destroyed and no valid stack markers can be found that define a user environment, then the above-defined subfields will be output containing a question mark (?).*

*EXAMPLE*

*Suppose that a user's process attempted an integer divide operation, using zero as the divisor. (The process is running the program in the file PROGX, and the current instruction is at Location 73 of Segment 32.) The process terminates, and the following message is output.*

        `ABORT:PROGX.%32.%73:INTEGER DIV ZERO`

| Message | Description |
|---|---|
| INTEGER OVERFLOW | Integer overflow. |
| FL. PT. OVERFLOW | Floating point overflow. |
| FL. PT. UNDERFLOW | Floating point underflow. |
| INTEGER DIV ZERO | Integer divide by zero attempted. |
| FL. PT. DIV ZERO | Floating point divide by zero attempted. |
| PRIVILEGED INSTRUCTION | Execution of privileged instruction when in non-privileged mode attempted. |
| ILLEGAL INSTRUCTION | Execution of an undefined instruction attempted. |
| ADDRESS VIOLATION | Illegal address referenced. |
| BOUNDS VIOLATION | Operation attempted outside legal bounds. |
| NON-RESP MODULE | Non-responding module. |
| DATA PARITY | Data parity error. |
| MEMORY PARITY | Memory parity error. |
| SYSTEM PARITY | System parity error. |
| STACK UNDERFLOW | Stack underflow. |
| CST VIOLATION | Code Segment Table (CST) bounds violation. |
| STT VIOLATION | Segment Transfer Table (STT) bounds violation. |
| STT UNCALLABLE | STT entry uncallable. |
| STACK OVERFLOW | Stack segment unable to be expanded upon stack overflow. |
| PROGRAM KILLED | Program aborted from an external source. |
| INVALID STACK MARKER | Stack marker contains invalid information, when interrogated by MPE/3000. |

Figure 10-3. Type 1 Error Messages

## Type 2 Error Messages

For Type 2 Errors, the format for a run-time error message is:

$$ABORT\underbrace{:pname.segment.location}_{p\text{-}field}\underbrace{:sname.segment.location}_{s\text{-}field}\underbrace{:PROCESS\ QUIT\ P=xxx}_{m\text{-}field}$$

The contents of the p- and s-fields are the same as noted under the discussion of Type 1 Error Messages. However, the m-field always contains the notation *PROCESS QUIT P=xxx*, where *xxx* is the value of any *num* parameter passed to the QUIT intrinsic (Section VIII) by the terminating process. (This value is output only if it is *not* zero.)

*EXAMPLE*

   *Suppose that the user's process (running the program in the file PROGB) called the QUIT intrinsic (with num=%10) at Location 23 of Segment 70. The following message would be output:*

       ABORT:PROGB.%70.%23:PROCESS QUIT P=%10

10-18

## Type 3 Error Messages

For Type 3 Errors, the format for a run-time error message is:

$$ABORT\text{:}\underbrace{pname.segment.location}\text{:}\underbrace{sname.segment.location}\text{:}\underbrace{PROGRAM\ QUIT\ P=xxx}$$

$$\qquad\qquad p\text{-field}\qquad\qquad\qquad s\text{-field}\qquad\qquad\qquad m\text{-field}$$

The contents of the p- and s-field are the same as noted under the discussions of Type 1 and 2 Error Messages. But, the m-field always contains the notation *PROGRAM QUIT P=xxx*, where *xxx* is the value of any *num* parameter passed to the QUITPROG intrinsic by the terminating process. (This value is only output if it is not zero.)

## Type 4 Errors Messages

In certain cases, the reporting of errors encountered during the execution of callable intrinsics (Type 4 Errors) may involve some complexity. For example, one callable intrinsic might detect an error actually reported by a second intrinsic that is invoked; the error detected by the second intrinsic might, in turn, have been reported by a third intrinsic invoked by the second. Consider, for instance, a case where the user invokes the CREATE intrinsic to create a process; CREATE, in turn, invokes the LOAD intrinsic to load the process, and LOAD invokes the FOPEN intrinsic to open the program file. Then, CREATE might signal that the process could not be created because of a LOAD error, which in fact, occurred because FOPEN could not open the file. To facilitate the reporting of such hierarchical errors, Type 4 Error Messages have the following format:

$$ABORT\text{:}\underbrace{pname.segment.location}\text{:}\underbrace{sname.segment.location}\text{:}\underbrace{I=intrinsicnumber\ M=e1(p1).e2(p2)...en(pn)}$$

$$\qquad\qquad p\text{-field}\qquad\qquad\qquad s\text{-field}\qquad\qquad\qquad\qquad m\text{-field}$$

The contents of the p- and s-fields are the same as noted under the discussions of Type 1, 2, and 3 Error Messages. However, the m-field is defined as follows:

| | |
|---|---|
| *intrinsicnumber* | A number that identifies the callable intrinsic being executed when the abort error occurred. (A list of intrinsic numbers versus intrinsics is presented in Figure 10-4). |
| *ei* | An error number (*i* is always less than or equal to 6) which is an index into a *message block* (collection of messages) shown in Figure 10-5A through 10-5H. |
| *pi* | An optional parameter used to modify the message corresponding to the error number *ei*. (In *pi*, *i* is also always less than or equal to 6). (If present, *pi* denotes the intrinsic parameter in error.) |

To understand how to interpret a Type 4 Error run-time message, consider the following message structure.

*ABORT:pname.segment.location. . .:I = xxx M = e1.e2.e3*

The message is interpreted from left to right. First, the intrinsic number (*xxx*) and a corresponding message block letter are obtained from Figure 10-4; call this *b1*.

> *NOTE:*     *Message Block A (Figure 10-5A) is a special block of messages that is accessed whenever the error number (index) is less than 20, regardless of which message block is indicated. It is concatenated at the low-index range to all message blocks.*

The first message corresponds to applying the error number e1 as an index in message block b1 in Figure 10-5. Associated with this message is (optionally) the next message block letter; call this b2. The second message, which is used to further explain the first message, is obtained from message block b2 using index e2, and so forth, until no more error numbers are available. Thus, the message actually corresponds to three messages which, when read together, describe the reason for the abort.

*EXAMPLE*

*Suppose, when a user is attempting a CREATE/LOAD/FOPEN sequence, his process is aborted and the following message is printed on his listing device:*

```
ABORT:ABCD.PUB.SYS.%1.%2054:I=100 M=30.52.51
```

The user would interpret the message as follows:

1.  The name ABCD is the name of the program file. The names PUB and SYS denote the group and account, respectively, to which this file belongs.

2.  The number 1 denotes that the error occurred in Logical Code Segment number 1 in the segmented program.

3.  The number 2054 is the octal (P-PB) relative address where the error occurred in the above code segment.

4.  The number 100 is the number of the intrinsic in Figure 10-4 (100 = CREATE) whose execution was aborted. The block index letter D associated with this intrinsic, directs the user to Block D in Figure 10-5D.

5.  The first message parameter (e1=30) points to Error Number 30 under Block D. This indicates a LOAD error, and also directs the user to an additional modifying message in Block C in Figure 10-5C (through another block index number).

6.  The second message parameter (e2=52) points to Error Number 52 under Block C. This indicates that the file could not be opened due to a violation in the group segmented library (SL), and also directs the user to still another modifying message in Block B.

7.  The third error-number parameter (e3=51) indicates that the reason the file could not be opened was that the group referenced did not exist. This completes the error analysis.

| INTRINSIC NO. | MESSAGE BLOCK LETTER (IN ADDITION TO A) | INTRINSIC NAME |
|---|---|---|
| 1 | B | FOPEN |
| 2 | B | FREAD |
| 3 | B | FWRITE |
| 4 | B | FUPDATE |
| 5 | B | FSPACE |
| 6 | B | FPOINT |
| 7 | B | FREADDIR |
| 8 | B | FWRITEDIR |
| 9 | B | FCLOSE |
| 10 | B | FCHECK |
| 11 | B | FGETINFO |
| 12 | B | FREADSEEK |
| 13 | B | FCONTROL |
| 14 | B | FSETMODE |
| 15 | B | FLOCK |
| 16 | B | FUNLOCK |
| 17 | B | FRENAME |
| 18 | B | FRELATE |
| 19 | B | FREADLABEL |
| 20 | B | FWRITELABEL |
| | | |
| 30 | | GETLOCRIN |
| 31 | | FREELOCRIN |
| 32 | | LOCKLOCRIN |
| 33 | | UNLOCKLOCRIN |
| 34 | H | LOCKGLORIN |
| 35 | | UNLOCKGLORIN |
| | | |
| 40 | | TIMER |
| 41 | | CHRONOS |
| 42 | | PROCTIME |
| | | |
| 50 | | XARITRAP |
| 51 | | ARITRAP |
| 52 | | XLIBTRAP |
| 53 | | XSYSTRAP |
| 54 | | XCONTRAP |

Figure 10-4.  Intrinsic Numbers vs. Intrinsics

| INTRINSIC NO. | MESSAGE BLOCK LETTER (IN ADDITION TO A) | INTRINSIC NAME |
|---|---|---|
| 55 | | RESETCONTROL |
| 56 | | CAUSEBREAK |
| 60 | | TERMINATE |
| 62 | | BINARY |
| 63 | | ASCII |
| 64 | | READ |
| 65 | | PRINT |
| 66 | | PRINTOP |
| 68 | | COMMAND |
| 69 | | WHO |
| 70 | | SEARCH |
| 71 | G | MYCOMMAND |
| 72 | | SETJCW |
| 73 | | GETJCW |
| 74 | | DBINARY |
| 75 | | DASCII |
| 76 | | QUIT |
| 80 | C | LOADPROC |
| 81 | C | UNLOADPROC |
| 82 | | INITUSLF |
| 83 | | ADJUSTUSLF |
| 84 | | EXPANDUSLF |
| 100 | D | CREATE |
| 102 | | KILL |
| 103 | F | SUSPEND |
| 104 | E | ACTIVATE |
| 105 | | GETORIGIN |
| 106 | | MAIL |
| 107 | | SENDMAIL |
| 108 | | RECEIVEMAIL |
| 109 | | FATHER |
| 110 | | GETPROCINFO |

Figure 10-4. Intrinsic Numbers vs. Intrinsics (Continued)

| INTRINSIC NO. | MESSAGE BLOCK LETTER (IN ADDITION TO A) | INTRINSIC NAME |
|---|---|---|
| 112 | | GETPROCID |
| 120 | D | GETPRIORITY |
| 130 | | GETDSEG |
| 131 | | FREEDSEG |
| 132 | | DMOVIN |
| 133 | | DMOVEOUT |
| 134 | | ALTDSEG |
| 135 | | DLSIZE |
| 136 | | ZSIZE |
| 139 | | SWITCHDB |
| 191 | | PTAPE |
| 200 | | GETPRIVMODE |
| 201 | | GETUSERMODE |
| None | | CHECKDEV |

Figure 10-4. Intrinsic Numbers vs. Intrinsics (Continued)

| Message Block A (Common) | |
|---|---|
| Index No. | Message |
| 1 | Illegal DB-register |
| 2 | Illegal Capability |
| 3 | Omitted Parameter |
| 4 | Incorrect S-register |
| 5 | Parameter address violation |
| 6 | Parameter end address violation |
| 7 | Illegal parameter |
| 8 | Parameter value invalid |
| 9 | Incorrect Q-register |

Figure 10-5A. Message Block A

| | Message Block B (File System) | |
|---|---|---|
| **Index No.** | **Message** | |
| 20 | Invalid operation. | |
| 21 | Data parity error. | |
| 22 | Software time-out. | |
| 23 | End of tape. | |
| 24 | Unit not ready. | |
| 25 | No write-ring on tape. | |
| 26 | Transmission error. | |
| 27 | Input/output time-out. | |
| 28 | Timing error or data overrun. | |
| 29 | Start input/output (SIO) failure. | |
| 30 | Unit failure. | |
| 31 | End of time (special character terminator). | |
| 32 | Software abort. | |
| 33 | Data lost. | |
| 34 | Unit not on-line. | |
| 35 | Data set not ready. | |
| 36 | Invalid disc address. | |
| 37 | Invalid memory address. | |
| 38 | Tape parity error. | |
| 39 | Recovered tape error. | |
| 40 | Operation inconsistent with access type. | |
| 41 | Operation inconsistent with record type. | |
| 42 | Operation inconsistent with device type. | |
| 43 | The TCOUNT parameter value exceeded the RECSIZE parameter value in this intrinsic, but the multiaccess aoption was not specified in the currently-effective FOPEN intrinsic. (Applies to output operations only.) | |
| 44 | The FUPDATE intrinsic was called, but the file was positioned at record zero. (FUPDATE must reference the last record read, but no previous record was, in fact, read.) | |
| 45 | Privileged-file violation. | |
| 46 | Insufficient disc space. | |
| 47 | Input/output error occurs on a file label. | |
| 48 | Invalid operation due to multiple file access. | |
| 49 | Unimplemented function. | |
| 50 | The account referenced does not exist. | |
| 51 | The group referenced does not exist. | |
| 52 | The file referenced does not exist in the system file domain. | |

**Figure 10-5B. Message Block B**

| Index No. | Message |
|---|---|
| 53 | The file referenced does not exist in the job temporary file domain. |
| 54 | The file reference is invalid. |
| 55 | The device referenced is not available. |
| 56 | The device specification is invalid. |
| 57 | Virtual memory is not sufficient for the file specified. |
| 58 | The file was not passed. |
| 59 | Standard label violation. |
| 60 | Global RIN not available. |
| 61 | Group disc space exceeded. |
| 62 | Account disc space exceeded. |
| 63 | User does not have the non-sharable device capability. |
| 64 | User does not have the multiple RIN capability. |
| 71 | Too many files for process. |
| 72 | Invalid file number. |
| 73 | Bounds-check violation. |
| 90 | The calling process requested exclusive access to a file to which another process has access. |
| 91 | The calling process requested access to a file to which another process has exclusive access. |
| 92 | Lock word violation. |
| 93 | Security violation. |
| 94 | Creator conflict. |
| 100 | Duplicate file name in the system file directory. |
| 101 | Duplicate file name in the job temporary file directory. |
| 102 | Directory input/output error. |
| 103 | System directory overflow. |
| 104 | Job temporary directory overflow. |
| 110 | The intrinsic attempted to save a system file in the job temporary file directory. |

Figure 10-5B.  Message Block B (Continued)

| | Message Block C (Loader) | |
| --- | --- | --- |
| Index No. | Next Message Block | Message |
| 20 | | Illegal library search. |
| 21 | | Unknown entry point. |
| 22 | | TRACE/3000 subsystem not present. |
| 23 | | Stack size too small. |
| 24 | | Maximum data specified greater than 32,768 words. |
| 25 | | Data segment greater than maximum data segment size permitted. |
| 26 | | Program loaded in opposite mode from that first requested. |
| 27 | | Segmented library binding error. |
| 28 | | Invalid system segmented library file. |
| 29 | | Invalid public segmented library file. |
| 30 | | Invalid group segmented library file. |
| 31 | | Invalid program file |
| 32 | | Invalid list file. |
| 33 | | Code segment greater than system—specified maximum. |
| 34 | | Program uses more than one extent. |
| 35 | | Data segment greater than 32,768 words. |
| 36 | | Data segment greater than system-specified maximum. |
| 37 | | Number of code segments greater than 152. |
| 38 | | Number of code segments greater than system-specified maximum. |
| 39 | | Illegal capability. |
| 40 | | Too many procedures loaded. |
| 41 | | Unknown procedure name. |
| 42 | | Invalid procedure number. |
| 43 | | Illegal procedure unload attempted. |
| 50, | B | Unable to open system segmented library file. |
| 51, | B | Unable to open public segmented library file. |
| 52, | B | Unable to open group segmented library file. |
| 53, | B | Unable to open program file. |
| 54, | B | Unable to open list file. |
| 55, | B | Unable to close system segmented library file. |
| 56, | B | Unable to close public segmented library file. |
| 57, | B | Unable to close group segmented library file. |
| 58, | B | Unable to close program file. |
| 59, | B | Unable to close list file. |
| 60, | B | End-of-file or input/output error on system segmented library file. |

Figure 10-5C.  Message Block C

| Index No. | Next Message Block | Message |
|-----------|-------------------|---------|
| 61, | B | End-of-file or input/output error on public segmented library file. |
| 62, | B | End-of-file or input/output error on group segmented library file. |
| 63, | B | End-of-file or input/output error on program file. |
| 64, | B | End-of-file or input/output error on list file. |
| 65 | | Unable to obtain code segment table entries. |
| 66 | | Unable to obtain process data segment table entry. |
| 67 | | Unable to obtain mail data segment table entry. |
| 68 | | Unable to create load process. |
| 70 | | Segment table overflow. |
| 71 | | Unable to obtain sufficient DL storage |
| 72 | | Attach input/output (ATTIO routine) error. |
| 73 | | Unable to obtain virtual memory. |
| 74 | | Directory input/output error. |
| 75 | | Print input/output error. |
| 76 | | Illegal DLSIZE. |
| 80 | | Program already allocated. |
| 81 | | Illegal program allocation. |
| 82 | | Program not allocated. |
| 83 | | Illegal program deallocation. |
| 84 | | Procedure already allocated. |
| 85 | | Illegal procedure allocation. |
| 86 | | Procedure not allocated. |
| 87 | | Illegal procedure deallocation. |

Figure 10-5C. Message Block C (Continued)

| Message Block D (CREATE) | | |
| --- | --- | --- |
| Index No. | Next Message Block | Message |
| 20 | | Unknown subqueue name. |
| 21 | | Subqueue *A* requested without frozen stack. |
| 23 | | Non-standard subqueue requested without proper capability. |
| 24 | | Unknown portion of master queue. |
| 25 | | Master queue requested without capability. |
| 26 | | Absolute priority requested without capability. |
| 27 | | Illegal priority class specified. |
| 28 | | Priority omitted while father process in master queue. |
| 29 | | Priority rank reserved to supervisor capability. |
| 30, | C | Load error. |
| 31 | | Lack of system resource. |

Figure 10-5D. Message Block D

| Message Block E (ACTIVATE) | |
| --- | --- |
| Index No. | Message |
| 20 | Activation of system process not allowed. |
| 21 | Activation of main process not allowed. |

Figure 10-5E. Message Block E

| Message Block F (SUSPEND) | |
| --- | --- |
| Index No. | Message |
| 20 | Additional capability required. (The *susp* parameter was greater than 3.) |

Figure 10-5F. Message Block F

| Message Block G (MYCOMMAND) | |
| --- | --- |
| Index No. | Message |
| 20 | A parsed parameter of COMIMAGE was greater than 255 characters long. |

Figure 10-5G. Message Block G

| Message Block H (LOCKGLORIN) | |
|---|---|
| Index No. | Message |
| 20 | Incorrect password for RIN. |
| 21 | Standard or process-handling user can lock only one RIN at a time, but more than one was requested. |
| 22 | The RIN requested is not allocated. |
| 23 | The RIN number is too large for the RIN table. |
| 24 | The RIN is not a global RIN. |

**Figure 10-5H. Message Block H**

## USER MESSAGES

When a user's batch job or session receives a message from another user's job or session, that message appears in the following format:

*FROM/[jsname,] username.acctname/message*

*jsname*
*username*  } The names of the transmitting job/session and user, and the name of the
*acctname*    account under which they are running.

*message*     The message.

*EXAMPLE*

*A user identified as BOB running a session named S, under an account named A, sends a message to a user telling him that he is changing the name of a file frequently used by both persons; the receiving user would see the following message:*

```
FROM/S,BOB.A/FILE NAMED BAKER IS NOW RENAMED JONES.
```

## OPERATOR MESSAGES

When a user's batch job or session receives a message from the console operator, that message appears in one of two formats, depending on its degree of urgency. Urgent messages which pre-empt any form of input/output being conducted on the standard list device, appear in this format:

*WARN/message*

*message* is the message text.

Less serious messages used for normal communication between the operator and user do not pre-empt input/output in progress, and appear on the standard list device is this format:

*FROM/OPERATOR/message*

*message* is the message text.


## SYSTEM MESSAGES

Miscellaneous conditions that terminate or otherwise affect a users' job/session are reported through system messages, shown in Figure 10-6. These messages may appear, asynchronously, during the course of a running job/session on the standard list device.

* $\left\{ \begin{array}{c} \text{SESSION} \\ \text{JOB} \end{array} \right\}$  ABORTED BY SYSTEM MANAGEMENT*

The job/session has been aborted by the computer operator or system supervisor user through the appropriate command. An immediate log-off then takes place.

* $\left\{ \begin{array}{c} \text{SESSION} \\ \text{JOB} \end{array} \right\}$  HAS EXCEEDED TIME LIMIT*

The job/session has exceeded the time limit which was specified in the TIME=parameter of the JOB/HELLO command. An immediate log-off then takes place.


WARNING:        PRIORITY = XXX

The priority passed to the CREATE intrinsic resulted in a conflict with another process, and the priority then assigned was XXX instead of the requested value.


LMAP NOT AVAILABLE

An LMAP of the process being created, or program file being :RUN, is not available because the code segments are already loaded.

Figure 10-6.  System Messages

10-30

## FILE INFORMATION DISPLAY

In addition to Command Interpreter and run-time (abort) error messages, certain file input/ output errors result in the output of a file information display. For files not yet opened, or for which the FOPEN intrinsic failed, this display appears as in the example below.

```
      +-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
 ①—-!   FILE NUMBER #      IS UNDEFINED.               !
 ②—-!   ERROR NUMBER: 56   RESIDUE: 0                  !
 ③—-!   BLOCK NUMBER: 0              NUMREC: 0          !
      +--------------------------------------------------+
```

In this display, the lines indicated show the following information:

| Line | Content |
|------|---------|
| 1 | A message stating why the file could not be opened. |
| 2 | The appropriate *error number* relating to Figure 10-5B; in this case, an invalid device specification occurred. The *residue*, which is the number of bytes *not* transferred in an input/output request; since no such request applies in this case, this is *zero*. |
| 3 | The *block number*, indicating the physical record where the error was encountered. The *numrec* (number of logical records in the block). In this case, since the file was not opened, both *block number* and *numrec* are zero. |

For files that were open when a CCG (end-of-file error) or CCL (irrecoverable file error) condition code was returned, the file information display appears as shown in this example:

```
       +-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
  ①—-!   FILE NAME IS FTN05      .                       !
  ②—-!   FOPTIONS: NEW,A,   $STDIN,U,N,SL,FEQ            !
  ③—-!   AOPTIONS:   INPUT,SREC,NOLOCK,DEF,NOBUFF        !
  ④—-!   DEVICE TYPE: 16 LU: 17   DRT: 13    UNIT: 6     !
  ⑤—-!   RECORD SIZE: 72     BLOCK SIZE: 72    (BYTES)   !
  ⑥—-!   EXTENT SIZE: 0      MAX EXTENTS: 0              !
  ⑦—-!   RECPTR: 0           RECLIMIT: 0                 !
  ⑧—-!   LOGCOUNT: 0            PHYSCOUNT: 0             !
  ⑨—-!   EOF AT: 0           LABEL ADDR: %02100000000    !
  ⑩—-!   FILE CODE: 0        ID IS          ULABELS: 0   !
  ⑪—-!   PHYSICAL STATUS: 0001000100000000              !
  ⑫—-!   ERROR NUMBER: 0     RESIDUE: 0                  !
  ⑬—-!   BLOCK NUMBER: 0              NUMREC: 1          !
       +--------------------------------------------------+
```

The lines indicated show the following information:

| Line | Contents |
|------|----------|

1      The *file name:* in this case, the name if *FTN05*.

2      The *foptions* in effect, including:

Domain:
- NEW = New file (as in this case).
- SYS = System file domain.
- JOB = Job temporary file domain.
- ALL = System *and* job temporary file domain.

Type:
- A = ASCII File (as in this case).
- B = Binary File.

Default File
Designator:
- *FORMAL* = Actual file designator is same as formal file designator.
- (In this case, actual designator is $STDIN.)

Record
Format:
- F = Fixed length
- V = Variable length.
- U = Undefined length (as in this case).
- ? = Unknown format.

Carriage
Control:
- N = None (as in this case).
- C = Carriage control character expected.

File Format:
- SL = An SL file (as in this case).
- RL = An RL file.
- STAR = A STAR/3000 file.
- PROG = A program file.
- BASFP = A BASIC/3000 fast program file.
- BASP = A BASIC/3000 program file.
- BASD = A BASIC/3000 data file.
- USL = A USL file.

File Equation
Option:
- FEQ = :FILE allowed (as in this case).
- DEQ = :FILE not allowed.

3      The *aoptions* in effect, including:

Access Type:
- INPUT = Read access (as in this case).
- OUTPUT = Write access.
- OUTKEEP = Write-only access, without deleting.
- APPEND = Append access.
- IN/OUT = Input and output access.
- UPDATE = Update access.

| Line | | | Contents |
|------|---|---|---------|

| Line | Contents |
|------|----------|
| 3 (Cont.) | Multi-record Option:    SREC = Single record access (as in this case). <br>        MREC = Multi-record access. |

**3 (Cont.)**

| | | | |
|---|---|---|---|
| Multi-record Option: | SREC | = | Single record access (as in this case). |
| | MREC | = | Multi-record access. |
| | | | |
| Dynamic Locking Option: | NOLOCK | = | No locking permitted (as in this case). |
| | LOCK | = | Locking permitted. |
| | | | |
| Exclusive Access Option: | DEF | = | Default specification (as in this case). |
| | EXC | = | Exclusive access allowed. |
| | SEA | = | Semi-exclusive access allowed. |
| | SHR | = | Sharable file. |
| | | | |
| Buffering: | BUFFER | = | Automatic buffering. |
| | NOBUFF | = | Inhibit buffering (as in this case). |

**4**    The *Device Type, LU (Logical Device Number), DRT (Device Reference Table Entry Number)* and *Unit* of the device on which the file resides. (These are 16, 17, 18, and 6 respectively, in this case.)

**5**    The *record size* and *block size* of the offending record, in *bytes.* (In this case, these are both specified as 72 bytes.)

**6**    The *extent size* (of the current extent) and the *max extents* (maximum number of extents) allowed the file.

**7**    The *recptr* (current record pointer) and *reclimit* (limit on number of records in the file).

**8**    The *logcount* (present count of logical records) and *physcount* (present count of physical records) in the file).

**9**    The *EOF at* (location of the current end-of-file) and the *label addr* (location of the header label of the file).

**10**    The *file code, id* (name of creating user), and *ulabels* (number of user-created labels) for the file.

**11**    The *physical status* of the file.

**12**    The *error number* and *residue*, as described under the abbreviated file information display format, above.

**13**    The *block number* and *numrec*, as described under the abbreviated file information display format, above.

# PART 3
## Optional Capabilities

# SECTION XI
# Process-handling Optional Capability

As noted earlier, all user and system programs under MPE/3000 are run on the basis of *processes*—the basic executable entities in the operating system. Processes are invisible to the programmer accessing MPE/3000 through the *standard capabilities* described in Part II of this manual. This programmer has no control over processes or their structure; for him, MPE/3000 automatically creates, handles, and deletes all processes. But, the user with certain *optional capabilities* can interact with processes directly. The most basic of these optional capabilities is the *Process-Handling Optional Capability*, discussed in this section. This capability, assigned and used independently of the other optional capabilities, allows the user to

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

These operations can be very useful to a user. For instance, they allow him to have several independent processes running concurrently on his behalf, all communicating with one another.

> *NOTE:* In order to fully understand the information in this section, the user should first read Section II of this manual, "How MPE/3000 Operates." That section introduces background information, definitions, and concepts that are further developed in this section.

## PROCESS LIFE-CYCLE

The user can activate processes to run any kind of code—programs, subroutines, or procedures. To illustrate how processes are created and managed, the steps in the life-cycle of a typical user process are discussed below and illustrated in Figure 11-1.

1. The process life-cycle begins with a source program on punched cards (for example) which is compiled (by a process running a compiler) into relocatable binary modules (RBM's) on a user subprogram library (USL) file. This USL resides on disc.

TIME ⟶

PROCESS "RUNS"          I/O START          TERMINATE

EXECUTING

CPU TIME

EXECUTABLE

MEMORY SPACE

ACTIVE

SUSPEND
[STATES]

ALIVE

PROGRAM FILE

OBJECT (USL) FILE

SOURCE FILE

① Compile
② Prepare
③ Create
④ Activate
⑤ Dispatch
⑥ Execute
⑦ Remove from Execution
⑧ Return to Execution
⑨ Suspend for I/O

⑪ Delete Process

⑩ Return to Active State
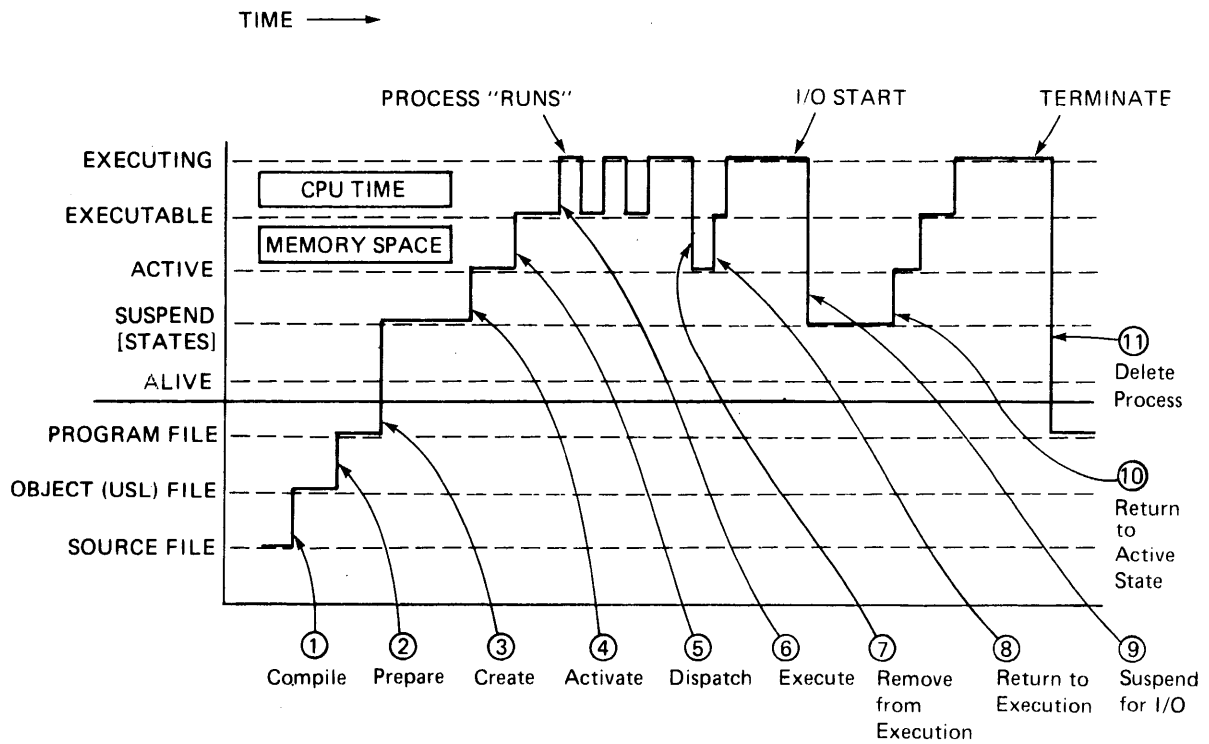
Figure 11-1. Process Life Cycle

2. The program is next prepared for execution (by a process running the MPE/3000 Segmenter). The program now resides as linked code segments on a program file which also contains the initially-defined stack for the process to be executed.

3. When an explicit or implied :RUN command is issued, allocation/execution begins with the creation of the process. The :RUN command invokes the CREATE intrinsic, which establishes a Process Control Block (PCB) entry in the PCB table, and calls the LOAD intrinsic to obtain Code Segment Table (CST) and Data Segment Table (DST) entries and virtual memory space for the process, initialize global variables, satisfy external references, and load the program into virtual memory. The LOAD intrinsic returns control to the CREATE intrinsic, which then calls the PROCREATE intrinsic to transform the virtual-memory program into a process. The PROCREATE intrinsic formats the PCB Extension, writes return and entry stack markers on the initially-defined stack, links the new process to its family, inserts the PCB entry into the master scheduling queue or subqueue according to its priority, and places the process in the *alive* state by turning on the *alive* bit in the PCB. The new process now exists, but it is not yet active and thus cannot be executed.

At this point, all the basic elements of the process exist—the PCB that defines and controls the process, the code segments that the process executes, and the data stack upon which the process operates. Within MPE/3000, the process is identified by a unique number called a process identification number (PIN). The significant tables resident in main memory (Figure 11-2) that relate to the process are

a. *The PCB Table* contains one entry (PCB) for each process. The PCB holds information needed during the entire life of the process, whether or not it is running in main memory. When the process executes, the contents of the PCB change continually. Among other elements, the PCB contains: links to the process' father and sons, its priority, *alive* bit, and *active* bit. In addition to the PCB, the PCB Extension (physically part of the stack segment) contains process control information needed when the process is running in main memory. The PCB Extension need not be continually resident in main memory, since it may be swapped out to disc with the stack segment if the process is interrupted. The information in the PCB Extension includes the process' most recent register settings and information about files referenced by the process.

b. *The CST* contains, for each code segment in virtual memory, a two-word entry. This entry shows the length and current address of the segment and bit-settings that indicate:

   • Whether the segment is present in or absent from main memory.

   • Whether the segment runs in privileged or non-privileged mode.

   • Whether the segment calls the TRACE routine (as distinct from the TRACE/3000 facility).

   • Reference information, used statistically by MPE/3000.
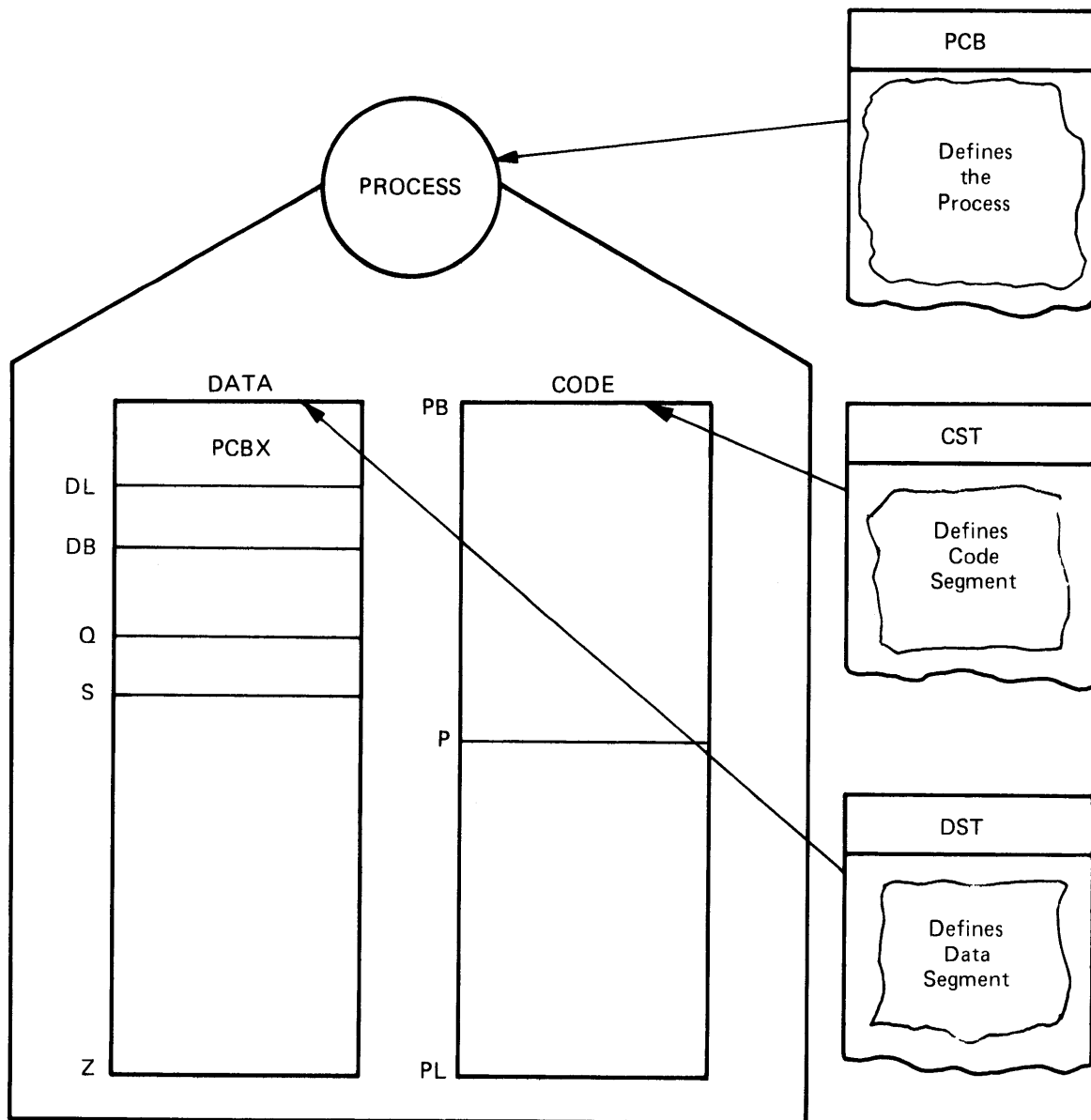
Figure 11-2. Process Components and Tables

c. *The DST* contains, for each data segment in virtual memory, a two-word entry. This entry shows the length and current address of the segment, and bit-settings that indicate:

- Whether the segment is present in or absent from main memory.
- Reference information, used statistically by MPE/3000.

4. The father of this process (in this case, the job main process) calls the ACTIVATE intrinsic which places the process in the *active* substate by turning the active bit in the PCB *on.* Now, when the dispatcher scans the scheduling queue, it will recognize the process as active.

5. The dispatcher calls the MAPP (Make-a-Process-Present) process to move the process' data stack into main memory. This moves the process from the *active* to the *executable* state. As soon as the process becomes the highest-priority process that is both *present* and *active* in the queue, the dispatcher allocates it central processor time, sets the stack segment registers (DL,DB,Q,S and Z), and transfers control to the starting address (entry-point label) of the initial code segment.

6. The process now enters execution. From the entry-point on, it follows the sequence dictated by its code segments until its suspension or deletion. At any point, the process may call the CREATE intrinsic to create a son process, using a program file specified by the user for its code segments. It may then call the ACTIVATE intrinsic to activate the son process. At this time, the father process may suspend itself (through an implied or explicit call to the SUSPEND intrinsic) or may run concurrently with its son.

   Figuratively speaking, a father process is responsible for what happens to its son—creation, activation, suspension, deletion, or other special operations. Each process carries a set of logical pointers linking it to other members of its family: one pointer indicates its father and one or more other pointers correspond to its sons. (All of these pointers are actually PIN's.) The pointers function as shown in Figure 11-3, with the solid lines indicating father pointers and the broken lines indicating son pointers.

7. If a new process is dispatched, the memory manager may remove the current process' stack from main memory and overlay its code segments with those of the new process (swapping). The original process is still active in virtual memory, but is no longer executable until its segments are returned to main memory.

8. When the original process' segments are returned to main memory, that process resumes execution.

9. A request for blocked input/output suspends the process while input/output is performed. In this case, the process' segments are again swapped from main memory to disc. (Note, however, that segments are not *always* swapped out when a process is suspended for input/output — this occurs here only for illustrative purposes.)

10. When input/output is complete, the process again becomes active, then executable, and then runs to completion or until the next interruption.
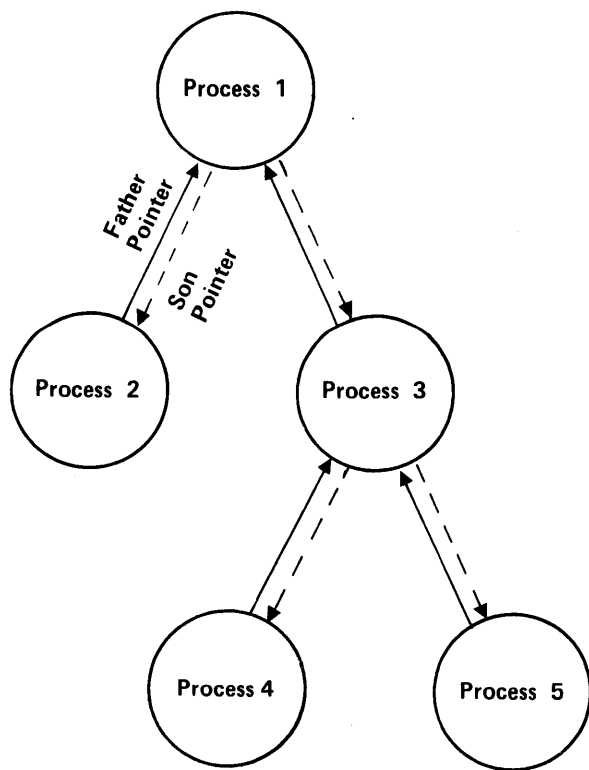
Figure 11-3. Process Linking

11. When the process is completed, it typically deletes itself by calling the TERMINATE intrinsic. This intrinsic, and others it invokes, accomplish the following:

    a.    Close all open files associated with the process (based on information from the PCB Extension).

    b.    Deallocate all resources, such as CST and DST entries, code and data space, and system input/output buffers.

    c.    Request the UCOP to delete the PIN, and remove the process from the process family structure and from the queue by deleting the PCB.

    d.    Turn on the *active* bit in the father process (main process, in this case) if specified by the user.

The process is now deleted from the system. The user code still exists as a program file on disc. To execute this code again, the user must create a new process.

## PROCESS-HANDLING

The user creates, manages, and deletes processes with the intrinsics described below.

### Creating Processes

Any running process can request the creation of a son process by issuing the CREATE intrinsic call, described below. 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 PIN to the calling process.

The CREATE intrinsic is written in the following format:

*PROCEDURE*
| |
|---|
| *CREATE*    *(progname,* entryname,*pin,* param,flags, stacksize,dlsize,maxdata,priorityclass,rank;) |

*VALUE* param,stacksize,dlsize,priorityclass,maxdata,flags,rank;

*LOGICAL* priorityclass,flags;

*INTEGER* stacksize,dlsize,maxdata,*pin,* param,rank;

*BYTE ARRAY* *progname,* entryname;

*OPTION VARIABLE, EXTERNAL;*

The parameters for this intrinsic call are as follows. All parameters except *progname* and *pin* are optional.

| | |
|---|---|
| *progname* | A byte-array containing a string, terminated by a blank, specifying the name and optionally, the account and group (*filereference* format) of the file containing the program to be run. |
| **entryname** | A byte array, containing a string terminated by a blank, specifying the entry-point (label) in the program where execution is to begin when the process is activated. The *primary* entry-point in the program can be specified in this array by a blank character alone. If *entryname* is omitted, the primary entry-point is specified by default. |
| *pin* | A word in which the PIN of the new process is returned to the calling process. This PIN is used in other intrinsics to reference the new process. The PIN can range from 1 to 255. (If an error is detected, a PIN of zero is returned.) |
| **param** | 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). If *param* is omitted, this word is filled with zeros. |
| **flags** | A word whose bits, if on, specify loading options: |

Bit (15:1) = *ACTIVE Bit.* If this bit is *on*, MPE/3000 reactivates the calling process (father) when the new process terminates. If this bit is *off*, the calling process is not activated at that time. The default setting is *off.*

Bit (14:1) = *LOADMAP Bit.* If this bit is *on*, a listing of the allocated (loaded) program is produced on the job/session listing device. This map shows CST entries used by the new process. If this bit is *off*, no map is produced. The default setting is *off.*

Bit (13:1) = (Not used.)

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

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

00 = System Library.

01 = Account Public Library, followed by System Library.

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

The default setting for these bits is 00.

11-8

Bits (0:10) = (Not used by MPE/3000).

If *flags* omitted, all default values noted are taken.

**stacksize**      An integer (Z-Q) denoting the number of words assigned to the local stack area (the area bounded by the initial Q and Z registers). The default value is that specified in the program file.

**dlsize**      An integer (DB-DL) denoting the number of words in the user-managed stack area (bounded by the DL and DB registers). The default value is that specified in the program file.

**maxdata**      The maximum size allowed for the process' stack (Z-DL) area in words. When specified, this value overrides the one established at program-preparation time. The default value is specified at system generation time.

**priorityclass**      A string of two ASCII characters describing the priority class (subqueue) in which the new process is scheduled. For users having the process-handling capability as their *only* optional capability, this may be: "CS," "DS," or "ES," as described in the discussion *Rescheduling Processes* later in this section. (For users with other optional capabilities, the possible entries also include "AS" and "BS".) The default value is the priority of the calling process.

**rank**      The relative rank of the process within a linear subqueue. (This parameter is available only to users with the *System Supervisor Capability*, as discussed in *HP 3000 Multiprogramming Executive System Manager/Supervisor Capabilities (03000-90038).)*

*NOTE:*    *For the* stacksize, dlsize, *and* maxdata *parameters, a value of –1 indicates that the MPE/3000 Segmenter is to assign default values; specifying –1 is equivalent to omitting the parameter.*

When the CREATE intrinsic is called, the DB register must be pointing at the user's stack.

The CREATE intrinsic returns one of the following condition codes to the calling process:

**CCE**      Request was granted; the new process is created.

**CCG**      Request was granted; the *maxdata* and/or *dlsize* parameters given were illegal, but other values were used. Specifically,

    1.    If the *maxdata* specified exceeds that maximum Z-DL allowed by the configuration, then that configuration maximum value is assigned.

    2.    If (*dlsize* + 100) modulo 128 is *not* zero, then *dlsize* is rounded upward so that (*dlsize* + 100) modulo 128 = 0.

**CCL**      The request was not granted because the *progname* or *entryname* specified does not exist.

The creating process is aborted if:

1. Request was rejected because of illegal parameters; a PIN of zero is returned. Specifically, this occurs:

   - If *progname* is illegal.

   - If *entryname* is illegal.

   - If *stacksize* is less than 512 (decimal) and is not –1. *(Note than –1 specifies the default value.)*

   - If *dlsize* is less than 0 and is not –1.

   - If *maxdata is less than or equal to 0, and is not –1.*

   - If *(dlsize + globsize + stacksize +* 128) exceeds *maxdata*. Note that *dlsize* may have been modified to satisfy Condition 2, under CCG, described above. (The *globsize* value is the sum of the primary DB plus the secondary DB values — the total DB given at program preparation time by the program map (PMAP).)

   - If *(dlsize + globsize + stacksize +* 128) exceeds the maximum *stacksize* defined during system configuration. Note that *dlsize* may have been modified to satisfy Condition 2, under CCG, described above.

   - If *(maxdata +* 90) exceeds 32768, where *maxdata* is either the value passed as a parameter or a value re-computed by the Loader under Condition 1 of CCG (above).

2. The user does not have the *Process-Handling Optional Capability.*

3. An illegal value (a non-existent subqueue) was specified for the *priorityclass* parameter.

4. A required parameter *(progname* or *pin)* is omitted.

5. A reference parameter was not within the required range.


*EXAMPLE:*

*Suppose the user wants to create a process that, when activated, runs a program in the file PROGA, beginning at the entry-point START1. He also wants the libraries searched in the following order: account public library, followed by system library. The process' PIN is to be returned to the word PINRET. The process is to be assigned the priorityclass CS. The user enters the following intrinsic call. (The byte array PROG contains "PROGA", the byte array ENTER contains "START1", and the tenth bit of the word FLAGGER is on.)*

**CREATE (PROG,ENTER,PINRET,,FLAGGER,,,,"CS");**

The PIN assigned to the processes is 026, returned to the word PINRET.

## Activating Processes

After a process has been created, it must be activated in order to run. When this is done, 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 call discussed later) can be reactivated by its father or any of its sons (as specified in the SUSPEND call *susp* parameter). To activate or reactivate a process, the ACTIVATE call is issued:

PROCEDURE $\boxed{ACTIVATE \ (pin, \text{susp}) \ ;}$

*VALUE pin*, susp;

*LOGICAL* susp;

*INTEGER pin;*

*OPTION VARIABLE, EXTERNAL;*

The ACTIVATE call parameters are

*pin*       An integer specifying the PIN of the calling process' father or son to be activated. (A non-zero integer is used for sons, and zero is used for the father.) In order for the ACTIVATE intrinsic to work, the receiving process must be expecting the activation signal from the calling process (as noted in the discussion of the SUSPEND intrinsic.)

susp      A word that specifies whether or not the calling process is to be suspended when the called process is activated. When *susp* is not present or is set zero, the calling process remains active. When *susp* is specified, the calling process is suspended. The 14th and 15th bits of *susp* specify the anticipated source of the call that later will reactivate the calling process.

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

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

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

Bits (0:14)   =   Not usable by programmers with *only* the *Process-Handling Optional Capability.*

MPE/3000 guarantees that no interrupts occur between activation of the called process and suspension of the calling process.

The following condition codes can be returned.

CCE      Request granted; Process *pin* is activated. The calling process is suspended if *susp* was specified.

CCG      Process *pin* is already active. The calling process is suspended if *susp* was specified.

CCL Request rejected, because Process *pin* was not expecting activation by this calling process; an illegal *pin* parameter was specified; or the *susp* parameter was not valid.

The process is aborted if:

1. The user does not have the *Process-Handling Optional Capability.*

2. A required parameter is omitted.

3. A job or session Main Process, or a System Process, is specified.


*EXAMPLE:*

*To activate the process whose PIN is stored in PINVAL, but without suspending itself, the calling process issues:*

**ACTIVATE (PINVAL);**


## Suspending Processes

A process can suspend itself by issuing the SUSPEND intrinsic call. 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 call. The format of the SUSPEND call is

*PROCEDURE*   | *SUSPEND (susp,*rin*) ;* |

*VALUE susp,*rin*;*

*LOGICAL susp;*

*INTEGER* rin*;*

*OPTIONAL VARIABLE, EXTERNAL;*

The parameters are

*susp* 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 Optional Capability*, at least *one* of these bits must be *on*.

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

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

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

11-12

Bits (0:14)  =   (These bits are not used by users with only the *Process-Handling Optional Capability.*)

rin          A RIN designation, (defined in Section IX). If *rin* is specified, it represents a *local* RIN locked by the process but released when the process is suspended. This facility can be used to synchronize processes within the same job. If *rin* is omitted, no RIN is unlocked.

The following condition codes can be returned:

CCE          Request granted.

CCG          (Not returned.)

CCL          Request rejected, because of invalid parameter (*susp* not valid, *RIN* not owned by process, or *RIN* not locked.)

The process is aborted if the user issuing the SUSPEND call does not have the Process-Handling Capability.


*EXAMPLE:*

*The following intrinsic call suspends the calling process, which then expects to be reactivated by its father. (The source of reactivation is specified by setting bit 15 of the word PARENT on.) No RIN's are involved.*

**SUSPEND (PARENT);**


## Deleting Processes

A process can request the deletion of itself, or any of its sons at any time. When this is done, all code and data segments in the process, and all resources owned by the process, are released; all temporary files opened by the process are closed; and finally, the PIN is released. When a process is deleted, MPE/3000 also automatically deletes all descendants of that process, as shown in Figure 11-4. Within a process tree structure, the newest generations are deleted first. Within each generation, processes are deleted in order of their creation.

A process deletes itself by issuing the TERMINATE intrinsic call.

In a job or session main process, the TERMINATE intrinsic is automatically invoked by detection of an end-of-job/session condition. This intrinsic removes the job or session from the system.

The format of the TERMINATE call is

*PROCEDURE*      | *TERMINATE ;*

*OPTION EXTERNAL;*

11-13

First (oldest)
Generation

Second
Generation

Third
Generation

Fourth (newest)
Generation

Process 0

Process 1

Process 2

Process 3

Process 4
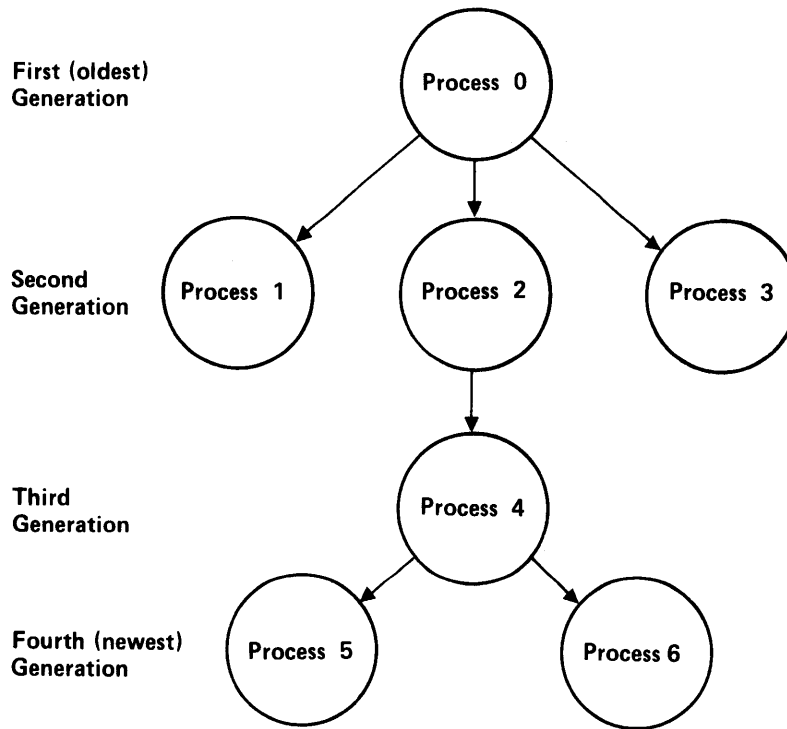
Process 5

Process 6

(When Process 2 is deleted, the following structure results.)

First
Generation

Second
Generation

Process 0
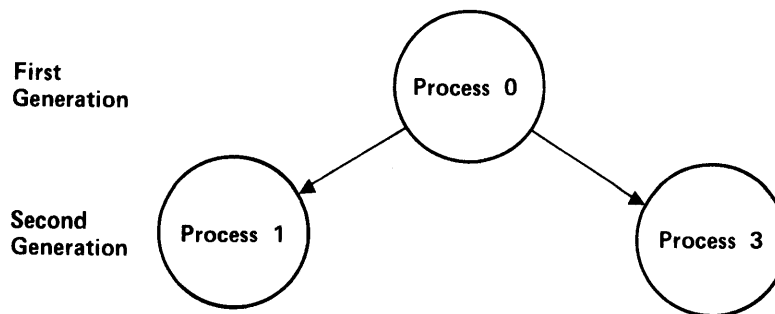
Process 1

Process 3

Figure 11-4. Process Deletion

A process can delete one of its sons by issuing the KILL intrinsic call

*PROCEDURE*  | *KILL (pin) ;*

*VALUE pin;*

*INTEGER pin;*

*OPTION EXTERNAL;*

In this call, *pin* is a word containing the PIN of the process to be deleted; it can be an integer value ranging from 1 to 255.

The following condition codes can be returned by the KILL intrinsic:

CCE     Request granted.

CCG     Request granted; the specified process was already terminating.

CCL     Request denied because an illegal PIN was specified.

## Interprocess Communication

The user can direct the communication of information between processes. This information transfer, however, is restricted to upward or downward paths through the process tree structure, so that any process can communicate only with its father or sons. Between any father/son pair, only one such transfer is allowed at any particular time.

Information transferred between processes is referred to as *mail*. It is sent from one process to another through an intermediate storage area called a *mailbox*. At any given time, a mailbox can contain only one item of mail (a *message*). For any process, there are two sets of mailboxes:

- The mailbox used for communication between the process and its father
  (each process has one of these)

- The set of mailboxes used for communication between the process and its sons
  (each process has one of these mailboxes for each of its sons)

The transmittal of mail is based upon a transaction between the sending and receiving processes that involves the following steps:

1.  Optionally, the sending process tests the mailbox to determine its status (whether it is empty, contains a message, or is being used by the receiving process).

2.  The sending process transmits the mail to the mailbox. (The message transferred is a word array in the sending process' stack, defined by a starting location and word count. The smallest message allowed is a single word. MPE/3000 automatically performs a *bounds check* that ensures that the array specified actually falls within the limits of the process' stack.)

3.  The receiving process optionally tests the mailbox to determine its status.

4.  If the mailbox contains a message, the receiving process collects this mail. If the mail is not collected, it is overwritten by additional mail from the sending process. (When the mail is collected, another bounds check is performed to validate the address given for the stack of the receiving process.)

## Testing Mailbox Status

A process can determine the status of the mailbox used by its father or son, with the MAIL intrinsic call. 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: this enables the calling process to initialize its stack in preparation for receipt of the message. The MAIL intrinsic call format is

*LOGICAL PROCEDURE*    |  *MAIL (pin, count) ;*  |

*VALUE pin;*

*INTEGER pin,count;*

*OPTION EXTERNAL;*

This intrinsic returns as the value of MAIL, the *status* of the mailbox, as noted below.

The parameters for this call are

*pin*    An integer specifying the mailbox tested. If this integer specifies the mailbox of a son process, it must be the PIN of that son. If it specifies the mailbox of a father process, it must be zero.

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

The mailbox status returned to the calling process (as the value of MAIL) will be one of the following digits:

| Status Returned | Meaning |
|---|---|
| 0 | The mail box 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 found in *count*. |
| 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. |

In addition to the mailbox status code, the MAIL intrinsic returns the following condition codes:

CCE          Request granted; the mailbox status was tested.

CCG          Request denied because an illegal *pin* parameter was specified (MAIL has the value of 3).

CCL          (Not returned by this intrinsic.)

If the user does not have the *Process-Handling Capability*, the calling process is aborted.


*EXAMPLE:*

*To test the status of the mailbox associated with one of its son processes, (whose PIN is stored in the word SONNY), a process issues the following call:*

**STATCOUNT := MAIL (SONNY,MCOUNT);**

*Because the mailbox contained an incoming message 10 words long, the integer 10 is stored in the word MCOUNT, and the status code 2 is stored in the word STATCOUNT.*


## Sending Mail

A process sends mail to its father or sons by issuing the SENDMAIL intrinsic call. 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 upon the *waitflag* parameter specified in SENDMAIL. If the mailbox is currently being used by other intrinsics, the SENDMAIL intrinsic waits until the mailbox is free and then sends the mail.

The SENDMAIL intrinsic call format is

*LOGICAL PROCEDURE*      | *SENDMAIL (pin,count,location,waitflag) ;* |

*VALUE pin,count,waitflag;*

*INTEGER pin,count;*

*LOGICAL waitflag;*

*ARRAY location;*

*OPTION EXTERNAL;*

This intrinsic returns, as the value of SENDMAIL, one of the status codes noted later in this discussion.

The call parameters are

*pin*  
An integer specifying the process to receive the mail. If a son process is specified, the integer is the PIN of that process. If a father process is specified, the integer is zero.

*count*  
An integer specifying the length of the message, in words transmitted from the sending process' stack. If zero is specified, SENDMAIL empties the mailbox of any previous incoming or outgoing mail.

*location*  
The starting address of the array containing the message in the sending process' stack (relative to the address in the DB register).

*waitflag*  
A word specifying (in Bit 15) the action to be taken if the mailbox contains previously-sent mail:

TRUE  = Wait until the receiving process collects the previous mail before sending current mail. (Bit 15 = 1)

FALSE = Cancel (overwrite) any previously-sent mail with the current mail. (Bit 15 = 0)

The SENDMAIL intrinsic returns one of the following status codes to the user's program.

| Status Returned | Meaning |
|---|---|
| 0 | The mail was successfully transmitted; the mailbox contained no previous mail. |
| 1 | The mail was successfully transmitted; 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 successfully transmitted because the mailbox contained incoming mail to be collected by the sending process (regardless of the *waitflag* setting). |
| 3 | An error occurred because an illegal *pin* was specified, or a bounds-check failed. |
| 4 | An illegal wait request would have produced a deadlock. |
| 5 | The request was rejected because *count* exceeded the maximum mailbox size allowed by the system. |
| 6 | The request was rejected because storage resources for the mail data segment were not available. |

The SENDMAIL intrinsic returns one of the following condition codes:

CCE   Request granted; the mail was sent.

CCG   Request not granted because of an illegal *count* parameter (SENDMAIL has the value of 5), or an illegal *pin* (SENDMAIL has the value of 3), or storage for the mail data segment was not available (SENDMAIL has the value of 6).

CCL   Request not granted because the bounds-check revealed that the *count* or *location* parameters did not define a legal stack area (SENDMAIL has the value of 3) or both processes are waiting to send mail (SENDMAIL has the value of 4).

If the user does not have the *Process-Handling Capability*, the process is aborted.

*EXAMPLE:*

*To send mail (defined in the stack by an array called LOCAT, and 3 words long) to its father, a process issues the following call. If the mailbox contains a previously-sent message, that message will be overwritten. (The word CNTR contains 3, and Bit 15 of the word NOWAIT is set off).*

```
STAT := SENDMAIL (O,CNTR,LOCAT,NOWAIT);
```

*Because the mailbox contained no previous mail, and the sender's mail was successfully transmitted, the value 0 is stored in the word STAT.*

## Receiving (Collecting) Mail

A process collects mail transmitted to it (from its father or son) by calling 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.

The RECEIVEMAIL intrinsic call is written in the following format:

*LOGICAL PROCEDURE*   | *RECEIVEMAIL (pin,location,waitflag) ;* |

*VALUE pin,waitflag;*

*INTEGER pin;*

*LOGICAL waitflag;*

*ARRAY location;*

*OPTION EXTERNAL;*

This intrinsic returns, as the value of RECEIVEMAIL, the mailbox status as noted below.

The call parameters are

*pin*        An integer specifying the process sending the mail. If a son process is specified, the integer is the PIN of that process. If a father process is specified, the integer is zero.

*location*   The starting address of the word array in the receiving process' stack where the collected message is to be written. (This address is relative to the address in the DB register.)

*waitflag*   A word specifying the action to be taken if the mailbox is empty:

TRUE       = Wait until incoming mail is ready for collection.
             (Bit 15 = 1)

FALSE      = Return immediately to the calling process. (Bit 15 = 0)

The status code indicating the result of the RECEIVEMAIL intrinsic will be one of the following:

| Status Returned | Meaning |
|---|---|
| 0 | The mailbox was empty (and the *waitflag* parameter was FALSE). |
| 1 | No message was collected because the mailbox contained outgoing mail from the receiving process. |
| 2 | The message was successfully collected. |
| 3 | An error occurred because of an illegal pin or bounds check failure. |
| 4 | The request was rejected because *waitflag* 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*. |

The RECEIVEMAIL intrinsic call returns one of the following condition codes:

CCE        Request granted; the mail was collected.

CCG        Request not granted because of illegal *pin* parameter (RECEIVEMAIL has the value of 3).

CCL        Request not granted because the bounds check revealed that the *location* parameter did not define a legal stack address (RECEIVEMAIL has the value of 3) or because both sending and receiving processes would be awaiting incoming mail (deadlock). (RECEIVEMAIL has the value of 4).

If the user does not have the *Process-Handling Capability*, the process is aborted.


*EXAMPLE:*

*To collect a message from a son process (whose PIN is contained in P2) and store this message in the stack array MDATA, a process could issue the following call. If the mailbox is empty, the RECEIVEMAIL intrinsic waits until a message arrives. (Bit 15 of the word WAIT is set on.)*

### STAT := RECEIVEMAIL (P2,MDATA,WAIT);

*Because the mail is successfully collected, the mailbox status code 2 is returned to the word STAT.*


## Avoiding Deadlocks

Since the simultaneous use of mail-transmission, process suspension, and RIN-locking intrinsics throughout a process structure could result in a deadlock if the intrinsic calls are not properly synchronized, the user should be aware of the following facts:

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/3000 does not determine whether all other processes in the tree are suspended. The user must exercise caution in avoiding such a situation.

2. An attempt by a process to lock a global RIN will succeed only if two conditions are met:

   a. No other process within the job/session has currently locked this RIN—a *global* RIN cannot be used as a *local* RIN, because deadlock within the same job/session could otherwise occur.

   b. The calling process currently has no other global RIN locked for itself; this could otherwise result in deadlock between two job/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 through the GETPRIORITY intrinsic call. (A process can change its own priority or that of a son but it cannot reschedule its father.)

Generally, MPE/3000 schedules processes in linear or circular subqueues, as described in Section II. The standard linear subqueues are

- The *AS* subqueue, containing processes of very high priority.

- The *BS* subqueue, containing processes of high priority.

- The *ES* subqueue, containing idle processes with low priority.

The circular subqueues are

- The *CS* subqueue, actually composed of two sub-subqueues, used for interactive and multiprogramming batch processes.

- The *DS* subqueue, also composed of two sub-subqueues, available for general use at a lower priority than the CS subqueue.

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


NOTE:   *Only the CS, DS, and ES priority classes are available to users with the Process-Handling Capability as their only optional capability (in addition to the standard capabilities). Users with Privileged Mode Optional Capability can schedule in other subqueues, as noted in Section XIV.*


The format of the GETPRIORITY intrinsic call is

*PROCEDURE*    | *GETPRIORITY (pin,priorityclass,rank );*

*VALUE pin,priorityclass,rank;*

*LOGICAL priorityclass;*

*INTEGER pin,*rank;

*OPTION VARIABLE, EXTERNAL;*

The parameters for the GETPRIORITY intrinsic are

| | |
|---|---|
| *pin* | An integer denoting the process whose priority class is to be changed. If this is a son process, the integer is the process' PIN. If this is the calling process, the integer is zero. |
| *priorityclass* | A string of two ASCII characters describing the priority class (subqueue) in which the process is rescheduled. For users with only the Process-Handling Optional Capability, this may be: "CS," "DS," or "ES." For users with other optional capabilities, this may be any subqueue or portion of the master queue permitted by those capabilities. (A process can be scheduled in the master queue by specifying $x$A, where $x$ is an actual priority number.) |
| rank | The relative rank of the process within a linear subqueue. (This parameter is available only to users with the *System Supervisor Capability*, as discussed in *HP 3000 Multiprogramming Executive System Manager/ Supervisor Capabilities (03000-90038).)* |

The GETPRIORITY intrinsic returns one of the following condition codes:

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

The process is aborted if the user does not have the Process-Handling Capability or specifies a non-existent priority class (subqueue).

*EXAMPLE:*

*To reschedule itself with the priority class "ES," a process issues the following call:*

```
GETPRIORITY (0,"ES");
```

## 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. It does this by issuing the GETORIGIN call:

*INTEGER PROCEDURE*  $\boxed{GETORIGIN\ ;}$

*OPTION EXTERNAL;*

This call returns either of the following codes (as the value of GETORIGIN):

| Code | Meaning |
|------|---------|
| 1 | Activated by father |
| 2 | Activated by a son |

The condition code is not changed by this intrinsic.

## Determining Father Process

A process can determine the PIN of its father by issuing the FATHER intrinsic call.

*INTEGER PROCEDURE*  $\boxed{FATHER\ ;}$

*OPTION EXTERNAL;*

This call returns the PIN to the calling process (as the value of FATHER).

In addition, one of the following condition codes is returned to the calling process:

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.

## Determining Son Processes

A process can request the return of the PIN assigned to any of its sons, with the GETPROCID intrinsic call:

*INTEGER PROCEDURE*   | *GETPROCID (son)* ; |

*VALUE son;*

*INTEGER son;*

*OPTION EXTERNAL;*

The PIN is returned as the value of GETPROCID.

The *son* parameter designates the specific son in chronological terms—in other words, the *son-th* son still in existence. If *son* exceeds the number of sons currently attached to the calling process, *zero* is returned as the value of GETPROCID.

The condition code is not changed by this intrinsic.


*EXAMPLE:*

*The following command returns the PIN of the sixth existing son of the calling process. (The word VAL contains the integer 6.) The PIN is transmitted to the word PINNO.*

```
PINNO := GETPROCID (VAL);
```

### Determining Process Priority and State

A process can request the return of a two-word message denoting the following information about its father or sons:

| Word | Bits | Meaning |
|------|------|---------|
| 1 | (8:8) | The process' priority number in the master queue. |
| | (0:8) | Not used by MPE/3000; these bits are set to zero by the system. |
| 2 | (15:1) | Activity state. If *on*, the process is active; if *off*, the process is suspended. |
| | (13:2) | Suspension condition (set only if bit 15 is *off*); the *on*-bit indicates the source of the expected activation: |

> Bit 14 = Father
>
> Bit 13 = Son

| Word | Bits | Meaning |
|------|------|---------|
| | (9:4) | Not used by MPE/3000; these bits are set to zero by the system. |
| | (7:2) | The origin of the last ACTIVATE intrinsic call, where 01 = Father and 10 = Son. If the value is *zero*, the process was activated by MPE/3000. |
| | (4:3) | Queue Characteristics: |

> 001 = Master queue.
>
> 111 = Linear subqueue
>
> 100 = Circular subqueue

| Word | Bits | Meaning |
|------|------|---------|
| | (0:4) | Not used by MPE/3000; these bits are set to zero by the system. |

The above information is returned with the GETPROCINFO intrinsic call:

*DOUBLE PROCEDURE*    `GETPROCINFO (pin);`

*VALUE pin;*

*INTEGER pin;*

*OPTION EXTERNAL;*

The double-word information described above is returned as the value of GETPROCINFO.

The parameter is

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

The GETPROCINFO call returns the following condition codes:

CCE    Request granted.

CCG    Request not granted because the process is being terminated.

CCL    Request not granted because an illegal PIN was specified.

*EXAMPLE:*

*To request information about its father, a process issues the following call:*

```
INFO := GETPROCINFO (0);
```

*The information returned to the double-word INFO is*

|  | (Not used) | | | | | | | | Priority | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Word 1:* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| *Word 2:* | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

0 1 2 3   4 5 6   7 8   9 10 11 12   13 14 15

Not used    Queue    (Not used)    Activity State

ACTIVATE Origin

Suspension Conditions
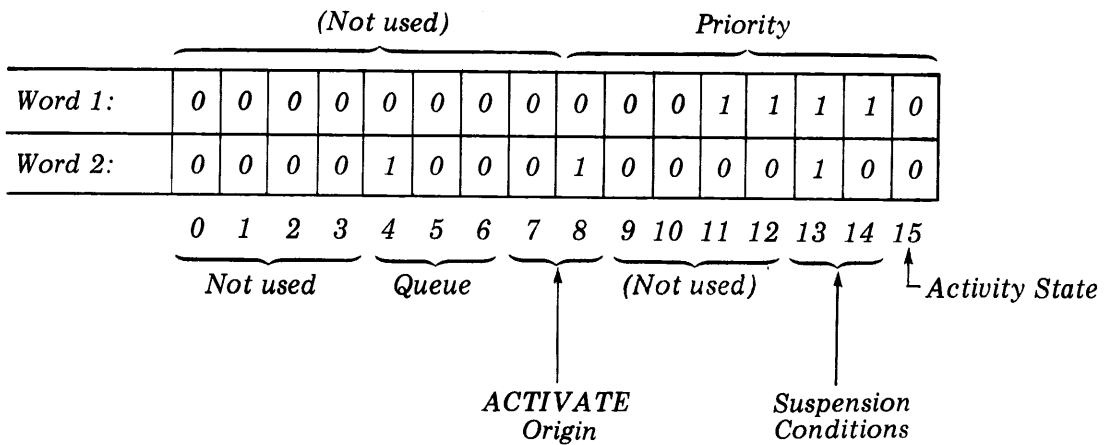
*This information is interpreted as follows, (regarding the* father *process).*

| Word 1: | Bit Nos. | Value | Meaning |
|---|---|---|---|
| | (8:8) | 30 | *Process has priority 32 in master queue.* |
| | (0:8) | 0 | *(Not used.)* |
| Word 2: | (15:1) | 0 | *Process is suspended.* |
| | (14:1) | 0 | *Process to be activated by its son.* |
| | (13:1) | 1 | |
| | (9:4) | 0 | *(Not used.)* |
| | (7:2) | 1 | *Origin of last ACTIVATE call was father of this process.* |
| | (4:3) | 4 | *Circular subqueue.* |

# SECTION XII
# Data-segment Management
# Optional Capability

In Section II, it was noted that each process possesses a private data segment that contains the data generated and manipulated by the process. In a user process, this segment is referred to as the user's stack segment, and can serve many different purposes.

MPE/3000 also allows users with the *Data-Segment Management Optional Capability* to create and access extra data segments for their processes during a job or session. These segments are used for temporary storage while the creating processes exist. Additionally, each segment is assigned an identity that either allows it to be shared between different processes in a job or session, or declares it private to the calling process. When a process terminates, all private data segments created by it are automatically destroyed. Sharable data segments are saved until explicitly deleted or until the job or session ends, at which point they are destroyed. Extra data segments are not directly addressable by user processes. They can only be accessed through intrinsics that move data between the user's stack and the extra data segments by switching the pointer in the DB-Register. If a process not assigned the *Data-Segment Management Capability* attempts to call these intrinsics, that process is aborted.

The maximum number of extra data segments allowed per process is determined at system configuration time, but is never more than four.

## CREATING AN EXTRA DATA SEGMENT

A process can create or acquire an extra data segment by issuing the GETDSEG intrinsic call. (The number of extra data segments that can be requested, and the maximum size allowed these segments, are limited by parameters specified when the system is configured.) When an extra data segment is created, the GETDSEG intrinsic returns to the calling process a *logical index number*, assigned by MPE/3000, that allows this process to reference the segment in later intrinsic calls. The GETDSEG intrinsic is also used to assign the segment the *identity* (noted previously) 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.

The format of the GETDSEG intrinsic call is

PROCEDURE | GETDSEG (index,length,id);

VALUE id;

LOGICAL index,id;

INTEGER length;

OPTION EXTERNAL;

The parameters are

index
A word to which the logical index of the data segment, assigned by MPE/3000, is returned.

length
The size of the data segment (if the segment is not yet created) or the word to which the size of the segment is returned (if the segment already exists).

id
A word containing the identity that declares the data segment sharable between other processes in the job, or private to the calling process. For a sharable segment, id is specified as a non-zero value. (If a data segment with the same id already exists, it is made available to the calling process. Otherwise, a new data segment, sharable within the job/session, is created with this id.) For a private data segment, an id of zero (0) should be specified.

The following condition codes may be returned:

CCE
Request granted; a new segment was created.

CCG
Request granted; an extra data segment with this identity already existed.

CCL
Request denied because an illegal length was specified (INDEX has the value of %2000); the process requested more than the maximum allowable number of data segments (INDEX has the value of %2001), or sufficient storage was not available for the data segment (INDEX has the value of %2002).

*EXAMPLES:*

*To create a new data segment with the identity specified in XSEG and a length of 200 words (specified in the word LNGTH), the user enters the following call. The logical index is returned to the word LI.*

**GETDSEG (LI,LNGTH,XSEG);**

*To determine the logical index and length of an existing extra data segment with the identity specified in ASEG, a process enters this next call. The logical index is returned to the word IND and its length to the word LEN.*

**GETDSEG (IND,LEN,ASEG);**

## DELETING AN EXTRA DATA SEGMENT

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

*PROCEDURE*   | *FREEDSEG (index,id);* |

*VALUE index,id;*

*LOGICAL index,id;*

*OPTION EXTERNAL;*

The intrinsic call parameters are

*index*      A word containing the logical index assigned to the data segment, obtained from the GETDSEG intrinsic call.

*id*         The identity (if any) assigned to the segment. If none is assigned, zero (0) should be entered.

The following condition codes may be returned:

CCE    Request granted; the data segment is deleted from the job.

CCG    Request granted; the data segment is deleted from process but continues
       to exist in the job.

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


*EXAMPLE:*

*To delete a data segment with the logical index contained in INDX, the user enters the following call. Because the segment has no id, it is deleted from both the process and the job.*


**FREEDSEG (INDX,0);**


## TRANSFERRING DATA FROM EXTRA DATA SEGMENT TO STACK

A process can copy a block of words from an extra data segment into the stack by issuing the
DMOVIN intrinsic call. A bounds check is performed by the intrinsic on both the extra data
segment and the stack to ensure that the data is taken from within the data segment boundaries
and moved to an area within the stack boundaries.

*PROCEDURE*      | *DMOVIN (index,disp,number,location);* |

*VALUE index,disp,number;*

*LOGICAL index;*

*INTEGER disp,number;*

*ARRAY location;*

*OPTION EXTERNAL;*

The parameters are

*index*   A word containing the logical index of the extra data segment, obtained from the GETDSEG call.

*disp*   The displacement of the first word in the block to be transferred, from the first word in the data segment. (This must be an integer value greater than or equal to zero.)

*number*  The size of the data block to be transferred, in words. (This must be an integer value greater than or equal to zero.)

*location*  The starting location of the array (buffer) in the stack (relative to the DB address) where the data block is to be moved.

The following condition codes may be returned:

CCE  Request granted.

CCG  Request denied because of bounds-check failure.

CCL  Request denied because of illegal *index* or *number* parameter.


*EXAMPLE:*

*To transfer a block of data 64 words long from an extra data segment whose logical index is specified by LOGX, to the stack, a process can issue to the following call. The data block begins at the tenth word (disp=9) in the data segment. It is entered in the stack beginning at the starting address specified in STDATA.*

**DMOVIN (LOGX,9,64,STDATA);**


## TRANSFERRING DATA FROM STACK TO EXTRA DATA SEGMENT

A process can copy a block of words from the stack to an extra data segment through the DMOVOUT intrinsic call. 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.

PROCEDURE        | DMOVOUT (index,disp,number,location); |

VALUE index,disp,number;

LOGICAL index;

INTEGER disp,number;

ARRAY location;

OPTION EXTERNAL;

The parameters are

index       A word containing the logical index of the extra data segment, obtained
            through the GETDSEG call.

disp        The displacement, in the extra data segment, of the first word of the
            receiving buffer from the first word in the data segment. (This must
            be an integer greater than or equal to zero.)

number      The size of the data block to be transferred, in words. (This must be
            an integer greater than or equal to zero.)

location    The starting address of the data block in the stack (relative to the DB
            address).

The following condition codes can be returned:

    CCE     Request granted.

    CCG     Request denied because of bounds-check failure.

    CCL     Request denied because of illegal *index* or *number* parameter.

EXAMPLE:

*To transfer a block of data 32 words long from the stack to an extra data segment whose
logical index is specified in LOGY, a process can issue the following call. The data block
begins at the address in the stack specified in GODATA. The receiving buffer begins at the
first word in the extra data segment (disp=0).*

**DMOVOUT (LOGY,0,32,GODATA);**

## Changing Size of Data Segment

The user can alter the current size of an extra data segment by calling the ALTDSEG intrinsic. As a typical application, after the user has obtained disc storage for a new segment by calling GETDSEG, he may issue ALTDSEG to reduce the storage required by the segment when it is moved into main memory, and later expand it as needed, for more efficient use of memory. (In no case, can ALTDSEG be used to increase segment size beyond that originally assigned through GETDSEG.)  The ALTDSEG intrinsic format is

*PROCEDURE*   | *ALTDSEG (index,inc,size);* |

*VALUE index, inc;*

*LOGICAL index;*

*INTEGER inc,size;*

*OPTION EXTERNAL;*

The parameters are

*index*      A word containing the logical index of the extra data segment, obtained through the GETDSEG call.

*inc*        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 denotes a decrease.

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

The following condition codes can be returned:

CCE     Request granted.

CCG     Request not fully granted; an illegal decrement, requesting a new total segment size of 0 or less, or an illegal increment, requesting a new size greater than the size originally assigned by GETDSEG, was attempted.  In the first case, the current size remains in effect; in the second case, the original size is granted (and this size is returned through the *size* parameter).

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

# SECTION XIII
# Multiple Resource Identification Number
# Optional Capability

As discussed in Section IX, users can manage the sharing of resources (such as input/output devices, files, programs, or procedures) between jobs or processes through resource identification numbers (RIN's). Users having only *Standard MPE/3000 Capabilities* can lock only one global or local RIN at a time. However, users having the *Multiple RIN Optional Capability* can lock as many global RIN's as desired simultaneously, with no checking by MPE/3000. (This applies to the FLOCK intrinsic for files as well as the LOCKGLORIN intrinsic for global RIN's.) Under multiprogramming, this capability introduces a risk that deadlocking may occur unless the users cooperating in resource management are very careful. In deadlocking, two jobs or processes that have been suspended cannot be resumed because they are mutually blocked. The following example illustrates how deadlocking can occur.

*EXAMPLE:*

*In this example, Process A and Process B, running concurrently, are managing two RINs, 30 and 31. At one point, Process A locks RIN 30 (designated by R30). Subsequently, Process B locks RIN 31 (designated by R31). Still later, Process A tries to lock RIN 31 while it is still locked by Process B. Thus, Process A is suspended. Process B then tries to lock RIN 30, still locked by Process A. As a result, Process B is also suspended. Since Process A has RIN 30 locked and needs RIN 31, and Process B has RIN 31 locked and needs RIN 30, both processes are blocked and cannot become unblocked; they are deadlocked.*

```
Running
 Time            Process A                          Process B
                    .                                  .
                    .                                  .
                    .                                  .
                LOCKGLORIN (R30,TRU)                   .
                    .                                  .
                    .                              LOCKGLORIN (R31,TRU)
                    .                                  .
                LOCKGLORIN (R31,TRU)                   .
                                                       .
                Since RIN 31 is locked,                .
                Process A is suspended.            LOCKGLORIN (R30,TRU)

                                                   Since RIN 30 is locked,
                                                   Process B is suspended.

                        ───────── DEADLOCK ─────────
```
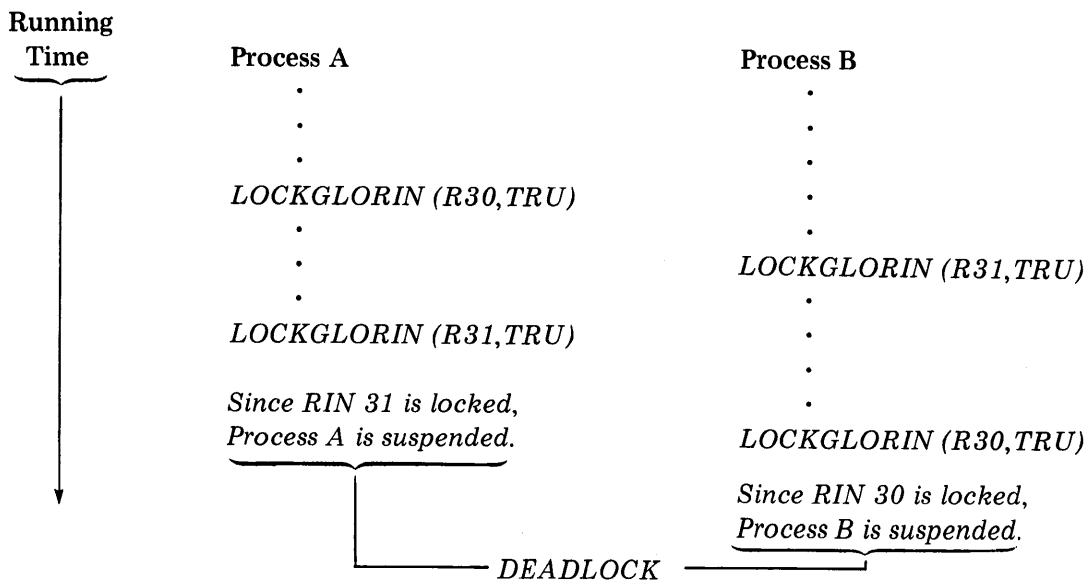
As a guide in avoiding deadlocks, users preparing jobs or processes that cooperate in RIN management can logically pre-order their use of RINs in the same way.   For example, Job A and Job B can both issue requests to lock RINs in ascending numerical order (RIN's 10, 11, 12, and so forth). This ensures that neither job ever attempts to lock a RIN of lower logical rank than any RIN currently locked, and a deadlock never occurs.  The RIN's can be released in any order.

# SECTION XIV
## Privileged Mode Optional Capability

Normally, an MPE/3000 user can access only his own code and data areas in main memory. But a user with the *Privileged Mode Optional Capability* can access all areas of the system and can use all features of the hardware. This user can access all normally-uncallable intrinsics and system tables. He can invoke all system instructions, including those in the privileged central processor instruction set. He can, in short, use the computer on the same terms as MPE/3000 itself.

> CAUTION: No hardware mechanism is provided to prevent a program running in privileged mode from damaging the environments of other users or destroying MPE/3000 itself. For such a program, the only bounds-checking performed is that for stack overflow (where the S register contains an address greater than that in the Z register). Thus, a privileged-mode program constitutes a potential hazard to the system. When preparing and running such programs, users should be very cautious, particularly regarding the locations referenced by the program.

A programmer with the *Privileged Mode Optional Capability* can use this capability in two ways:

1. He can write permanently privileged programs that are loaded and executed entirely in privileged mode.

2. He can write temporarily privileged programs that dynamically enter and leave privileged mode during execution, as required.

## PERMANENTLY PRIVILEGED PROGRAMS

A program's segments are loaded and executed directly in privileged mode when all three of the following conditions exist:

1. Any of the program's code segments contains privileged instructions.

2. The program is prepared with the *Privileged Mode Optional Capability*, by entering the appropriate capability-class attribute in the *caplist* parameter of the :PREP or :PREPRUN command that prepares the program. This implies that the *user* issuing this command must himself have been assigned the *Privileged Mode Optional Capability*.

3. The NOPRIV optional parameter is omitted from the :PREPRUN or :RUN command that executes the program, or the CREATE intrinsic that creates a process to run it. (This omission turns the privileged mode bit in the appropriate CST entries *on.*)

When a user adds a segment to a Segmented Library (through the –ADDSL Segmenter Command), the procedures within the segment are checked to determine if any one of them is privileged. If it is, the segment is always run in privileged mode. (In order to add a segment containing one or more privileged procedures to a library, the user must himself possess the *Privileged-Mode Optional Capability.*)

## TEMPORARILY PRIVILEGED PROGRAMS

Temporarily privileged programs are initiated, upon request, in the non-privileged mode. Then, intrinsics can be issued to change the program to and from the privileged mode dynamically. For example, just before a set of privileged instructions is encountered, the program can be switched to the privileged mode to allow execution of these instructions. Then, after the last privileged instruction in the set is encountered, the program can be returned to non-privileged mode. This bracketing of privileged instructions aids in reducing system violations, since the program cannot access locations or resources outside the user's environment when it is running in non-privileged mode.

The user running a temporarily-privileged program should understand how the central processor handles procedure calls (PCAL instructions) and exits (EXIT instructions) when encountered in either mode:

*In the privileged mode,* when a PCAL instruction is executed, privileged mode is retained even though the destination code segment may have a non-privileged CST entry. When an EXIT instruction is encountered, the resulting mode depends upon the status word in the stack marker.

*In the non-privileged mode*, when a PCAL instruction is encountered, the mode assumed is obtained from the CST entry for the destination code segment. When an EXIT instruction occurs, the resulting mode is taken from the CST entry indicated in the stack marker. If this entry indicates privileged mode, a system violation occurs.

In general, the status word determines the action taken in privileged mode, but the CST entry determines the action in non-privileged mode.

A program is loaded and begins execution as a temporarily privileged program (in the non-privileged mode) when these two conditions are met:

1. The program is prepared with the *Privileged Mode Optional Capability*, by entering the appropriate capability-class attribute in the *caplist* parameter of the :PREP or :PREPRUN command that prepares the program. This implies that the user issuing this command must himself have been assigned the *Privileged Mode Optional Capability*.

2. The NOPRIV optional parameter is *included* in the :PREPRUN or :RUN command that executes the program, or the CREATE intrinsic that creates a process to run it.

When a temporarily-privileged program is initiated, the CST entries corresponding to its segments have their privileged-mode bits set *off*.

The intrinsics for dynamically switching modes are described below.

## ENTERING PRIVILEGED MODE

To switch a temporarily-privileged program from the non-privileged mode to the privileged mode, the GETPRIVMODE intrinsic call is issued. This intrinsic turns the privileged mode bit in the status register *on*, but leaves the privileged mode bit in the CST entry for the executing segment *off*. Thus, if additional segments are to be run as part of the program, they will be run in privileged mode unless an intrinsic is specifically called to return to the non-privileged mode, because the status register rather than the CST determines a mode change when running in privileged mode. The GETPRIVMODE call is written simply as

*PROCEDURE*     | *GETPRIVMODE;* |

*OPTION EXTERNAL;*

14-3

The following condition codes may be returned:

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.)

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


## ENTERING NON-PRIVILEGED MODE

To change a temporarily-privileged program from the privileged to the non-privileged mode, the GETUSERMODE intrinsic call is issued. This intrinsic changes the privileged mode bit in the status register to *off*. The intrinsic call is written simply as

*PROCEDURE*    GETUSERMODE;

*OPTION EXTERNAL;*

The following condition codes may be returned:

CCE    Request granted; the program was in privileged mode when the intrinsic call was issued.

CCG    Request granted; the program was already in non-privileged mode when the intrinsic call was issued.

CCL    (Not returned by this intrinsic.)


## MOVING THE DB-POINTER

When a user with the *Data Segment Management Optional Capability* runs a process with an extra data segment in privileged mode, he can prepare for movement of data between this segment and the stack by calling 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. This intrinsic returns the logical index of the data segment indicated by the previous DB register setting, allowing the user to restore this setting later. (If the previous DB setting indicated the stack, *zero* is returned.)

14-4

*LOGICAL PROCEDURE*　　　　　 | SWITCHDB (index); |

*VALUE index;*

*LOGICAL index;*

*OPTION PRIVILEGED, EXTERNAL;*

The previous DB-register setting is returned as the value of SWITCHDB.

The *index* parameter denotes the logical index of the data segment to which the DB register is switched, as obtained through the GETDSEG intrinsic call. (MPE/3000 checks the value specified for *index*, to ensure that the process has previously acquired access to this segment.) For an extra data segment, the *index* must be a positive, non-zero integer. To switch back to the stack segment, the *index* must be zero.

The following condition codes can be returned by the SWITCHDB intrinsic:

CCE　　　Request granted.

CCG　　　(Not returned by this intrinsic.)

CCL　　　Request rejected, because an illegal data segment index was referenced.

The calling process is aborted if the user does not have the *Privileged Mode Optional Capability.*


*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, this intrinsic call is issued. (The previous DB-register setting is returned to the word SET.)*

**SET := SWITCHDB (INDEX2);**

## OTHER DATA-SEGMENT INTRINSICS

Users with the *Privileged-Mode Optional Capability* can also call all intrinsics available to users with the *Data Segment Management Optional Capability* (Section XII), provided that they acknowledge these rules:

1.  When calling the data segment intrinsics from *privileged mode*, ensure that the DB register points to its normal stack position. When GETDSEG is used to create extra data segments under these conditions, the number of segments that can be created is limited only by the space available in the process Control Block Extension; this is virtually unlimited.

2.  When a termporarily-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. But, if the process calls a data segment intrinsic in *privileged* mode, the index returned *cannot* be used to reference the segment in *non-privileged* mode.

## SCHEDULING PROCESSES

A user with the *Privileged Mode Optional Capability* can schedule processes in areas of the master scheduling queue that are not available to other users.

The master queue (Figure 14-1) is divided into logical areas, each corresponding to a particular type of dispatching and priority class for the processes within it. A logical area can be a linear subqueue, a circular subqueue, or a portion of the main master queue. In a linear subqueue, the process with highest priority accesses the central processor first and maintains this access until the process either is completed, terminated, or suspended to await the availability of a required resource. In a circular subqueue, all processes have equal priority and each accesses the central processor for an interval (time quantum) of maximum duration (or until completed, terminated, or suspended). At the end of this duration, control is transferred to the next process in the queue, continuing in a round-robin fashion. This time-slicing is controlled by the system timer. Processes that are not scheduled in a subqueue are scheduled in the master queue.

Each subqueue in the master queue is defined by a priority number. While the standard user is aware of the priority class associated with a subqueue, only a user with *Privileged Mode Capability* or a System Supervisor User is concerned with priority numbers. The standard subqueues (priority classes) are as follows; additional subqueues can be added at system generation time.

AS                                    is a linear subqueue containing processes of very high priority. (In a subsequent MPE/3000 release, it will be accessible only to users with a special *MPE/3000 Optional Capability* to be added to the system then.) Its priority number is 30.
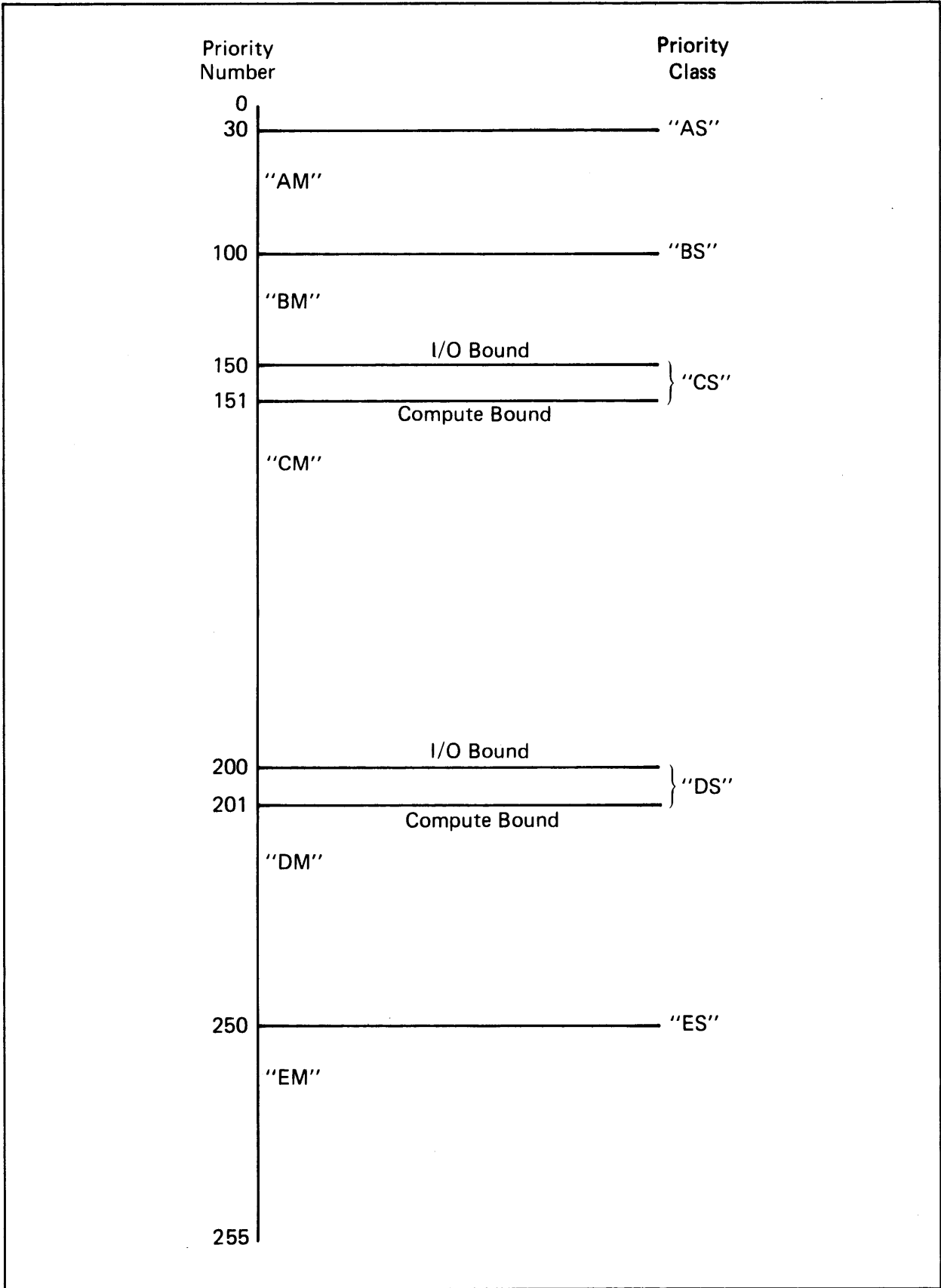
Figure 14-1. MPE/3000 Master Queue Structure

BS          is a linear subqueue containing processes of high priority. (In a subsequent MPE/3000 release, it will be accessible only to users having a special *MPE/3000 Optional Capability* to be added to the system.) Its priority number is 100.

CS          is composed of two circular sub-subqueues, each having a time-quantum determined at system generation time. This subqueue is used mainly for interactive processes and for multiprogramming batch jobs. The time quantum specifies the maximum execution time allowed any process in the sub-subqueue before going on to the next process. The priority numbers of the sub-subqueues are 150 and 151; the higher-priority sub-subqueue is used for input/output-bound processes, and the lower-priority sub-subqueue is used for compute-bound processes.

DS          is also composed of two circular sub-subqueues, each having a time-quantum determined at configuration time. This subqueue is available for general use at a lower priority than the CS subqueue. The priority numbers of the sub-subqueues are 200 and 201; as in the CS subqueue, one sub-subqueue is used for input/output-bound processes and the other is used for compute-bound processes.

ES          is a linear subqueue, primarily containing idle processes with low priority. Its priority number is 250.

AM,BM,CM,DM,EM          are discontiguous portions of the main master queue. Processes can be scheduled in these areas only if they are system processes or are initiated by a user with Privileged Mode or System Supervisor Capability. Such processes are dispatched linearly.

The priority class of a process can be specified by the normal user with standard or optional MPE/3000 capabilities. In the two-character string that comprises a priority-class reference, the first character refers to the location of a subqueue within the master queue (in alphabetical order) and the second character specifies whether the logical area is the subqueue itself (S) or the portion of the master queue (M) that immediately follows the subqueue. When a priority class is requested for a process, it is assigned the lowest priority within that class (relative to other processes already assigned the same class). In a circular subqueue, the actual priority of the process is modified as other processes use central processor time. In a linear subqueue, an automatic adjustment is made as necessary.

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 scheduled in the same subqueue or portion of the master queue. The privileged mode user also can specify the relative ranks of processes having identical numbers within a linear subqueue, and can assign them specific priorities in the master queue.

With each circular subqueue of priority P, can be associated another subqueue of priority P-1 called the input/output subqueue. (As noted above, these are the *subqueues* within the CS and DS classifications.) If any process is detected as being input/output bound, the system

switches it from the normal (compute-bound) subqueue to the input/output subqueue. (An input/output bound process is defined as one which, for a specified number of consecutive times, has been suspended for input/output instead of being timed out at the end of its time-quantum.) This gives the process a higher priority until it is no longer input/output bound. A process is considered to be compute-bound when, for a specified number of consecutive times, it reaches the end of its time-quantum. Users cannot directly access input/output bound subqueues — processes are always moved to and from these subqueues by the system to improve response time for input/output interactive processes. The time quantum on each input/output-bound subqueue is also dynamically-adjusted to regulate the load on each queue.

Priorities are assigned to processes through the *priorityclass* parameter of the CREATE and/or GETPRIORITY intrinsic (Section XI). Because users with the Privileged Mode Capability can schedule processes within the master queue, the *priorityclass* parameter can take on the following values:

1. A string of two ASCII characters describing the standard priority-class (subqueue) in which the process is to be scheduled; the standard subqueues are "AS", "BS", "CS", "DS", or "ES".

2. An ASCII character ($x$) specifying a valid subqueue, followed immediately by the single-character string "M", indicating the Master Queue. The word format is:

    | Bits | 0         7 | 8       15 |
    |------|-------|-------|
    |      | $x$   | "M"   |
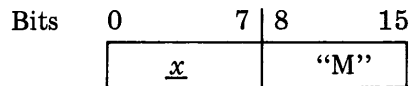
    This schedules the process in that region of the master queue directly adjacent to and below the subqueue $x$. The process is scheduled in the first available priority in that region.

3. An ASCII character ($y$) specifying an absolute priority number, followed by the single-character string "A" indicating that $y$ is an absolute priority number. The word format is:

    | Bits | 0         7 | 8       15 |
    |------|-------|-------|
    |      | $y$   | "A"   |
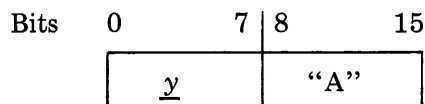
    This schedules the process at the location specified by $y$ in the Master Queue. If another process or subqueue already occupies the location designated by $y$, the process is placed in the first location available below $y$, and the process' priority number is changed to reflect that location; a warning message is then issued to the standard list device.

# PART 4
## Appendices

# APPENDIX A
# ASCII Character Set

| Character | Octal Value (Left Byte) | Octal Value (Right Byte) | Meaning |
|-----------|-------------------------|--------------------------|---------|
| NUL | 000000 | 000000 | Null (Normally ignored by MPE/3000) |
| SOH | 000400 | 000001 | Start of heading |
| STX | 001000 | 000002 | Start of text |
| ETX | 001400 | 000003 | End of text |
| EOT | 002000 | 000004 | End of transmission |
| ENQ | 002400 | 000005 | Enquiry |
| ACK | 003000 | 000006 | Acknowledge |
| BEL | 003400 | 000007 | Bell |
| BS | 004000 | 000010 | Backspace |
| HT | 004400 | 000011 | Horizontal tabulation |
| LF | 005000 | 000012 | Line feed (Normally ignored by MPE/3000) |
| VT | 005400 | 000013 | Vertical tabulation |
| FF | 006000 | 000014 | Form feed |
| CR | 006400 | 000015 | Carriage return |
| SO | 007000 | 000016 | Shift out |
| SI | 007400 | 000017 | Shift in |
| DLE | 010000 | 000020 | Data link escape |
| DC1 | 010400 | 000021 | Device control 1 ⎫ |
| DC2 | 011000 | 000022 | Device control 2 ⎬ (Normally ignored by MPE/3000) |
| DC3 | 011400 | 000023 | Device control 3 ⎪ |
| DC4 | 012000 | 000024 | Device control 4 ⎭ |
| NAK | 012400 | 000025 | Negative acknowledge |
| SYN | 013000 | 000026 | Synchronous idle |
| ETB | 013400 | 000027 | End of transmission block |

| Character | Octal Value (Left Byte) | Octal Value (Right Byte) | Meaning |
|---|---|---|---|
| CAN | 014000 | 000030 | Cancel |
| EM | 014400 | 000031 | End of medium |
| SUB | 015000 | 000032 | Substitute |
| ESC | 015400 | 000033 | Escape |
| FS | 016000 | 000034 | File separator |
| GS | 016400 | 000035 | Group separator |
| RS | 017000 | 000036 | Record separator |
| US | 017400 | 000037 | Unit separator |
| SP | 020000 | 000040 | Space |
| ! | 020400 | 000041 | Exclamation point |
| " | 021000 | 000042 | Quotation mark |
| # | 021400 | 000043 | Number sign |
| $ | 022000 | 000044 | Dollar sign |
| % | 022400 | 000045 | Percent sign |
| & | 023000 | 000046 | Ampersand |
| ' | 023400 | 000047 | Apostrophe |
| ( | 024000 | 000050 | Opening parenthesis |
| ) | 024400 | 000051 | Closing parenthesis |
| * | 025000 | 000052 | Asterisk |
| + | 025400 | 000053 | Plus |
| , | 026000 | 000054 | Comma |
| – | 026400 | 000055 | Hyphen (Minus) |
| . | 027000 | 000056 | Period (Decimal) |
| / | 027400 | 000057 | Slant |
| 0 | 030000 | 000060 | Zero |
| 1 | 030400 | 000061 | One |
| 2 | 031000 | 000062 | Two |
| 3 | 031400 | 000063 | Three |
| 4 | 032000 | 000064 | Four |
| 5 | 032400 | 000065 | Five |
| 6 | 033000 | 000066 | Six |
| 7 | 033400 | 000067 | Seven |

| Character | Octal Value (Left Byte) | Octal Value (Right Byte) | Meaning |
|---|---|---|---|
| 8 | 034000 | 000070 | Eight |
| 9 | 034400 | 000071 | Nine |
| : | 035000 | 000072 | Colon |
| ; | 035400 | 000073 | Semi-colon |
| < | 036000 | 000074 | Less than |
| = | 036400 | 000075 | Equals |
| > | 037000 | 000076 | Greater than |
| ? | 037400 | 000077 | Question mark |
| @ | 040000 | 000100 | Commercial at |
| A | 040400 | 000101 | Uppercase A |
| B | 041000 | 000102 | Uppercase B |
| C | 041400 | 000103 | Uppercase C |
| D | 042000 | 000104 | Uppercase D |
| E | 042400 | 000105 | Uppercase E |
| F | 043000 | 000106 | Uppercase F |
| G | 043400 | 000107 | Uppercase G |
| H | 044000 | 000110 | Uppercase H |
| I | 044400 | 000111 | Uppercase I |
| J | 045000 | 000112 | Uppercase J |
| K | 045400 | 000113 | Uppercase K |
| L | 046000 | 000114 | Uppercase L |
| M | 046400 | 000115 | Uppercase M |
| N | 047000 | 000116 | Uppercase N |
| O | 047400 | 000117 | Uppercase O |
| P | 050000 | 000120 | Uppercase P |
| Q | 050400 | 000121 | Uppercase Q |
| R | 051000 | 000122 | Uppercase R |
| S | 051400 | 000123 | Uppercase S |
| T | 052000 | 000124 | Uppercase T |
| U | 052400 | 000125 | Uppercase U |
| V | 053000 | 000126 | Uppercase V |
| W | 053400 | 000127 | Uppercase W |

| Character | Octal Value (Left Byte) | Octal Value (Right Byte) | Meaning |
|---|---|---|---|
| X | 054000 | 000130 | Uppercase X |
| Y | 054400 | 000131 | Uppercase Y |
| Z | 055000 | 000132 | Uppercase Z |
| [ | 055400 | 000133 | Opening bracket |
| \ | 056000 | 000134 | Reverse slant |
| ] | 056400 | 000135 | Closing bracket |
| ^ | 057000 | 000136 | Circumflex |
| _ | 057400 | 000137 | Underscore |
| ` | 060000 | 000140 | Grave accent |
| a | 060400 | 000141 | Lowercase a |
| b | 061000 | 000142 | Lowercase b |
| c | 061400 | 000143 | Lowercase c |
| d | 062000 | 000144 | Lowercase d |
| e | 062400 | 000145 | Lowercase e |
| f | 063000 | 000146 | Lowercase f |
| g | 063400 | 000147 | Lowercase g |
| h | 064000 | 000150 | Lowercase h |
| i | 064400 | 000151 | Lowercase i |
| j | 065000 | 000152 | Lowercase j |
| k | 065400 | 000153 | Lowercase k |
| l | 066000 | 000154 | Lowercase l |
| m | 066400 | 000155 | Lowercase m |
| n | 067000 | 000156 | Lowercase n |
| o | 067400 | 000157 | Lowercase o |
| p | 070000 | 000160 | Lowercase p |
| q | 070400 | 000161 | Lowercase q |
| r | 071000 | 000162 | Lowercase r |
| s | 071400 | 000163 | Lowercase s |
| t | 072000 | 000164 | Lowercase t |
| u | 072400 | 000165 | Lowercase u |
| v | 073000 | 000166 | Lowercase v |
| w | 073400 | 000167 | Lowercase w |

| Character | Octal Value (Left Byte) | Octal Value (Right Byte) | Meaning |
|-----------|-------------------------|--------------------------|---------|
| x | 074000 | 000170 | Lowercase x |
| y | 074400 | 000171 | Lowercase y |
| z | 075000 | 000172 | Lowercase z |
| { | 075400 | 000173 | Opening (left) brace |
| \| | 076000 | 000174 | Vertical line |
| } | 076400 | 000175 | Closing (right) brace |
| ∿ | 077000 | 000176 | Tilde |
| DEL | 077400 | 000177 | Delete |

# APPENDIX B
# Summary of Commands

All MPE/3000 commands are summarized below, grouped alphabetically. Following these commands, the MPE/3000 Segmenter commands are also summarized. In both summaries, the last column denotes when the command can be issued: during a batch job (B), during a time-sharing session (S), during a break (B), or programmatically (through the COMMAND intrinsic) (P).

## STANDARD CAPABILITY COMMANDS

| Command | Parameters | Function | When Issued | Text Discussion |
|---------|-----------|----------|-------------|-----------------|
| :ABORT | | Aborts the current program. | B | 5-49 |
| :ALTSEC | filereference [;([modelist: userlist[;...]])] | Changes security provisions for a file. | J,S,B,P | 3-22 |
| :BASIC | [commandfile] [,[inputfile] [,listfile]] | Calls BASIC/3000 interpreter. | J,S | 4-5 |
| :BUILD | filereference [;DEV=device] [;DISC=[filesize] [,[numextents] [,initialloc]]] [;REC=[recsize] [,[blockfactor] $\left[ , \begin{bmatrix} F \\ U \\ V \end{bmatrix} \begin{bmatrix} ,BINARY \\ ,ASCII \end{bmatrix} \right]$]] [;CCTL] [;TEMP] [;CODE=filecode] | Creates a new file. | J,S,B,P | 5-28 |
| :BYE | | Terminates a session. | S | 3-22 |
| :COBOL | [textfile] [,[uslfile] [,[listfile] [,[masterfile] [,newfile]]]] | Compiles a COBOL/3000 program. | J,S | 4-7 |

B-1

| Command | Parameters | Function | When Issued | Text Discussion |
|---------|-----------|----------|-------------|-----------------|
| :COBOLGO | [textfile] [,[listfile] [,[masterfile] [,newfile] ] ] | Compiles, prepares, and executes a COBOL/3000 program. | J,S | 4-11 |
| :COBOLPREP | [textfile] [,[progfile] [,[listfile] [,[masterfile] [,newfile] ] ] ] | Compiles and prepares a COBOL/3000 program. | J,S | 4-9 |
| :CONTINUE | | Disregards job-error condition. | J | 3-25 |
| :DATA | [jobname,] username[/upass] .acctname[/apass] [;filename] | Defines data from outside standard input stream. Cannot be read on $STDINX file. Acceptable for device recognition. | J,S | 3-24 |
| :EDITOR | [listfile] | Calls the EDITOR. | J,S | 4-22 |
| :EOD | | Denotes end of data. Cannot be read on $STDINX file. | J,S | 3-15 |
| :EOJ | | Denotes end of batch job. Cannot be read on $STDINX file. | J | 3-15 |
| :FILE | (This command has many parameters that can be combined in various different formats; the user is referred to Section V.) | Defines or redefines a file's characteristics. | J,S,B,P | 5-11 |
| :FORTGO | [textfile] [,[listfile] [,[masterfile] [,newfile] ] ] | Compiles, prepares, and executes a FORTRAN/3000 program. | J,S | 4-11 |
| :FORTPREP | [textfile] [,[progfile] [,[listfile] [,[masterfile] [,newfile] ] ] ] | Compiles and prepares FORTRAN/3000 program. | J,S | 4-9 |
| :FORTRAN | [textfile] [,[uslfile] [,[listfile] [,[masterfile] [,newfile] ] ] ] | Compiles a FORTRAN program. | J,S | 4-7 |
| :FREERIN | rin | Deallocates a global RIN, and returns it to RIN pool. | J,S | 9-6 |
| :GETRIN | [rinpassword] | Acquires a global RIN. | J,S,B,P | 9-2 |
| :HELLO | [sessionname,] username[/upass] .acctname[/apass] [,group[/gpass] ] [;TERM=termtype] [;TIME=cputime] [;PRI=executionpriority] [;INPRI=selectionpriority] | Initiates a session. Acceptable for device recognition. Requires IA capability class. | S | 3-18 |

| Command | Parameters | Function | When Issued | Text Discussion |
|---|---|---|---|---|
| :JOB | [jobname,] username[/upass] .acctname[/apass] [,groupname [/gpass] ] [;TERM=termtype] [;TIME=cputime] [;PRI=executionpriority] [;INPRI=selectionpriority] [;OUTCALSS=outputclass] | Initiates a batch job. Cannot be read on $STDINX. Acceptable for device recognition. Requires BA capability class. | J,S | 3-11 |
| :LISTF | [fileset] [,detail] [;listfile] | Lists descriptions of files. | J,S,B,P | 5-30 |
| :PREP | uslfile,progfile [;ZERODB] [;PMAP] [;MAXDATA=segsize] [;STACK=stacksize] [;DL=dlsize] [;CAP=caplist] [;RL=filename] | Prepares a compiled program into segmented form. | J,S | 4-12 |
| :PREPRUN | uslfile [,entrypoint] [;NOPRIV] [;PMAP] [;LMAP] [;ZERODB] [;MAXDATA=segsize] [;PARM=parameternum] [;STACK=stacksize] [;DL=dlsize] [;LIB=library] [;CAP=caplist] [;RL=filename] | Prepares and executes a program. | J,S | 4-17 |
| :PTAPE | [filename] | Reads a paper tape without X-OFF control. | S,B,P | 8-69 |
| :PURGE | filereference[,TEMP] | Deletes a file from the system. | J,S,B,P | 5-30 |
| :RELEASE | filereference | Releases the security provisions of a file. | J,S,B,P | 5-51 |
| :RENAME | oldfilereference, newfilereference [ ,TEMP] | Renames a file. | J,S,B,P | 5-42 |
| :REPORT | | Displays total accounting information for a log-on group. | J,S,B,P | 8-2 |
| :RESET | { @ / formaldesignator } | Resets a formal file designator. | J,S,B,P | 5-27 |
| :RESTORE | tapefile[;[filesetlist] [ [;KEEP] [;DEV=device] [;SHOW] ] ] | Restores a complete fileset, stored off-line. | J,S,B,P | 5-38 |

B-3

| Command | Parameters | Function | When Issued | Text Discussion |
|---|---|---|---|---|
| :RESUME | | Resumes an interrupted program. | B | 3-22,8-41 |
| :RUN | progfile[,entrypoint] [;NOPRIV] [;LMAP] [;MAXDATA=segsize] [;PARM=parameternum] [;STACK=stacksize] [;DL=dlsize] [;LIB=library] | Loads and executes a program. | J,S | 4-21 |
| :SAVE | { [$OLDPASS,newfilereference] [tempfilereference] } | Changes a file to permanent status. Requires SF capability for saving files. | J,S,B,P | 5-29 |
| :SECURE | filereference | Restores suspended security provisions for a file. | J,S,B,P | 5-52 |
| :SEGMENTER | [listfile] | Calls MPE/3000 Segmenter. | J,S | 4-24,7-2 |
| :SHOWJOB | [ # jsnumber jsname ] | Displays job/session status. | J,S,B,P | 8-2 |
| :SHOWTIME | | Displays current date and time-of-day. | J,S,B,P | 8-2 |
| :SPEED | { [inspeed] ,outspeed inspeed } | Changes input speed or output speed of terminal. | S,B,P | 8-60 |
| :SPL | [sourcefile] [,[uslfile] [,[listfile] [,[masterfile] [,newfile] ] ] ] | Compiles an SPL/3000 program. | J,S | 4-7 |
| :SPLGO | [sourcefile] [,[listfile] [,[masterfile] [,newfile] ] ] | Compiles, prepares, and executes an SPL/3000 program. | J,S | 4-11 |
| :SPLPREP | [sourcefile] [,[progfile] [,[listfile] [,[masterfile] [,newfile] ] ] ] | Compiles and prepares an SPL/3000 program. | J,S | 4-9 |
| :STAR | [listfile] [,NOLIST] | Calls the Statistical Analysis Routines. | J,S | 4-23 |
| :STORE | [filesetlist] ;tapefile [;SHOW] | Stores a set of files off-line. | J,S,B,P | 5-33 |
| :TELL | [jsname,] username. acctname;message | Transmits a message. | J,S,B,P | 8-5 |
| :TELLOP | message | Transmits a message from the user to the computer operator. | J,S,B,P | 8-6 |

B-4

## STANDARD CAPABILITY COMMANDS (SEGMENTER COMMANDS)

| Command | Parameters | Function | When Issued | Text Discussion |
|---------|------------|----------|-------------|-----------------|
| -ADDRL | name[(index)] | Adds a procedure to an RL. | J,S | 7-19 |
| -ADDSL | name[,PMAP] | Adds a segment to an SL. | J,S | 7-24 |
| -AUXUSL | filereference | Designates a source of RBM input for -COPY command. | J,S | 7-9 |
| -BUILDRL | filereference,records,extents | Creates a permanent, formatted RL file. | J,S | 7-18 |
| -BUILDSL | filereference,records,extents | Creates a permanent, formatted SL file. | J,S | 7-23 |
| -BUILDUSL | filereference,records,extents | Creates a temporary, formatted USL file. | J,S | 7-3 |
| -CEASE | [pointspec,] name[(index)] | Deactivates one or more entry-points in a USL. | J,S | 7-6 |
| -COPY | [rbmspec,] name[(index)] | Copies an RBM or segment from one USL to another. | J,S | 7-9 |
| -EXIT | | Exits from Segmenter, returning control to MPE/3000 Command Interpreter. | J,S | 7-2 |
| -HIDE | name[(index)] | Sets an RBM internal flag on. | J,S | 7-27 |
| -LISTRL | | Lists the procedures in an RL. | J,S | 7-20 |
| -LISTSL | | Lists the segments in an SL. | J,S | 7-24 |
| -LISTUSL | | Lists the RBM's in a USL. | J,S | 7-10 |
| -NEWSEG | newsegname,rbmname [(index)] | Changes the segment name of an RBM. | J,S | 7-8 |
| -PREPARE | progfile [;ZERODB] [;PMAP] [;MAXDATA=segsize] [;STACK=stacksize] [;DL=dlsize] [;CAP=caplist] [;RL=filename] | Prepares RBM's from a USL into a program file. | J,S | 7-12 |
| -PURGERBM | [rbmspec,] name[(index)] | Deletes one or more RBM's from a USL. | J,S | 7-7 |

| Command | Parameters | Function | When Issued | Text Discussion |
|---------|-----------|----------|-------------|-----------------|
| -PURGERL | [rbmspec,] name | Deletes an entrypoint or a procedure from an RL. | J,S | 7-19 |
| -PURGESL | [unitspec,] name | Deletes an entrypoint or a segment from an SL. | J,S | 7-24 |
| -REVEAL | name[(index)] | Sets an RBM internal flag off. | J,S | 7-27 |
| -RL | filereference | Designates an RL for management. | J,S | 7-19 |
| -SL | filereference | Designates an SL for management. | J,S | 7-23 |
| -USE | [pointspec,] name[(index)] | Activates one or more RBM entry-points. | J,S | 7-5 |
| -USL | filereference | Designates a USL for management. | J,S | 7-3 |

# APPENDIX C
# *Summary of Intrinsic Calls*

All intrinsic calls available to the user are summarized below, listed alphabetically. For each intrinsic, the complete declaration head appears; the intrinsic call format is distinguished from the remainder of the head by a box. The function of the intrinsic is described. For those intrinsics that are type procedures, the procedure *type* is noted. Optional parameters are **bold face** in the intrinsic head and are also noted separately. All condition codes that can be returned by the intrinsic are listed. The intrinsic error-code number is also presented. (This is the number that appears in the abort-error message generated when an error is encountered in the corresponding intrinsic.) The functional categories represented by the error number set are

| Error Number Range | Functional Category |
|---|---|
| 1 — 29 | File Management |
| 30 — 39 | Resource Management |
| 40 — 49 | System Timer (Clock) |
| 50 — 59 | Traps |
| 60 — 79 | Utility Routines |
| 80 — 99 | Program Management |
| 100 — 119 | Process Control |
| 120 — 129 | Scheduling |
| 130 — 149 | Data Segments |
| 180 — 199 | Input/Output Utilities |
| 200 — 209 | Special Utilities |

Where intrinsics have special attributes, those attributes are noted as follows:

- Uncallable intrinsics (those that can only be invoked by the user in privileged mode) are indicated by an asterisk (*).

- Intrinsics that can be called in privileged mode even though the user does not have the appropriate capability are denoted by a plus sign (+).

- Intrinsics that can be called by a process when the DB register is not pointing to that process' stack are denoted by a dollar sign ($).

The MPE/3000 Optional Capability (if any) required to invoke the intrinsic, and the page in the main text where the intrinsic is discussed, are also noted.

PROCEDURE | ACTIVATE (pin, susp);

VALUE              pin, susp;

LOGICAL            susp;

INTEGER            pin;

OPTION VARIABLE, EXTERNAL;


FUNCTION:                    Activates a process.

TYPE:                        None

OPTIONAL PARAMETERS:         susp

CONDITION CODES:             CCE, CCG, CCL

ERROR CODE:                  104

SPECIAL ATTRIBUTES:          $

OPTIONAL CAPABILITY:         Process Handling

TEXT DISCUSSION:             11-11

*INTEGER PROCEDURE*

| *ADJUSTUSLF (uslfnum, records);* |

*VALUE uslfnum, records;*

*INTEGER uslfnum; records;*

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Moves information block on a USL file.* |
| *TYPE:* | *Integer* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCL* |
| *ERROR CODE:* | *83* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-55* |

PROCEDURE

> ALTDSEG (index, inc, size);

VALUE          index, size;

LOGICAL        index;

INTEGER        inc, size;

OPTIONAL EXTERNAL;

FUNCTION:           Changes size of an extra data segment.

TYPE:                None

OPTIONAL PARAMETERS:   None

CONDITION CODES:       CCE, CCG, CCL

ERROR CODE:             134

SPECIAL ATTRIBUTE       +

OPTIONAL CAPABILITY:     Data Segment Management

TEXT DISCUSSION:        12-7

PROCEDURE                     | ARITRAP (state); |

VALUE          state;

LOGICAL        state;

OPTION EXTERNAL;

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

TYPE:                     None

OPTIONAL PARAMETERS:      None

CONDITION CODES:          CCE, CCG

ERROR CODE:               51

SPECIAL ATTRIBUTES:       None

OPTIONAL CAPABILITY:      None

TEXT DISCUSSION:          8-27

| | |
|---|---|
| INTEGER PROCEDURE | ASCII (word, base, string); |
| VALUE | word, base; |
| LOGICAL | word; |
| INTEGER | base; |
| BYTE ARRAY | string; |
| OPTION EXTERNAL; | |

| | |
|---|---|
| FUNCTION: | Converts a number from binary to ASCII code. |
| TYPE: | Integer |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | None |
| ERROR CODE: | 63 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 8-8 |

| | |
|---|---|
| *LOGICAL PROCEDURE* | *BINARY (string, length);* |
| *VALUE* | *length;* |
| *BYTE ARRAY* | *string;* |
| *INTEGER* | *length;* |
| *OPTION EXTERNAL;* | |


| | |
|---|---|
| *FUNCTION:* | *Converts a number from ASCII to binary code.* |
| *TYPE:* | *Logical* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCG, CCL* |
| *ERROR CODE:* | *62* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-7* |

*PROCEDURE*

> *CAUSEBREAK;*

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Requests a session break.* |
| *TYPE:* | *None* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCL* |
| *ERROR CODE:* | *56* |
| *SPECIAL ATTRIBUTES:* | *$* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-41* |

LOGICAL PROCEDURE

CHECKDEV (ldev, hdwraddr);

VALUE                    ldev;

INTEGER                  ldev, hdwaddr;

OPTION EXTERNAL;

| | |
|---|---|
| FUNCTION: | Verifies input/output device acquisition |
| TYPE: | Logical |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCG |
| ERROR CODE: | None |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 8-53 |

*LONG PROCEDURE*          | *CHRONOS;* |

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Returns current date and time.* |
| *TYPE:* | *Long* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *None* |
| *ERROR CODE:* | *41* |
| *SPECIAL ATTRIBUTES:* | *$* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-14* |

*PROCEDURE*

> *COMMAND (comimage, error, parm);*

*BYTE ARRAY*   *comimage;*

*INTEGER*        *error, param;*

*OPTIONAL EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Executes an MPE/3000 command programmatically.* |
| *TYPE:* | *None* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCG, CCL* |
| *ERROR CODE:* | *68* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-25* |

| PROCEDURE | CREATE | (*progname*, entryname, *pin*, param, flags, stacksize, dlsize, maxdata, priorityclass, rank); |
|-----------|--------|------|

VALUE          param, stacksize, dlsize, priority-class, maxdata, flags, rank;

LOGICAL        priorityclass, flags;

INTEGER        stacksize, dlsize, maxdata, *pin*, param, rank;

BYTE ARRAY     *progname*, entryname;

OPTION VARIABLE, EXTERNAL;


FUNCTION:              *Creates a process.*

TYPE:                  *None*

OPTIONAL PARAMETERS:   entryname, param, flags, stacksize, dlsize, maxdata, priorityclass, rank

CONDITION CODES:       *CCE, CCG, CCL*

ERROR CODE:            *100*

SPECIAL ATTRIBUTES:    *None*

OPTIONAL CAPABILITY:   *Process Handling*

TEXT DISCUSSION:       *11-7*

INTEGER PROCEDURE | DASCII   (dword, base, string);

VALUE           dword, base;

DOUBLE          dword;

INTEGER         base;

BYTE ARRAY      string;

OPTION EXTERNAL;

FUNCTION:                    Converts a value from double-word binary to ASCII.

TYPE:                        Integer

OPTIONAL PARAMETERS:         None

CONDITION CODES:             None

ERROR CODE:                  75

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             8-9

| DOUBLE PROCEDURE | | DBINARY   (string, length); |
| --- | --- | --- |
| VALUE | length; | |
| BYTE ARRAY | string; | |
| INTEGER | length; | |
| OPTION EXTERNAL; | | |

| | |
| --- | --- |
| FUNCTION: | Converts a number from ASCII to double-word binary value. |
| TYPE: | Double |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCG, CCL |
| ERROR CODE: | 74 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 8-8 |

PROCEDURE  | DEBUG |

OPTION EXTERNAL;

FUNCTION:                      Sets breakpoints and modifies or displays stack or register
                               contents.

TYPE:                          None

OPTIONAL PARAMETERS:           None

CONDITION CODES:               None

ERROR CODE:                    None

SPECIAL ATTRIBUTES:            None

OPTIONAL CAPABILITY:           None

TEXT DISCUSSION:               8-43

INTEGER PROCEDURE      | DLSIZE  (size);

VALUE             size;

INTEGER          size;

OPTION EXTERNAL;


FUNCTION:               Changes size of DL-DB area.

TYPE:                  Integer

OPTIONAL PARAMETERS:   None

CONDITION CODES:      CCE, CCG, CCL

ERROR CODE:            135

SPECIAL ATTRIBUTES:    None

OPTIONAL CAPABILITY:   None

TEXT DISCUSSION:      8-39

PROCEDURE

| DMOVIN (index, disp, number, location); |

VALUE           index, disp, number;

LOGICAL         index;

INTEGER         disp, number;

ARRAY           location;

OPTION EXTERNAL;

FUNCTION:                   Copies block from data segment to stack.

TYPE:                       None

OPTIONAL PARAMETERS:        None

CONDITION CODES:            CCE, CCG, CCL

ERROR CODE:                 132

SPECIAL ATTRIBUTES:         +

OPTIONAL CAPABILITY:        Data Segment Management

TEXT DISCUSSION:            12-4

PROCEDURE                      | DMOVOUT (index, disp, number, location); |

VALUE            index, disp, number;

LOGICAL          index;

INTEGER          disp, number;

ARRAY            location;

OPTION EXTERNAL;


FUNCTION:                Copies block from stack to data segment.

TYPE:                    None

OPTIONAL PARAMETERS:     None

CONDITION CODES:         CCE, CCG, CCL

ERROR CODE:              133

SPECIAL ATTRIBUTES:      +

OPTIONAL CAPABILITY:     Data Segment Management

TEXT DISCUSSION:         12-6

INTEGER PROCEDURE              EXPANDUSLF (uslfnum, records);

VALUE              uslfnum, records;

INTEGER              uslfnum, records;

OPTION EXTERNAL:

FUNCTION:                    Changes length of a USL file

TYPE:                        Integer

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCL

ERROR CODE:                  84

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             8-55

*INTEGER PROCEDURE*       | *FATHER;* |

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Requests PIN of father process.* |
| *TYPE:* | *Integer* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCG, CCL* |
| *ERROR CODE:* | *109* |
| *SPECIAL ATTRIBUTES:* | *$* |
| *OPTIONAL CAPABILITY:* | *Process Handling* |
| *TEXT DISCUSSION:* | *11-24* |

*PROCEDURE*

| FCHECK  *(filenum,* **errorcode, tlog, blknum, numrecs***);* |
| --- |

*VALUE*          *filenum;*

*INTEGER*         *filenum,* **errorcode, tlog, numrecs;**

*DOUBLE*          **blknum;**

*OPTION VARIABLE; EXTERNAL;*


*FUNCTION:*               *Requests details about file input/output errors.*

*TYPE:*                   *None*

*OPTIONAL PARAMETERS:*    **errorcode, tlog, blknum, numrecs**

*CONDITION CODES:*        *CCE, CCL*

*ERROR CODE:*             *10*

*SPECIAL ATTRIBUTES:*     *$*

*OPTIONAL CAPABILITY:*    *None*

*TEXT DISCUSSION:*        *6-40*

PROCEDURE | FCLOSE (filenum, disposition, seccode);

VALUE         filenum, disposition, seccode;

INTEGER       filenum, disposition, seccode;

OPTION EXTERNAL;


FUNCTION:              Closes a file.

TYPE:                    None

OPTIONAL PARAMETERS:    None

CONDITION CODES:       CCE, CCL

ERROR CODE:           9

SPECIAL ATTRIBUTES:      $

OPTIONAL CAPABILITY:     None

TEXT DISCUSSION:        6-18

PROCEDURE

| FCONTROL (filenum, controlcode, param); |

VALUE       *filenum, controlcode;*

INTEGER     *filenum, controlcode;*

LOGICAL     *param;*

OPTION EXTERNAL;


FUNCTION:                  *Performs various control operations on a file or terminal device.*

TYPE:                      *None*

OPTIONAL PARAMETERS:       *None*

CONDITION CODES:           *CCE, CCL, CCG*

ERROR CODE:                *13*

SPECIAL ATTRIBUTES:        *$*

OPTIONAL CAPABILITY:       *None*

TEXT DISCUSSION:           *6-44, 8-62, 8-65, 8-66, 8-67, 8-68 8-70, 8-71, 8-72*

PROCEDURE

> FGETINFO *(filenum,* filename, foptions, aoptions, recsize, devtype, ldnum, hdaddr, filecode, recpt, eof, flimit, logcount, physcount, blksize, extsize, numextents, userlabels, creatorid, labaddr*)*;

VALUE:          *filenum;*

INTEGER:        *filenum,* recsize, detype, filecode, blksize, numextents, userlabels;

BYTE ARRAY      filename, creatorid;

LOGICAL         foptions, aoptions, ldnum, hdaddr, extsize;

DOUBLE          recpt,  eof, flimit, logcount, physcount, labaddr;

OPTION VARIABLE, EXTERNAL;

FUNCTION:                   *Requests access and status information about a file.*

TYPE:                       *None*

OPTIONAL PARAMETERS:        filename, foptions, aoptions, recsize, detype, ldnum, hdaddr, filecode, recpt, eof, flimit, logcount, physcount, blksize, extsize, numextents, userlabels, creatorid, labaddr

CONDITION CODES:            CCE, CCL

ERROR CODE:                 *11*

SPECIAL ATTRIBUTES:         *None*

OPTIONAL CAPABILITY:        *None*

TEXT DISCUSSION:            **6-36**

PROCEDURE

| FLOCK (filenum, lockcond); |
|---|

VALUE           filenum, lockcond;

INTEGER        filenum;

LOGICAL        lockcond;

OPTION EXTERNAL;


FUNCTION:                  Dynamically locks a file.

TYPE:                      None

OPTIONAL PARAMETERS:    None

CONDITION CODES:         CCE, CCG, CCL

ERROR CODE:             15

SPECIAL ATTRIBUTES:      $

OPTIONAL CAPABILITY:     None

TEXT DISCUSSION:          9-10

| *INTEGER PROCEDURE* | *FOPEN* *(formaldesignator, foptions,* aoptions, recsize, device, formmsg, userlabels, blockfactor, numbuffers, filesize, numextents, initialloc, filecode); |
|---|---|

*VALUE*           *foptions,* aoptions, recsize, userlabels, blockfactor, numbuffers, filesize, numextents, initialloc, filecode;

*BYTE ARRAY*   *formaldesignator,* device, formmsg;

*LOGICAL*        *foptions,* aoptions;

*INTEGER*        recsize, userlabels, blockfactor, numbuffers, numextents, initialloc, filecode;

*DOUBLE*         filesize;

*OPTION VARIABLE, EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Opens a file.* |
| *TYPE:* | *Integer* |
| *OPTIONAL PARAMETERS:* | formaldesignator,foptions, aoptions, recsize, device, formmsg, userlabels, blockfactor, numbuffers, filesize, numextents, initialloc, filecode |
| *CONDITION CODES:* | *CCE, CCL* |
| *ERROR CODE:* | *1* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *6-7* |

PROCEDURE                    | FPOINT (filenum, recnum); |

VALUE            filenum, recnum;

INTEGER         filenum;

DOUBLE          recnum;

OPTION EXTERNAL;


FUNCTION:                  Resets the logical record pointer for a sequential disc file.

TYPE:                      None

OPTIONAL PARAMETERS:       None

CONDITION CODES:           CCE, CCG, CCL

ERROR CODE:                6

SPECIAL ATTRIBUTES:        $

OPTIONAL CAPABILITY:       None

TEXT DISCUSSION:           6-35

INTEGER PROCEDURE

| FREAD (filenum, target, tcount); |

VALUE           filenum, tcount;

INTEGER         filenum, tcount;

ARRAY           target;

OPTION EXTERNAL;

FUNCTION:                    Reads a logical record from a sequential file (on any device)
                             to the user's stack.

TYPE:                        Integer

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCG, CCL

ERROR CODE:                  2

SPECIAL ATTRIBUTES:          $

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             6-20

| PROCEDURE | | FREADDIR  (filenum, target, tcount, recnum); |
|---|---|---|
| VALUE | filenum, tcount, recnum; | |
| INTEGER | filenum; | |
| ARRAY | target; | |
| INTEGER | tcount; | |
| DOUBLE | recnum; | |
| OPTION EXTERNAL; | | |

| FUNCTION: | Reads a logical record from a direct-access disc file to the user's data stack. |
|---|---|
| TYPE: | None |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCG, CCL |
| ERROR CODE: | 7 |
| SPECIAL ATTRIBUTES: | $ |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 6-22 |

| PROCEDURE | FREADLABEL (filenum, target, tcount); labelid) |
|---|---|

| | |
|---|---|
| VALUE | filenum, **tcount**, labelid |
| INTEGER | filenum, **tcount**, labelid |
| ARRAY | target |
| OPTION VARIABLE, EXTERNAL; | |

| | |
|---|---|
| FUNCTION: | Reads a user file label. |
| TYPE: | None |
| OPTIONAL PARAMETERS: | **tcount** |
| CONDITION CODES: | CCE, CCG, CCL |
| ERROR CODES: | 19 |
| SPECIAL ATTRIBUTES: | $ |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 6-31 |

PROCEDURE

| FREADSEEK  (filenum, recnum); |

VALUE           filenum, recnum;

INTEGER         filenum;

DOUBLE          recnum;

OPTION EXTERNAL;


FUNCTION:                  Prepares, in advance, for reading from a direct-access file.

TYPE:                      None

OPTIONAL PARAMETERS:       None

CONDITION CODES:           CCE, CCG, CCL

ERROR CODE:                12

SPECIAL ATTRIBUTES:        $

OPTIONAL CAPABILITY:       None

TEXT DISCUSSION:           6-24

*PROCEDURE*  | *FREEDSEG (index, id);*

*VALUE*  index, id;

*LOGICAL*  index, id;

*OPTIONAL EXTERNAL;*

*FUNCTION:*  *Releases an extra data segment.*

*TYPE:*  *None*

*OPTIONAL PARAMETERS:*  *None*

*CONDITION CODES:*  *CCE, CCG, CCL*

*ERROR CODE:*  *131*

*SPECIAL ATTRIBUTES:*  *+*

*OPTIONAL CAPABILITY:*  *Data Segment Management*

*TEXT DISCUSSION:*  *12-3*

*PROCEDURE*          | *FREELOCRIN;* |

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Frees all local RIN's from allocation to a job.* |
| *TYPE:* | *None* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCG, CCL* |
| *ERROR CODE:* | *31* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *9-9* |

LOGICAL PROCEDURE:    | FRELATE  (infilenum, listfilenum); |

VALUE         infilenum, listfilenum;

INTEGER        infilenum, listfilenum;

OPTION EXTERNAL;


FUNCTION:                Declares a file pair interactive or duplicative.

TYPE:                    Logical

OPTIONAL PARAMETERS:     None

CONDITION CODES:         CCE, CCL

ERROR CODE:              18

SPECIAL ATTRIBUTES:      $

OPTIONAL CAPABILITY:     None

TEXT DISCUSSION:         6-50

```
PROCEDURE               ┌─────────────────────────────────────────┐
                        │ FRENAME  (filenum, newfilereference);   │
                        └─────────────────────────────────────────┘
VALUE          filenum;

INTEGER        filenum;

BYTE ARRAY     newfilereference;

OPTION EXTERNAL;
```

| | |
|---|---|
| FUNCTION: | Renames a disc file. |
| TYPE: | None |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCL |
| ERROR CODE: | 17 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 6-49 |

PROCEDURE

| FSETMODE (filenum, modeflags); |
| --- |

VALUE          filenum, modeflags;

INTEGER        filenum;

LOGICAL        modflags;

OPTION EXTERNAL;


FUNCTION:                    Activates or deactivates file-access modes.

TYPE:                        None

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCL

ERROR CODE:                  14

SPECIAL ATTRIBUTES:          $

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             6-47

PROCEDURE | FSPACE (filenum, displacement);

VALUE            filenum, displacement;

INTEGER          filenum, displacement;

OPTION EXTERNAL;

| | |
|---|---|
| FUNCTION: | Spaces forward or backward on a file. |
| TYPE: | None |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCG, CCL |
| ERROR CODE: | 5 |
| SPECIAL ATTRIBUTES: | $ |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 6-34 |

PROCEDURE

| FUNLOCK (filenum); |

VALUE          filenum;

INTEGER        filenum;

OPTION EXTERNAL;

FUNCTION:                 Dynamically unlocks a file.

TYPE:                     None

OPTIONAL PARAMETERS:      None

CONDITION CODES:          CCE, CCG, CCL

ERROR CODE:               16

SPECIAL ATTRIBUTES:       $

OPTIONAL CAPABILITY:      None

TEXT DISCUSSION:          9-11

PROCEDURE

> FUPDATE (filenum, target, tcount);

VALUE                filenum, tcount;

INTEGER              filenum, tcount;

ARRAY                target;

OPTIONAL EXTERNAL;


FUNCTION:               Updates a logical record residing in a disc file.

TYPE:                   None

OPTIONAL PARAMETERS:    None

CONDITION CODES:        CCE, CCG, CCL

ERROR CODE:             4

SPECIAL ATTRIBUTES:     $

OPTIONAL CAPABILITY:    None

TEXT DISCUSSION:        6-33

PROCEDURE

FWRITE (filenum, target, tcount, control);

VALUE          filenum, tcount, control;

INTEGER        filenum, tcount;

ARRAY          target;

LOGICAL        control;

OPTION EXTERNAL;


FUNCTION:                    Writes a logical record from the user's stack to a sequential
                             file (on any device).

TYPE:                        None

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCG, CCL

ERROR CODE:                  3

SPECIAL ATTRIBUTES:          $

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             6-25

| PROCEDURE | FWRITEDIR (filenum, target, tcount, recnum); |
|---|---|

VALUE   *filenum, tcount, recnum;*

INTEGER  *filenum, tcount;*

ARRAY   *target;*

INTEGER  *tcount;*

DOUBLE  *recnum;*

OPTION EXTERNAL;


| | |
|---|---|
| FUNCTION: | Writes a logical record from the user's stack to a direct-access disc file. |
| TYPE: | None |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCG, CCL |
| ERROR CODE: | 8 |
| SPECIAL ATTRIBUTES: | $ |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 6-29 |

PROCEDURE

> FWRITELABEL (filenumber, target, tcount);

VALUE                filenum, tcount;

INTEGER              filenum, tcount;

ARRAY                target;

OPTION VARIABLE, EXTERNAL;

FUNCTION:                Writes a user's file label.

TYPE:                    None

OPTIONAL PARAMETERS:     tcount

CONDITION CODES:         CCE, CCG, CCL

ERROR CODE:              20

SPECIAL ATTRIBUTES:      $

OPTIONAL CAPABILITY:     None

TEXT DISCUSSION:         6-32

| PROCEDURE | | GETDSEG (index, length, id); |
|---|---|---|

VALUE            *id;*

LOGICAL          *index, id;*

INTEGER          *length;*

OPTION EXTERNAL;

FUNCTION:                    *Creates an extra data segment.*

TYPE:                        *None*

OPTIONAL PARAMETERS:         *None*

CONDITION CODES:             *CCE, CCG, CCL*

ERROR CODE                   *130*

SPECIAL ATTRIBUTES:          *+*

OPTIONAL CAPABILITY:         *Data Segment Management*

TEXT DISCUSSION:             *12-2*

*LOGICAL PROCEDURE*

| *GETJCW;* |
|---|

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Fetches contents of job control word.* |
| *TYPE:* | *Logical* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *None* |
| *ERROR CODE:* | *73* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-52* |

PROCEDURE | GETLOCRIN (rincount);

VALUE        rincount;

LOGICAL      rincount;

OPTION EXTERNAL;



FUNCTION:              Acquires local RIN's. (The number acquired = rincount.)

TYPE:                  None

OPTIONAL PARAMETERS:   None

CONDITION CODES:       CCE, CCG, CCL

ERROR CODE:            30

SPECIAL ATTRIBUTES:    None

OPTIONAL CAPABILITY:   None

TEXT DISCUSSION:       9-6

*INTEGER PROCEDURE*

> *GETORIGIN;*

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Determines source of activation call.* |
| *TYPE:* | *Integer* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *None* |
| *ERROR CODE:* | *105* |
| *SPECIAL ATTRIBUTES:* | *$* |
| *OPTIONAL CAPABILITY:* | *Process Handling* |
| *TEXT DISCUSSION:* | *11-24* |

PROCEDURE

| GETPRIORITY *(pin, priorityclass, rank)*; |
| --- |

VALUE           *pin, priorityclass*, **rank**;

LOGICAL       *priorityclass*;

INTEGER       *pin*, **rank**;

OPTION VARIABLE, EXTERNAL;

FUNCTION:           *Reschedules a process.*

TYPE:           *None*

OPTIONAL PARAMETERS:    **rank**

CONDITION CODES:     *CCE, CCL*

ERROR CODE:         *120*

SPECIAL ATTRIBUTES:    *$*

OPTIONAL CAPABILITY:    *Process Handling*

TEXT DISCUSSION:      *11-22*

## PRIVILEGED-MODE OPTIONAL CAPABILITY INTRINSICS

*PROCEDURE*        | *GETPRIVMODE;* |

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Dynamically enters privileged mode.* |
| *TYPE:* | *None* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCG* |
| *ERROR CODE:* | *200* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *Privileged Mode* |
| *TEXT DISCUSSION:* | *14-3* |

INTEGER PROCEDURE  | GETPROCID (son);

VALUE            son;

LOGICAL          son;

OPTION EXTERNAL;

| | |
|---|---|
| FUNCTION: | Requests PIN of a son process. |
| TYPE: | Integer |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | None |
| ERROR CODE: | 112 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | Process Handling |
| TEXT DISCUSSION: | 11-25 |

*DOUBLE PROCEDURE*   | *GETPROCINFO (pin);* |

*VALUE*            pin;

*INTEGER*          pin;

*OPTION EXTERNAL;*

*FUNCTION:*                 Requests status information about a father or son process.

*TYPE:*                     Double

*OPTIONAL PARAMETERS:*      None

*CONDITION CODES:*          CCE, CCG, CCL

*ERROR CODE:*               110

*SPECIAL ATTRIBUTES:*       $

*OPTIONAL CAPABILITY:*      Process Handling

*TEXT DISCUSSION:*          11-27

PROCEDURE          | GETUSERMODE; |

OPTION EXTERNAL;

FUNCTION:                    *Dynamically returns to non-privileged mode.*

TYPE:                        *None*

OPTIONAL PARAMETERS:         *None*

CONDITION CODES:             *CCE, CCG*

ERROR CODE:                  *201*

SPECIAL ATTRIBUTES:          *None*

OPTIONAL CAPABILITY:         *Privileged Mode*

TEXT DISCUSSED:              *14-4*

INTEGER PROCEDURE          | INITUSLF (uslfnum, rec0); |

VALUE          uslfnum;

INTEGER          uslfnum;

INTEGER ARRAY rec 0;

OPTION EXTERNAL;

FUNCTION:                Initializes buffer for a USL file to the empty state.

TYPE:                Integer

OPTIONAL PARAMETERS:     None

CONDITION CODES:         CCE, CCL

ERROR CODE:              82

SPECIAL ATTRIBUTES:      None

OPTIONAL CAPABILITY:     None

TEXT DISCUSSION:         8-54

PROCEDURE

| KILL (pin); |

VALUE            pin;

INTEGER          pin;

OPTION EXTERNAL;

FUNCTION:                    Deletes a process.

TYPE:                          None

OPTIONAL PARAMETERS:    None

CONDITION CODES:        CCE, CCG, CCL

ERROR CODE:              102

SPECIAL ATTRIBUTES:     None

OPTIONAL CAPABILITY:     Process Handling

TEXT DISCUSSION:        11-15

| INTEGER PROCEDURE | | LOADPROC (procname, lib, plabel); |

INTEGER PROCEDURE           | LOADPROC   (procname, lib, plabel); |

VALUE            lib;

BYTE ARRAY       procname;

INTEGER          lib, plabel.

OPTION EXTERNAL;



| | |
|---|---|
| FUNCTION: | Dynamically loads a library procedure. |
| TYPE: | Integer |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCL |
| ERROR CODE: | 80 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 7-28 |

PROCEDURE
$\boxed{LOCKGLORIN \ (rinnum, \ lockcond, \ rinpassword);}$

VALUE            rinnum;

LOGICAL          rinnum, lockcond;

BYTE ARRAY   rinpassword;

OPTION EXTERNAL;

FUNCTION:                    Locks a global RIN.

TYPE:                        None

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCG

ERROR CODE:                  34

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             9-3

PROCEDURE | LOCKLOCRIN  (rinnum,lockcond);

VALUE          rinnum;

LOGICAL          rinnum, lockcond;

OPTION EXTERNAL;

FUNCTION:                    Locks a local RIN.

TYPE:                        None

OPTIONAL PARAMETERS:    None

CONDITION CODES:            CCE, CCG, CCL

ERROR CODE:                  32

SPECIAL ATTRIBUTES:        None

OPTIONAL CAPABILITY:       None

TEXT DISCUSSION:            9-7

LOGICAL PROCEDURE    MAIL (pin, count);

VALUE          pin;

INTEGER        pin, count;

OPTION EXTERNAL;

FUNCTION:              Tests mailbox status.

TYPE:                  Logical

OPTIONAL PARAMETERS:   None

CONDITION CODES:       CCE, CCG

ERROR CODE:            106

SPECIAL ATTRIBUTES:    None

OPTIONAL CAPABILITY:   Process Handling

TEXT DISCUSSION:       11-16

| *INTEGER PROCEDURE* | *MYCOMMAND* *(comimage, delimiters, maxparms, numparms,* *parms,* **dict, defn***);* |

*VALUE*          *maxparms;*

*BYTE ARRAY*    *comimage, delimiters,* **dict***;*

*INTEGER*       *maxparms, numparms;*

*DOUBLE ARRAY*  *parms;*

*BYTE POINTER*   **defn***;*

*OPTION VARIABLE, EXTERNAL;*


| | |
|---|---|
| *FUNCTION:* | *Parses (delineates and defines parameters) for user-supplied command image.* |
| *TYPE:* | *Integer* |
| *OPTIONAL PARAMETERS:* | **dict, defn** |
| *CONDITION CODES:* | *CCE, CCG, CCL* |
| *ERROR CODE:* | *71* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-21* |

| PROCEDURE | PRINT (message, length, control); |
|---|---|

VALUE          length, control;

ARRAY          message;

INTEGER        length, control;

OPTION EXTERNAL;


FUNCTION:                    Prints character string on job/session listing device.

TYPE:                        None

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCL, CCG

ERROR CODE:                  65

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             8-12

| PROCEDURE | PRINTOP (message, length, control); |
|-----------|--------------------------------------|

VALUE         length, control;

BYTE ARRAY   message;

INTEGER       length, control;

OPTION EXTERNAL;


FUNCTION:               Prints a character string on Operator's Console.

TYPE:                   None

OPTIONAL PARAMETERS:    None

CONDITION CODES:        CCE, CCL, CCG

ERROR CODE:             66

SPECIAL ATTRIBUTES:     None

OPTIONAL CAPABILITY:    None

TEXT DISCUSSION:        8-13

DOUBLE PROCEDURE | PROCTIME; |

OPTION EXTERNAL;

| FUNCTION: | Returns a process' accumulated central processor-time. |
| TYPE: | Double |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODE: | None |
| ERROR CODE: | 42 |
| SPECIAL ATTRIBUTE: | $ |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 8-14 |

PROCEDURE

> PTAPE    *(filenum1, filenum2);*

VALUE            *filenum1, filenum2;*

INTEGER          *filenum1, filenum2;*

OPTION EXTERNAL;

FUNCTION:                    *Accepts input from paper tapes not containing X-OFF control characters.*

TYPE:                        *None*

OPTIONAL PARAMETERS:         *None*

CONDITION CODES:             *CCE, CCG, CCL*

ERROR CODE:                  *191*

SPECIAL ATTRIBUTES:          *$*

OPTIONAL CAPABILITY:         *None*

TEXT DISCUSSION:             *8-70*

PROCEDURE                    | QUIT (num); |

VALUE            num;

INTEGER          num;

OPTION EXTERNAL;

FUNCTION:                 A borts a process.

TYPE:                     None

OPTIONAL PARAMETERS:      None

CONDITION CODE:           None

ERROR CODE:               76

SPECIAL ATTRIBUTES:       $

OPTIONAL CAPABILITY:      None

TEXT DISCUSSION:          8-42

*PROCEDURE*                    | *QUITPROG (num);* |

*VALUE*          *num;*

*INTEGER*        *num;*

*OPTION EXTERNAL;*


*FUNCTION:*                  *Aborts a program*

*TYPE:*                      *None*

*OPTIONAL PARAMETERS:*       *None*

*CONDITION CODE:*            *61*

*SPECIAL ATTRIBUTES:*        *$*

*OPTIONAL CAPABILITY:*       *None*

*TEXT DISCUSSION:*           *8-43*

| | |
|---|---|
| INTEGER PROCEDURE | READ (message, expectedl); |
| VALUE | expectedl; |
| ARRAY | message; |
| INTEGER | expectedl; |
| OPTION EXTERNAL; | |


| | |
|---|---|
| FUNCTION: | Reads an ASCII string from an input device. |
| TYPE: | Integer |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCG, CCL |
| ERROR CODE: | 64 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 8-10 |

*LOGICAL PROCEDURE*  |‾ *RECEIVEMAIL  (pin, location, waitflag);* ‾|

*VALUE*          *pin, waitflag;*

*INTEGER*      *pin;*

*LOGICAL*      *waitflag;*

*ARRAY*        *location;*

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Receives mail from another process.* |
| *TYPE:* | *Logical* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCG, CCL* |
| *ERROR CODE:* | *108* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *Process-Handling* |
| *TEXT DISCUSSION:* | *11-19* |

*PROCEDURE*　　　　　　　　| *RESETCONTROL* |

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Resets terminal to accept CONTROL-Y signal.* |
| *TYPE:* | *None* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCL* |
| *ERROR CODE:* | *55* |
| *SPECIAL ATTRIBUTES:* | *None* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-32* |

INTEGER PROCEDURE

| SEARCH  (target, length, dict, **defn**); |

VALUE           length;

BYTE ARRAY    target, dict;

INTEGER        length;

BYTE POINTER **defn**;

OPTION VARIABLE, EXTERNAL;

FUNCTION:                    Searches an array for specified entry or name

TYPE:                        Integer

OPTIONAL PARAMETERS:         **defn**

CONDITION CODES:             None

ERROR CODE:                  70

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             8-18

LOGICAL PROCEDURE | SENDMAIL (pin, count, location, waitflag);

VALUE               pin, count, waitflag;

INTEGER             pin, count;

LOGICAL             waitflag;

ARRAY               location;

OPTION EXTERNAL;

FUNCTION:               Sends mail to another process.

TYPE:                   Logical

OPTIONAL PARAMETERS:    None

CONDITION CODES:        CCE, CCG, CCL

ERROR CODE:             107

SPECIAL ATTRIBUTES:     None

OPTIONAL CAPABILITY:    Process Handling

TEXT DISCUSSION:        11-17

| PROCEDURE | | SETJCW (word); |
| --- | --- | --- |
| VALUE | word; | |
| LOGICAL | word; | |
| OPTION EXTERNAL; | | |

| | |
| --- | --- |
| FUNCTION: | Sets bits in job control word. |
| TYPE: | None |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | None |
| ERROR CODE: | 72 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 8-52 |

PROCEDURE | SUSPEND *(susp, rin);*

VALUE          *susp*, rin;

LOGICAL       *susp;*

INTEGER       rin;

OPTION VARIABLE, EXTERNAL;

FUNCTION:               *Suspends a process.*

TYPE:                    *None*

OPTIONAL PARAMETERS:    rin

CONDITION CODES:        *CCE, CCL*

ERROR CODE:            *103*

SPECIAL ATTRIBUTE:      *$*

OPTIONAL CAPABILITY:     *Process Handling*

TEXT DISCUSSION:        *11-12*

*LOGICAL PROCEDURE*  ┌─────────────────────────┐
                     │ *SWITCHDB  (index);*    │
                     └─────────────────────────┘

*VALUE*           *index;*

*LOGICAL*         *index;*

*OPTION PRIVILEGED, EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Switches DB-register pointer.* |
| *TYPE:* | *Logical* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *CCE, CCL* |
| *ERROR CODE:* | *139* |
| *SPECIAL ATTRIBUTES:* | *\** |
| *OPTIONAL CAPABILITY:* | *Privileged Mode* |
| *TEXT DISCUSSION:* | *14-5* |

*PROCEDURE*

| *TERMINATE;* |

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Terminates a process.* |
| *TYPE:* | *None* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *None* |
| *ERROR CODE:* | *60* |
| *SPECIAL ATTRIBUTES:* | *$* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-42, 11-13* |

*DOUBLE PROCEDURE*  | *TIMER;* |

*OPTION EXTERNAL;*

| | |
|---|---|
| *FUNCTION:* | *Returns system-timer bit-count.* |
| *TYPE:* | *Double* |
| *OPTIONAL PARAMETERS:* | *None* |
| *CONDITION CODES:* | *None* |
| *ERROR CODE:* | *40* |
| *SPECIAL ATTRIBUTES:* | *$* |
| *OPTIONAL CAPABILITY:* | *None* |
| *TEXT DISCUSSION:* | *8-13* |

PROCEDURE

| | UNLOADPROC (procid); |
|---|---|

VALUE          procid;

LOGICAL         procid;

OPTION EXTERNAL;

| | |
|---|---|
| FUNCTION: | Dynamically unloads a library procedure. |
| TYPE: | None |
| OPTIONAL PARAMETERS: | None |
| CONDITION CODES: | CCE, CCL |
| ERROR CODE: | 81 |
| SPECIAL ATTRIBUTES: | None |
| OPTIONAL CAPABILITY: | None |
| TEXT DISCUSSION: | 7-29 |

PROCEDURE

<div style="border: 1px solid">UNLOCKGLORIN (rinnum);</div>

VALUE            rinnum;

LOGICAL          rinnum;

OPTION EXTERNAL;

FUNCTION:                    Unlocks a global RIN.

TYPE:                        None

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCG, CCL

ERROR CODE:                  35

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             9-4

PROCEDURE

<div style="border: 1px solid black; display: inline-block; padding: 4px;">UNLOCKLOCRIN  *(rinnum)*;</div>

VALUE          *rinnum;*

LOGICAL        *rinnum;*

OPTION EXTERNAL;



FUNCTION:                    *Unlocks a local RIN.*

TYPE:                        *None*

OPTIONAL PARAMETERS:         *None*

CONDITION CODE;              *CCE, CCG, CCL*

ERROR CODE:                  *33*

SPECIAL ATTRIBUTES:          *None*

OPTIONAL CAPABILITY:         *None*

TEXT DISCUSSION:             *9-8*

*PROCEDURE*

> *WHO* *(*mode, capability, lattr, usern, groupn, acctn,
> homen, termn*);*

*LOGICAL*          mode, termn;

*DOUBLE*          capability, lattr;

*BYTE ARRAY*    usern, groupn, acctn, homen;

*OPTION VARIABLE, EXTERNAL;*

*FUNCTION:*                 *Returns user attributes.*

*TYPE:*                      *None*

*OPTIONAL PARAMETERS:*     mode, capability, lattr, usern, groupn, acctn, homen, termn

*CONDITION CODES:*          *None*

*ERROR CODE:*               69

*SPECIAL ATTRIBUTES:*       *None*

*OPTIONAL CAPABILITY:*      *None*

*TEXT DISCUSSION:*          *8-15*

PROCEDURE

| XARITRAP *(mask,plabel,oldmask,oldplabel);* |

VALUE          *mask, plabel;*

INTEGER        *mask, plabel, oldmask, oldplabel;*

OPTION EXTERNAL;

.

FUNCTION:              *Arms the software arithmetic trap.*

TYPE:                  *None.*

OPTIONAL PARAMETERS:   *None*

CONDITION CODES:       *CCE, CCG, CCL*

ERROR CODE:            *50*

SPECIAL ATTRIBUTES:    *None*

OPTIONAL CAPABILITY:   *None*

TEXT DISCUSSION:       *8-27*

PROCEDURE

| XCONTRAP (plabel, oldplabel); |
| --- |

VALUE            plabel;

INTEGER          plabel, oldplabel;

OPTION EXTERNAL;

FUNCTION:                    Arms or disarms the control-Y trap.

TYPE:                        None

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCG, CCL

ERROR CODE:                  54

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             8-32

PROCEDURE  | XLIBTRAP  (plabel, oldplabel);

VALUE        plabel;

INTEGER      plabel, oldplabel;

OPTION EXTERNAL;


FUNCTION:                  *Arms or disarms the software library trap.*

TYPE:                      *None*

OPTIONAL PARAMETERS:       *None*

CONDITION CODES:           *CCE, CCG, CCL*

ERROR CODE:                *52*

SPECIAL ATTRIBUTES:        *None*

OPTIONAL CAPABILITY:       *None*

TEXT DISCUSSION:           *8-29*

PROCEDURE | XSYSTRAP (plabel, oldlabel);

VALUE        plabel;

INTEGER      plabel, oldplabel;

OPTION EXTERNAL;

FUNCTION:                    Arms or disarms the system trap.

TYPE:                        None

OPTIONAL PARAMETERS:         None

CONDITION CODES:             CCE, CCG, CCL

ERROR CODE:                  53

SPECIAL ATTRIBUTES:          None

OPTIONAL CAPABILITY:         None

TEXT DISCUSSION:             8-30

INTEGER PROCEDURE       | ZSIZE  (size); |

VALUE           size;

INTEGER         size;

OPTION EXTERNAL;

FUNCTION:              Changes size of Z-DB area.

TYPE:                 Integer

OPTIONAL PARAMETERS:   None

CONDITION CODES:       CCE, CCG, CCL

ERROR CODE:           136

SPECIAL ATTRIBUTES:    None

OPTIONAL CAPABILITY:   None

TEXT DISCUSSION:       8-40

The following listing shows intrinsic error numbers versus the intrinsics to which they pertain:

| Error Number | Intrinsic | Error Number | Intrinsic |
|---|---|---|---|
| 1 | FOPEN | 40 | TIMER |
| 2 | FREAD | 41 | CHRONOS |
| 3 | FWRITE | 42 | PROCTIME |
| 4 | FUPDATE | 50 | XARITRAP |
| 5 | FSPACE | 51 | ARITRAP |
| 6 | FPOINT | 52 | XLIBTRAP |
| 7 | FREADDIR | 53 | XSYSTRAP |
| 8 | FWRITEDIR | 54 | XCONTRAP |
| 9 | FCLOSE | 55 | RESETCONTROL |
| 10 | FCHECK | 56 | CAUSEBREAK |
| 11 | FGETINFO | 60 | TERMINATE |
| 12 | FREADSEEK | 62 | BINARY |
| 13 | FCONTROL | 63 | ASCII |
| 14 | FSETMODE | 64 | READ |
| 15 | FLOCK | 65 | PRINT |
| 16 | FUNLOCK | 66 | PRINTOP |
| 17 | FRENAME | 68 | COMMAND |
| 18 | FRELATE | 69 | WHO |
| 19 | FREADLABEL | 70 | SEARCH |
| 20 | FWRITELABEL | 71 | MYCOMMAND |
| 30 | GETLOCRIN | 72 | SETJCW |
| 31 | FREELOCRIN | 73 | GETJCW |
| 32 | LOCKLOCRIN | 74 | DBINARY |
| 33 | UNLOCKLOCRIN | 75 | DASCII |
| 34 | LOCKGLORIN | 76 | QUIT |
| 35 | UNLOCKGLORIN | 80 | LOADPROC |

| Error Number | Intrinsic | Error Number | Intrinsic |
|---|---|---|---|
| 81 | UNLOADPROC | 112 | GETPROCID |
| 82 | INITUSLF | 120 | GETPRIORITY |
| 83 | ADJUSTUSLF | 130 | GETDSEG |
| 84 | EXPANDUSLF | 131 | FREEDSEG |
| 100 | CREATE | 132 | DMOVIN |
| 102 | KILL | 133 | DMOVEOUT |
| 103 | SUSPEND | 134 | ALTDSEG |
| 104 | ACTIVATE | 135 | DLSIZE |
| 105 | GETORIGIN | 136 | ZSIZE |
| 106 | MAIL | 139 | SWITCHDB |
| 107 | SENDMAIL | 191 | PTAPE |
| 108 | RECEIVEMAIL | 200 | GETPRIVMODE |
| 109 | FATHER | 201 | GETUSERMODE |
| 110 | GETPROCINFO | None | CHECKDEV |

# APPENDIX E
# Disc File Labels

Whenever a disc file is created, MPE/3000 automatically supplies a file label in the first sector of the first extent occupied by that file. Such labels always appear in the format described below. (User-supplied labels, if present, are located in the sectors immediately following the MPE/3000 file label.) The contents of a label may be listed by using the *:LISTF -1* command described in Section V.

| Words | | Contents |
|---|---|---|
| 0-3 | | Local file name. |
| 4-7 | | Group name. |
| 8-11 | | Account name. |
| 12-15 | | Identity of file creator. |
| 16-19 | | File lockword. |
| 20-21 | | File security matrix. |
| 22 | (Bits 0:15) | Not used. |
| | (Bit 15:1) | File secure bit:<br><br>If 1, file secured.<br><br>If 0, file released. |
| 23 | | File creation date |
| 24 | | Last access date. |
| 25 | | Last modification date. |
| 26 | | File code. |
| 27 | | File control block vector. |
| 28 | (Bit 0:1) | Store Bit. (If on, :STORE or :RESTORE, in progress.) |
| | (Bit 1:1) | Restore Bit. (If on, :RESTORE in progress.) |
| | (Bit 2:1) | Load Bit. (If on, program file is loaded.) |
| | (Bit 3:1) | Exclusive Bit. (If on, file is opened with exclusive access.) |

For words 23, 24, 25: Dates in same format as return value for CHRONOS intrinsic (Section VIII.)

| Words | | Contents |
|---|---|---|
| | (Bits 4:4) | Device sub-type. |
| | (Bits 8:6) | Device type. |
| | (Bit 14:1) | File is open for write. |
| | (Bit 15:1) | File is open for read. |
| 29 | (Bits 0:8) | Number of user labels written. |
| | (Bits 8:8) | Number of user labels. |
| 30-31 | | Maximum number of logical records. |
| 32-34 | | Not used. |
| 35 | | Cold-load identity. |
| 36 | | Foptions specifications. |
| 37 | | Logical record size (in negative bytes). |
| 38 | | Block size (in words). |
| 39 | (Bits 0:8) | Sector offset to data. |
| | (Bits 8:4) | Not used. |
| | (Bits 12:4) | Number of extents. |
| 40 | | Not used. |
| 41 | | Extent size. |
| 42-43 | | Number of logical records in file. |
| 44-45 | | Address of first extent. |
| 46-47 | | Address of second extent. |
| 48-49 | | Address of third extent. |
| 50-51 | | Address of fourth extent. |
| 52-53 | | Address of fifth extent. |
| 54-55 | | Address of sixth extent. |
| 56-57 | | Address of seventh extent. |
| 58-59 | | Address of eighth extent. |
| 60-61 | | Address of ninth extent. |
| 62-63 | | Address of tenth extent. |
| 64-65 | | Address of eleventh extent. |
| 66-67 | | Address of twelfth extent. |
| 68-69 | | Address of thirteenth extent. |
| 70-71 | | Address of fourteenth extent. |
| 72-73 | | Address of fifteenth extent. |
| 74-75 | | Address of sixteenth extent. |

# APPENDIX F
# :STORE Tape Format

The format of a tape created by the :STORE command is:

| Item No. | Item |
|---|---|

**1**    End-of-File (EOF) Mark

**2**    EOF Mark

**3**    Header label (40 words), used as follows:

| Words | Contents |
|---|---|
| 0-13 | "STORE/RESTORE LABEL-HP/3000." |
| 14-22 | Unused by MPE/3000. |
| 23 | Reel number. |
| 24 | Bits (0:7) = last 2 digits of year<br>(7:9) = Julian date } Date of creation. |
| 25 | Bits (0:8) = hours<br>(8:8) = minutes } Time of creation. |
| 26 | Bits (0:8) = seconds<br>(8:8) = tenth-of-seconds } |
| 27-39 | Unused by MPE/3000. |

ε

**4**    EOF Mark

**5**    Tape directory — Consists of 12-word entries, blocked 85 per 1020-word record. (The last record may be shorter.) There is one entry for each file on the tape, and the entries are ordered the same as the files. The 12-word entry is:

| Word | Contents |
|---|---|
| 0-3 | File name. |
| 4-7 | Group name. |
| 8-11 | Account name. |

**6**    EOF Mark

| Item No. | Item |
|---|---|
| 7 | First file. The data is blocked with 1024 words per physical tape record. (The last record may be shorter, but will aways be a multiple of 128 words.) For fixed-length and undefined-length record files, only data up to the end-of-file is dumped; intervening zero-length extents are not dumped. For variable-length record files, only allocated extents are dumped. |
| 8 | EOF Mark |
| 9 | Second File |
| 10 | EOF Mark |

.

.

.

| 11 | Last File |
| 12 | EOF Mark |
| 13 | Trailer Label (40 words). Identical to header label (Item 3) except that Words 21 and 22 are used as follows: |

| Word | Use |
|---|---|
| 21 | =1 means that preceding file ended with preceding EOF mark |
| 22 | =1 means that entire tape set ends with preceding EOF mark |

| 14 | EOF Mark |
| 15 | EOF Mark |
| 16 | EOF Mark |

:STORE tapes may have multiple reels. If end-of-tape (EOT) is detected during a write data operation, a file mark is written followed by Items 13 to 16 above, with word 21 of the trailer label set to 1 if this was the last record of the file and 0 otherwise. If EOT is detected on a write file mark operation, Items 13 to 16 are written, with word 21 set to 1 and word 22 set to 1 if this is the last file on the tape, and 0 otherwise. Reels subsequent to Reel 1 have the following format:

1. Header label

2. EOF mark

3. Remainder of preceding file or next file

4. EOF mark

5. Next file; the rest of the tape is written in the same format as the first reel.

# APPENDIX G
# *End-of-File Indication*

The end-of-file indication will be returned by the card reader and tape drivers under conditions specified by the initiators of read requests. The type of requests are as follows:

| Type | Class of end-of-file |
|------|----------------------|
| A | All records that begin with a colon (:). |
| B | All records that contain, starting in the first byte, :EOD, :EOJ, :JOB and :DATA (See Note.) |
| E | Hardware-sensed end-of-file |

> *NOTE:* If the word count is less than 3 or the byte count is less than 6, then Type B reads are converted to Type A reads.

In utilizing the card/tape devices as files via the File System, the following types are assigned:

| File Specified | Type |
|----------------|------|
| $STDIN | Type A. |
| $STDINX | Type B. |
| Dev=CARD/TAPE | Type B, if device job/data accepting.<br>Type E, if device not job/data accepting. |

Any subsequent requests initiated to the driver following sensing of an end-of-file condition will be rejected with an end-of-file indication.

When reading from an unlabeled tape file, the request encountering a tape mark will respond with an end-of-file indication but succeeding requests will be allowed to continue to read data past the tape mark. Under these conditions, it is the responsibility of the caller to protect against the occurrence of data beyond an end-of-file and to prevent reading off the end of the reel.

# *Index*

# Index of Commands

# *Index of Intrinsics*

# READER COMMENT SHEET

We welcome your evaluation of this manual. Your comments and suggestions help us improve our publications. Please use additional pages if necessary.

**Is this manual technically accurate?**

**Is this manual complete?**

**Is this manual easy to read and use?**

**Other comments?**

_____

**FROM:**

    **Name**    _____

    **Company**    _____

    **Address**    _____

                 _____

                 _____

03000-90005