MP/M II
Operating System
PROGRAMMER'S GUIDE

Digital Research
P.O. Box 579
801 Lighthouse Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CP/M is a registered trademark of Digital Research. CP/NET, MP/M, and MP/M II are trademarks of Digital Research.

The "MP/M II Programmer's Guide" was prepared using the Digital Research TEX Text Formatter and printed in the United States of America by Commercial Press of Monterey.

```
*******************************
*  First Edition: August 1981 *
* Second Edition: August 1982 *
*******************************
```

Foreword

MP/M II T.M., is a multi-user operating system for microcomputers that use the Intel 8080, the Zilog Z800, or similar 8-bit type architecture. It will support multi-terminal access with multi programming at each terminal. It uses the same Basic Disk Operating System (BDOS) as CP/M@ thus assuring compatibilty of existing programs running under CP/M.

The minimum hardware environment for MP/M II must include an 8080 or Z80 processor, 32K bytes of random access memory (RAM),a system console, and a real-time clock. A typical MP/M II kernel occupies approximately 15K bytes.

This manual describes the programming interface to MP/M II. It gives a general description of the modules that comprise the operating system, the manner in which MP/M II manages the memory resource and monitors running processes, as well as detailed descriptions of all the system entry points. Also included are descriptions of several utility programs that are useful for creating and debugging programs under MP/M II. This manual is not intended as a tutorial. Therefore, familiarity with the material covered in the User's Guide and with processor architecture and assembly language in general is required.

Table of Contents

Table of Contents
(continued)

Table of Contents
(continued)

Table of Contents
(continued)

Appendixes

Section I
Introduction to MP/M 11

1.1 Overview of MP/M II Features

MP/M II is a microcomputer operating system that supports multiple terminals with multi-programming at each terminal. Upward compatible with CP/M, MP/M II presents a CP/M interface to each terminal. In fact, most CP/M programs can run without modification under MP/M II. However, MP/M II is not limited to this model. Using MP/M II's powerful multi-programming capability, a single terminal can initiate more than one program. In addition, the system functions used by MP/M II to control the multi-programming environment are available to application programs. As a result, MP/M II supports extended features beyond the CP/M model such as communication between and synchronization of independently running programs.

Under MP/M II, there is an important distinction between a program and a process. A program is simply a block of code residing somewhere in memory or on disk; it is essentially static. A process on the other hand, is dynamic, and can be thought of as a "logical machine" that not only executes the program's code, but also executes code in the operating system. When MP/M II loads a program, it also creates a process that is associated with the loaded program. Subsequently, it is the process, rather than the program that controls all access to the system's resources. Thus, MP/M II monitors the process, not the program. This distinction is a subtle one, but vital to understanding the operation of the system as a whole.

Programs running under MP/M II fall into three categories: CP/M programs, MP/M II system processes, and MP/M II Resident System Processes. The first category consists of CP/M-like programs that MP/M II loads into an available memory segment. MP/M II supports from 1 to 7 memory segments or partitions that can be loaded with programs. once loaded and initiated, a program becomes associated with a process that is maintained by the MP/M II real-time nucleus.

The second category consists of MP/M II system processes that perform operating system tasks. For example, the Command Line Interpreter (CLI), is the system process that loads and initiates user programs.

The final category consists of those processes that can be optionally integrated into MP/M II during system generation, thus becoming a part of the system. These processes are called Resident System Processes (RSPS) . With RSPs, users can write custom processes and include them in the system along with those supplied with MP/M II (see Section 1.7 and Section 8) . All processes running under MP/M II compete for the CPU and other system resources on a priority basis under control of the real-time nucleus.

The following list briefly summarizes MP/M II's capabilities.

- Multi-terminal support. MP/M II supports up to 16 terminals. Also, a single process can access multiple terminals.

- Multi-programming at each terminal. Any system console can initiate multiple programs or processes. In addition, a process can generate sub-processes.

- Support for bank-switched memory. MP/M II's memory segments can either reside in common memory or be distributed through separate memory banks, thereby extending the system's effective memory capacity.

- Inter-process communication, synchronization, and mutual exclusion. These functions are provided by queues.

- Logical interrupt mechanism using flags. This allows MP/M II to interface with any physical interrupt structure.

- System timing functions. These functions enable processes running under MP/M II to compute elapsed times, delay execution for specified intervals, and to access and set the current date and time. In addition, the user can schedule programs to be run by date and time. The system timing is also used to provide round-robin scheduling of compute-bound processes executing at the same priority.

- User-selected options at system generation time. The available options include the number of system consoles, the number, size, and location of memory segments, and the maximum number of files and locked records supported by the system at one time. Also, the user can select which RSPs to include with MP/M II during system generation.

Functionally, MP/M II is composed of three distinct modules: the Basic Disk Operating System (BDOS) , the Extended Disk Operating System (XDOS) , and the Extended I/O System (XIOS) . The MP/M II BDOS is an upward-compatible version of the single-user CP/M BDOS. In most cases, CP/M programs that make BDOS calls for I/O or direct BIOS calls for printer and console I/O, can run under MP/M II without modification. However, MP/M II's BDOS is extended to provide support for multiple console and list devices. In addition, the file system is extended to provide services required in multi user environments.

Two major extensions to the file system are:

- File locking. Normally, files opened under MP/M II cannot be opened or deleted by other users. This feature prevents accidental conflicts with other users.

- Shared access to files. As a special option, independent users can open the same file in shared or unlocked mode. MP/M II supports record locking and unlocking commands for files opened in this mode, and protects files opened in shared mode from deletion by other users.

The XDOS module gives MP/M II its multi-programming capabilities. It contains the real-time nucleus that monitors the execution of processes and arbitrates conflicts for the system's resources. It also includes the Terminal message Process (TMP) which reads and echoes command lines for the system consoles, and the Command Line Interpreter (CLI) which accepts TMP command lines and initiates user programs and RSPs. The XDOS also contains the set of extended MP/M II functions that can be accessed by user programs.

The XIOS module is similar to the CP/M BIOS module but is extended in several ways. Primitive functions such as console I/O are modified to support multiple consoles. Several new primitive functions support MP/M II's additional features. Also, new facilities are added to eliminate wait loops. The XIOS is the hardware-dependent module that defines MP/M II's interface to a particular hardware environment. Although a standard XIOS is supplied by Digital Research, the XIOS is usually customized to support the user's own hardware environment. Note: processes running under MP/M II can make direct XIOS calls only for console and list I/O.

When MP/M II is configured for a single console and is executing a single program, its speed approximates that of CP/M. The overhead of the MP/M II dispatcher in such an environment will be 7 to 15%. In environments where either multiple processes and/or users are running, the speed of each individual process is degraded in proportion to the amount of I/O and compute resources required. A process that performs a large amount of I/O in proportion to computing exhibits only minor speed degradation. This also applies to a process that performs a large amount of computing, but is running concurrently with other processes that are largely I/O bound. on the other hand, significant speed degradation occurs in those environments where more than one compute-bound process is running.

1.2 MP/M II Nucleus

MP/M II is controlled by a real-time multi-tasking nucleus that resides within the XDOS module. This nucleus performs process dispatching, memory management, and system timing tasks. It also performs queue management, flag management, device polling, and console and list device management. The following sections describe these functions in greater detail. Many of the system functions that perform these tasks can also be called by user programs with the XDOS functions.

Although MP/M II is a multi-processing operating system, at any given point in time, only one process has access to the CPU resource. Unless it is specifically written to communicate or synchronize execution with other processes, it runs unaware that other processes may be competing for the system's resources. Eventually, the system suspends the process from execution and gives another process the opportunity to run.

1.2.1 Process Dispatching

The primary task of the nucleus is transferring the CPU resource from one process to another. This task is called dispatching and is performed by a part of the nucleus called the Dispatcher. Under MP/M II, each process is associated with a data structure called a Process Descriptor (see Section 3.2) The Dispatcher uses this data structure to save and restore the current state of a running process. Every process in the system resides in one of three states: ready, running, or suspended. A ready process is one that is waiting for the CPU resource. A suspended process is one that is waiting for some other system resource or a defined event. A running process is one that the CPU is currently executing.

A dispatch operation for a running process can be described as follows:

1) The Dispatcher suspends the process from execution and stores the current state in the Process Descriptor.

2) The Dispatcher scans all the suspended processes on the Ready List and selects the one with the highest priority.

3) The Dispatcher restores the state of the selected process from its Process Descriptor and gives it the CPU resource.

4) The process executes until it makes a system call, or an interrupt, or a tick of the system clock occurs. Then, dispatching is repeated.

Only processes that are placed on the Ready List are eligible for selection during dispatch. By definition, a process is on the Ready List if it is waiting for the CPU resource only. Processes waiting for other system resources cannot execute until their resource requirements are satisfied. Under MP/M II, a process is blocked from execution if it is waiting for:

- a queue message so that it can complete a read queue operation.
- space to become available in a queue so it can complete a queue write operation.
- system flag to be set.
- console or list device to become available.
- a specified number of system clock ticks before it can be removed from the system Delay List.
- an I/O event to complete.

These situations are discussed in more detail in the following sections.

MP/M II is a priority-driven system. This means that the Dispatcher selects the highest priority ready process and gives it the CPU resource. Processes with the same priority are "round robin" scheduled. That is, they are given equal CPU time slices when executing CPU bound code. With priority dispatching, control is never passed to a lower priority process if there is a higher priority process on the Ready List. Since high priority compute bound processes tend to monopolize the CPU resource, it is advisable to lower their priority to avoid degrading overall system performance. In addition, compute-bound processes can make XDOS Dispatch calls periodically to promote sharing of the CPU resource in those systems that do not support a clock. When a process makes a Dispatch call, the call appears as a null operation to the process, but allows other processes to gain access to the CPU resource.

MP/M II requires that at least one process be running at all times. To ensure this, the system maintains the IDLE process on the Ready List so it can be dispatched if there are no other processes available. The IDLE process runs at a very low priority and is never blocked from execution. It does not perform any useful task, but simply gives the system a process to run when no other ready processes exist.

1.2.2 Queue Management

Queues perform several critical functions for processes running under MP/M II. They are used for communicating messages between processes, for synchronizing process execution, and for mutual exclusion. Queues are special data structures, implemented in MP/M II as "memory files" that contain room for a specified number of fixed length messages (see Section 3.3). Like files, queues are made, opened, deleted, read from, and written to with XDOS function calls. When a queue is created with the XDOS Make Queue command, it is assigned an 8-character name that identifies the queue in XDOS Open Queue commands. As the name implies, messages are read from a queue on a first-in, first-out basis.

A process can read messages from a queue or write messages to a queue in two ways: conditionally or unconditionally. If no messages exist in the queue when a conditional read is performed, or the queue is full when a conditional write is performed, the system returns an error code to the calling process. On the other hand, if a process performs an unconditional read from an empty queue, the system suspends the process from execution until another process writes a message to the queue. A process suspended in this manner is placed on the queue's Dequeue list. A similar situation occurs when a process makes an unconditional write to a full queue. A process suspended in this way is placed on the queue's Enqueue list. MP/M II uses these Enqueue/Dequeue lists to synchronize process execution.

When more than one process resides on a queue's Enqueue or Dequeue list, preference is given to the higher priority process. Conflicts involving processes with the same priority are resolved on a first-come first-serve basis.

Mutual exclusion queues are a special type of queue under MP/M II. They contain one message of zero length and are assigned a name beginning with the upper-case letters, MX. In effect, a mutual exclusion queue is a binary semaphore. Mutual exclusion queues ensure that only one process has access to a resource at a time. Access to a resource protected by a mutual exclusion queue takes place as follows:

1) The process issues an unconditional Read Queue call from the queue protecting the resource, thereby suspending itself until the message is available.

2) The process accesses the protected resource.

3) The process writes the message back to the queue when it has finished using the protected resource, thus freeing the resource for other processes.

As an example, the disk system mutual exclusion queue, MXdisk, ensures that processes serially access the BDOS file system.

Mutual exclusion queues have one other feature that is different from normal queues. When a process reads a message from a mutual exclusion queue, the nucleus saves the address of the Process Descriptor for the process reading the message in a two-byte buffer area of the queue. If the process is aborted while it owns the mutual exclusion message, the nucleus automatically writes the message back to the queue for the aborted process, thus enabling other processes to gain access to the protected resource.

1.2.3 Flag Management

MP/M II's nucleus uses flags for signaling and synchronizing processes with defined events. Processes access the system's flags with the XDOS functions, Flag Set and Flag Wait. Internally, a flag can reside in two states: set or reset. The reset state is further divided into two categories:

- No process is waiting for the flag to be set.
- A process is waiting for the flag to be set, and blocked from execution until it is set.

Note: Two processes are not allowed to wait on the same flag. This is an error situation referred to as flag "under-run". Similarly, a process attempting to set a flag that is already set is another error situation, called flag "over-run".

Flags provide a logical interrupt system independent of the physical interrupt system of the microcomputer. They are primarily intended for use by the XIOS module to support the Interrupt Handler. For example, when the Interrupt Handler receives a physical interrupt indicating an I/0 operation is complete, it sets a flag and branches to the Dispatcher. A process suspended from execution because it is waiting for the flag to be set, is placed on the Ready List, making it eligible for selection during dispatch. once dispatched, the process can assume the I/O operation is complete.

MP/M II supports 32 flags, several of which are reserved. For example, Flag 1 is reserved for the system clock tick. Because of their limited number, their use by the XIOS module, and the single process nature of their design, flags should not be used in application software except in very special situations. In most cases, process communication and synchronization are better accomplished with queues.

1.2.4 Device Polling

Device polling is another mechanism a process can use to wait for an I/O or external event without using flags or consuming the CPU resource with a programmed delay loop. Polling is implemented in the XIOS module exclusively. For example, assuming that the XIOS supports polled console input, when a process makes a BDOS console input call, the process eventually reaches the XIOS console input routine where the actual hardware dependent input operation is performed. Before performing the input operation, the nucleus tests to see if a character is ready for input. If it is ready, the nucleus performs the input operation and execution of the process continues. If a character is not ready, the process must wait. In a single-user environment under CP/M, the BIOS can simply loop on console status until a character is read. Under MP/M II, this technique cannot be used because it consumes the CPU resource. If the looping process has a high priority, any other lower-priority processes on the Ready List are denied the CPU resource.

Device polling avoids this situation because the Dispatcher makes the console status test. If a character is not ready, the XIOS makes an XDOS Poll call. This suspends the running process on the system Poll List. Subsequently, in every dispatch operation, the Dispatcher makes a single console status call for the process. When the status call indicates a character is ready, the nucleus removes the process from the Poll List and places it on the Ready List. Thus device polling is one of the ways a process can wait for an external or I/O event to occur without monopolizing the CPU resource.

1.2.5 Console and List Device Management

Console and List devices are special resources under MP/M II. When the system gives a console or list device to a process, it internally stores the address of the Process Descriptor, thereby recording ownership of the device by that process. If another process attempts to use the device, the nucleus suspends the calling process and places it on the device's Wait List. It remains on this list until the process owning the device either terminates execution or detaches from the device. When this occurs, the nucleus selects the highest priority waiting process, gives it the device, places it on the Ready List, and performs a dispatch.

Processes can own more than one console or list device. Fields within the Process Descriptor designate which device is to be used in I/O operations. A process gains ownership of a device by a mechanism called attaching. If a process attaches a device when the device is free, the process gains ownership of the device. otherwise, the process is suspended from execution, as described above. As an option, a process can conditionally attach to a device in which case it is notified if another process owns the device. Conditional attachment gives a process more control over its own execution instead of leaving it up to the nucleus. Thus a process can avoid being suspended when it does not depend on a specific device.

1,2.6 Memory Management

The MP/M II nucleus can manage from one to eight memory segments. These segments are of fixed length, and used primarily as regions for loading transient programs. The partitions are page aligned, which means that they must begin on a page boundary. Because a page is defined as 256 bytes, a page boundary always begins at an address where the low-order byte is 0. The nucleus manages the memory resource with XDOS functions that allocate and free memory segments. Figure 1-1 illustrates how memory is organized under MP/M II.

```
Top of Memory
                  +------------+
                  : MP/M II    :
       (Common)   : Operating  :
                  : System     :
                  :            :
 Top of Banked    :------------:     +--------+     +--------+
      Memory      :            :     ://///////:     ://///////:
                  : Segment 0  :     ://///////:     ://///////:
                  :            :     ://///////:     ://///////:
                  :            :     ://///////:     ://///////:
                  :            :     ://///////:     ://///////:
                  : MP/M II    :     ://///////:     ://///////:
(Bank Switched)   : Extension  :     ://///////:     ://///////:
                  :------------:     ://///////:     ://///////:
                  :////////////:     ://///////:     ://///////:
                  :////////////:     ://///////:     ://///////:
                  :////////////:     ://///////:     ://///////:
Low               :////////////:     ://///////:     ://///////:
Memory            :////////////:     ://///////:     ://///////:
                  :////////////:     ://///////:     ://///////:
                  +------------+     +--------+     +--------+
                     Bank 0            Bank 1   ....  Bank N
```

Figure 1-1. MP/M II Memory Organization

The shaded areas represent those regions that can support memory segments. If bank-switched memory is not used, available memory is restricted to bank zero. The total number of memory segments, in addition to their size and bank locations, are system generation options. Segment 0, however, is a special segment reserved for system modules and RSPs. It always resides immediately below the operating system region in bank 0.

In bank-switched systems, the operating system module resides in common memory. In addition, all Process Descriptors and queues must reside in the common memory region. Typically, the common memory size is 16K but the size can vary on systems capable of switching memory in units smaller than 16K. As a result, the typical maximum memory segment size is 48K. The largest user memory segment that can be allocated to bank 0 is usually much less than this value.

More than one memory segment can be defined in a single bank. Memory segments that do not begin at 0 can only be used to execute page relocatable (PRL) programs. Memory segments beginning at 0, can execute COM or PRL programs.

1.2.7 System Timing Functions

MP/M II's system timing functions include: keeping the time of day, delaying the execution of a process for a specified period of time, and scheduling programs to be loaded from disk and executed. The XDOS internal process, CLOCK, provides the time of day for the system. This process issues Flag Wait calls on the system one second flag, Flag 2. When the XIOS Interrupt Handler sets this flag, it wakes up the CLOCK process which then increments the internal time and date. Subsequently, the CLOCK process makes another Flag Wait call and suspends itself until the flag is set again. MP/M II provides functions that allow the user to set and access the internal date and time. In addition, the BDOS uses the internal time and date to record when a file is updated, created, or last accessed.

The XDOS Delay function replaces the typical programmed delay loop for delaying process execution. The Delay function requires that a tick be supported in the XIOS and that Flag 1, the system tick flag, be set every 16 to 20 milliseconds (usually 60 times a second) . When a process makes a Delay call, it specifies the number of ticks it is to be suspended from execution. The system maintains the address of the Process Descriptor for the process on an internal Delay List along with its current delay tick count. A system process, TICK, waits on the tick flag and decrements this delay count on each system tick. When the delay count goes to zero, the process is removed from the Delay List and placed on the Ready List.

MP/M II can schedule the execution of a transient program or a Resident System Process only if the Resident System Process, SCHED, is included at system generation time.

1.3 MP/M II Memory Structure

The memory structure of the MP/M II operating system is shown in Figure 1-2.

```
High       +-------------+  < - - +- - +
Memory     : SYSTEM.DAT  :        :    :
           +-------------+        :    :
           : TMPD.DAT    :        :    :
           +-------------+        :    :
           : USERSYS.STK :        :    :
           +-------------+        :    :
           : XIOSJMP.TBL :        :    :
           +-------------+        :    :
           : RESBDOS.SPR :        :    :--- Common
           +-------------+        :    :              Memory
           : XDOS.SPR    :        :    :
           +-------------+        :    :
           : RSP's       :        :    :
           +-------------+        :    :
           : BNKX:OS.SPR :        :< - +
           +-------------+        :
           : BNKBDOS.SPR :        :--- Memory Segment 0
           +-------------+        :    Bank 0
           : BNKXDOS.SPR :        :
           +-------------+        :
           : TMP.SPR     :        :
           +-------------+        :
           : BRS'S       :        :
           +-------------+        :
           : LCKLSTS.DAT :        :
Low        +-------------+        :
Memory     : CONSOLE.DAT :        :
           +-------------+  < - - +
```

Figure 1-2. MP/M II Memory Structure

The exact memory addresses of each of the memory segments shown above vary with the MP/M II version and depend on the user specifications made during the system generation process.

If the host system is bank-switched, the modules above the BNKXIOS.SPR module must reside in common memory. Common memory is always accessible no matter what bank is used. The modules below the BNKXIOS.SPR module must reside in bank 0, which is defined as the bank of memory active when MP/M II is loaded. The BNKXIOS.SPR module itself can reside partly in common memory and partly in bank 0. If bank-switching is not used, then all of memory is common. The memory segments shown in Figure 1-2 are described below.

The SYSTEM.DAT segment contains 256 bytes used by the MP/M II GENSYS to dynamically configure the system. After loading, the system uses this area for storage of system data such as submit flags. See Section 3.5 for the details of the SYSTEM.DAT segment

The size of the TMPD.DAT segment depends on the number of consoles specified for the system during the system generation process. MP/M II supports from 1 to 16 consoles, and associated with each console is a Terminal Message Processor (TMP) , identified as TMPO through TMP15. The TMP provides the command line support for each console. Each console uses 64 bytes within the TMPD.DAT segment to contain a TMP Process Descriptor. The size of the USERSYS.STK segment varies according to the number of consoles, as shown in Table---1-1.

Table 1-1. TMPD.DAT Segment Size

Size	Number of Consoles
OOOH	No user system stacks
100H	1 to 4 consoles
200H	5 to 7 consoles

The USERSYS.STK segment is included if the user selects the option to add system call user stacks during system generation. If included, the system temporarily uses 64 bytes of stack space in this segment when user programs make BDOS function calls. This option allows users to run CP/M *.COM files under MP/M II. Some BDOS function calls, especially console I/O functions, consume more stack under MP/M II than CP/M. The system allocates space for user system stacks from the USERSYS.STK segment for each user memory segment. The size of the USERSYS.STK segment varies according to the number of memory segments, as shown in Table 1-2.

Table 1-2. USERSYS.STK Segment Size

Size	Number of Memory Segments
OOOH	No user system stacks
100H	1 to 4 memory segments
200H	5 to 7 memory segments

The XIOSJMP.TBL segment is a copy of the first page of the BNKXIOS.SPR module. It is required because the system divides the BDOS module into two sub-modules, RESBDOS.SPR and BANKBDOS.SPR. The RESBDOS module accesses the BNKXIOS via the XI.OSJMP.TBL module. The BANKBDOS module accesses the BNKXIOS module directly. The XIOSJMP.TBL module is 256 bytes in length.

The RESBDOS.SPR segment contains the resident portion of the BDOS module. The BDOS functions supported by this segment include those not involved with the BDOS file system such as console and list I/O. The RESBDOS.SPR segment is approximately OBOOH bytes in length.

The XDOS.SPR segment contains the MP/M II nucleus and the extended disk operating system. This segment is approximately 2300H bytes in length.

RSPs can use two segments within MP/M II. The first segment resides in common memory, and exists only if one or more RSPs are included during system generation. This common memory segment RSP contains all RSP Process Descriptors and queues. The second segment named the BRS segment exists in the non-common portion of memory segment 0. It is present only when one or more banked RSPs are included during system generation (See Section 1.7).

The BNKXIOS.SPR module contains the user-customized Basic and Extended I/O System in page- relocatable format (PRL) . It can extend across the common memory boundary. In general, code supporting the BDOS file system can reside in bank 0 while code supporting console and list I/O must reside in common memory. Refer to the MP/M II System Guide for more information regarding the BNKXIOS module.

The BNKBDOS.SPR module contains the non-resident portion of the BDOS module. All BDOS functions related to the file system are supported by this segment. This segment is approximately 2300H bytes in length.

The BNKXDOS.SPR module contains the non-resident portion of the XDOS module. This segment will vary in length with MP/M II version.

The TMP.SPR module contains the code for the reentrant Terminal Message Process. This module is approximately 300H bytes in length.

The BRS segment contains data and code used by banked RSPs that does not have to be in common memory. Banked RSPs are valuable because they minimize the common memory requirement.

The LCKLSTS.DAT segment is a special data structure that maintains a record of open files and locked records on the system. Each open file and locked record consumes a 10-byte entry in this segment. The size of this segment is determined by parameters specified during system generation.

The size of the CONSOLE.DAT segment depends on the number of consoles specified for the system during the system generation process. MP/M II supports from 1 to 16 consoles, and associated with each console is a Terminal Message Processor (TMP) , identified as TMPO through TMP15. The TMP provides the command line support for each console. Each console uses 256 bytes within the CONSOLE.DAT segment to contain the stack and buffers for its TMP. The code for the TMP's is reentrant and resides within the TMP.SPR segment.

The remaining memory is available for allocation to user memory segments. The size, bank location, and number of user memory segments are system generation options. MP/M II uses these memory segments to load and execute transient programs.

1.4 Terminal Message Process

The Terminal Message Process (TMP) refers to one of a collection of XDOS system processes that are associated with the system consoles. Each system console has its own TMP designated as TMPO through TMP15. The number of system consoles implemented depends on the number supported in the XIOS and how many are specified during system generation. Clearly, the number of system consoles cannot exceed the number supported in the XIOS. However, a smaller number than the XIOS supported maximum can be specified during system generation.

The system maintains the buffers, stack, and local variables for each TMP in each system console's region of the CONSOLE.DAT segment. The process descriptors for the TMP's are located in the TMPD.DAT segment. The code, which is shared by all the TMP's, is a single re-entrant routine within the TMP.SPR module. Thus, while each TMP performs the same function for each system console, they compete with each other as well as with any other running processes for the CPU resource.

The TMP provides the command line support for system consoles within MP/M II. This includes displaying the system prompt at the console:

```
OA>
```

and reading the command line. The TMP reads the command line from one of two sources: the console or a Submit file. Normally, it reads from the console with the BDOS Read Buffer Input command. Alternatively, it reads from the N.SUB file (N = the console number) on the MP/M II system disk. This occurs only if the user has previously entered a submit file at the console with the SUBMIT facility.

After reading a command line, the TMP performs one of two actions depending on the type of command entered. If the command line is a new drive specification:

```
OA>B:
```

the TMP issues a BDOS Select Disk call to select the new drive. If the system supports the newly selected drive, the TMP updates the drive field of its Process Descriptor, displays the new prompt:

```
OB>
```

and waits for the next command line.

If the command is in any other form, the TMP assigns its console to another system process, the Command Line Interpreter, (CLI). The TMP then sends the command line along with fields specifying its default drive, user number, list device and console number to the CLI with the XDOS Send CLI Command. It then attempts to attach the console. This suspends the TMP from execution because it no longer owns the system console. When the console becomes free, the TMP reissues the prompt and the cycle repeats.

Note: The command level default drive and current user number are maintained in the TMP Process Descriptor for each system console. This information is displayed in the system prompt. If an application program changes the current disk or user number by making an explicit BDOS call, the TMP Process Descriptor values are not changed. The USER utility does update the TMP Process Descriptor user number when it sets the user number to a new value. To do this, it locates the TMP Process Descriptor associated with the console and updates its user number field.

1.5 Command Line Interpreter

When the Command Line Interpreter (CLI) receives a command line sent to it with the XDOS Send Cli Command, it interprets the command, and initiates the requested transient program or RSP. Normally, the TMP sends the command line to the CLI. However, other processes can also use the Send CLI Command function. Also, the BDOS Program Chain function is implemented internally with the Send CLI Command. Note: Any process making a Send Cli Command call must first assign its console to the CLI.

The Send CLI Command function sends the command line to the CLI by attempting to write the command line message to the system queue, "CliQ". The command line message contains the current disk, user number, list device and system console number in addition to the ASCII command line. The CLIQ is a single message queue with a length of 129 bytes. If the CLIQ already contains a command line message, the nucleus suspends the process issuing the Send CLI Command, and places it on the CLIQ's Enqueue List, where it remains until the CLI reads the message. Once the CLI reads the message, the process must compete with any others that may also reside on the Enqueue List for the opportunity to write its message and regain the ready state. The process with the highest priority that has been on the list the longest always goes first.

The CLI accepts command line messages by reading the CLIQ. If the queue is empty, the CLI is blocked from execution when it issues the CLIQ read command. In this case, the CLI is suspended on the CLIQ Dequeue List until another process issues a Send CLI Command, at which point the CLI is removed from the Dequeue List and placed on the Ready List. When it gets the CPU resource, the CLI's read queue operation is completed and it receives the command line message.

The command line read by the CLI must be in ASCII and usually takes the form:

```
<command> <command tail>
```

where

```
<command>          => {d:}filename{;password} or
                   => queuename

<command tail>     => <file spec> or
                   => <file spec><delimiter><file spec>

<file spec>        => {d:}filename{.typ}{;password}

<delimiter>        => one or more blanks or a tab or
                      one of the following: "=,/[]<>"

d:                 => MP/M II drive specification, "A"
                      through "P"

filename           => 1 to 8 character file name

typ                => 1 to 3 character file type

password           => 1 to 8 character password value

queuename          => 1 to 8 character queue name of
                      Resident System Process
```

Fields enclosed in curly brackets are optional. If there is no drive specification {d:}, the current default drive is assumed. If the type field {.typ} is omitted, a type field of all blanks is implied. If the password field {;password} is omitted, a password field of all blanks is implied. No type field is included in the <command> file specification because the CLI assumes either a PRL or COM type.

After the CLI reads a command line, it performs the following steps:

1) It parses the command line to pick up the <command> field.

2) If there is no drive specification or password field, the CLI attempts to open a queue named by the command field. If the queue open is successful, the CLI assumes the queue belongs to an RSP, and attempts to assign the console to that RSP. If the RSP name is the same as its queue name, the console assignment is made. In fact, this is the way a RSP controls whether or not it receives the console resource when it is initiated by the CLI. The CLI then writes the <command tail> message along with the current disk, user number, list device and system console number to

the RSP's queue. Because the RSP is typically blocked from execution because of a queue read from its queue, this sequence initiates the RSP for execution.

3) If the command field does not name a RSP queue, indicated by an unsuccessful queue open or the presence of a drive specification or password field, the CLI assumes it names a file residing on the default or specified drive. It then attempts to open the file, filename.PRL. If the open is unsuccessful, it tries again with the file, filename.COM. When the current user number is non-zero and the file to be opened does not exist under that user number, the BDOS attempts to open the file under user 0. The open operation is successful if the file exists under user 0, and has the system attribute set.

   If neither open is successful, and no explicit drive reference was made the CLI repeats the same sequence on the MP/M II system drive. (The system drive is designated during system generation) . The CLI does not make this second attempt if the system drive was referenced in the first attempt. In addition, regardless of the file's user number, only files with the system attribute set are accepted in the second open sequence.

   In all cases, if the file password specified in the <command> field does not match the password of a file protected in Read mode, a password error terminates the CLI's open operation.

4) If the command file open is successful, the CLI performs different actions depending on whether the opened file is of type PRL or COM. For PRL files, the CLI selects the smallest available memory segment which can fit the PRL the file. For COM files, the CLI selects the first available absolute memory segment to load the file. Note: More than one absolute memory segment can exist in a bank-switched system.

5) If no memory segment is available, the program loading by the CLI is terminated and the system returns an error message. otherwise, the CLI loads the program into its selected memory segment beginning at BASE+100H (BASE = memory segment base address). If the command file is of the PRL type and the selected memory segment is not absolute, the CLI performs a relocation operation at this time (See Section 1.6).

   The load operation can be aborted if a read error occurs, or in the case of COM files, if the selected memory segment is not large enough to contain the file.

6) Once the program has been loaded, the CLI initializes the memory segment base page beginning at BASE+OOOH. The base page initialization is covered in more detail in Section 2.4.

7) Once the base page is initialized, the CLI sets up a Process Descriptor for the loaded program, and assigns the command file name to the process. The CLI also sets the current disk, user number, list device and console number fields of the Process Descriptor to the values received in the command line message, and gives the process a 20-byte stack. It then initiates the transient program with an XDOS Create Process call. The CLI is then ready to read the next command line and repeat the cycle.

1.6 Transient Programs

Under MP/M II, a transient program is one that the CLI loads and initiates. As the name transient implies, the program is not system resident. The system must load it into an available memory segment every time it executes.

MP/M !I can execute two types of transient programs. The first type, absolute programs, must run in an absolute memory segment. An absolute memory segment is one that has a base address of zero (BASE = OOOOH). The command files of absolute transient programs are identified by a file type field of COM. A COM file contains the absolute memory image of the file beginning at 100H. Thus, the CLI loads a COM file into memory beginning at 100H. MP/M II COM files are equivalent to those in CP/M.

The second type of transient program, Page Relocatable Programs (PRLS), can run in relocatable or absolute memory segments. PRL command files have a type field of PRL. A PRL file contains three regions: a 1-page header, a code region, and a relocation bit map. The header has a field containing the length of the program's code region and a field specifying the minimum amount of additional data space required by the program. The CLI uses this information to select a memory segment for the program. The code region contains the code and initialized data for the program. The CLI loads the code region into memory beginning at BASE+100H, where BASE is the memory segment base address.

The bit map consists of a bit string where each bit corresponds to a byte in the code region. The first bit corresponds to the first byte, the one loaded into BASE+100H. Because the bit map immediately follows the code region in a PRL file, the offset of the bit map equals the program length value stored in the PRL header. Each bit equal to 1 identifies the high byte of an address field that requires relocation. During the program load operation, the CLI adds the high byte or page offset of the address BASE to the bytes identified for relocation by the bit map. This dynamically relocates the program and allows it to run in relocatable memory segments. PRL's loaded into absolute memory segments require no relocation. Note: It is not possible to convert a COM file into a PRL file. However, the reverse operation is possible and is performed with the utility, PRLCOM (see Section 6.3).

As part of the program load operation, the CLI initializes the memory segment base page as follows:

```
BASE+OOOH : Direct XIOS and program termination jump
BASE+005H : BDOS and XDOS function jump
BASE+050H : Command file drive
BASE+051H : Password address of lst file in the command tail
BASE+053H : Password length of lst file in the command tail
BASE+054H : Password address of 2nd file in the command tail
BASE+056H : Password length of 2nd file in the command tail
BASE+05CH : Parsed FCB for lst file in the command tail
BASE+06CH : Parsed FCB for 2nd file in the command tail
BASE+080H : Command tail
```

During execution, a transient program makes BDOS or XDOS system calls by calling BASE+5. Direct XIOS calls are made with the jump at BASE+OOOH. Note: Direct XIOS calls are restricted to console and list I/O. All memory within the segment below the address contained in BASE+6 is available to the transient program. Thus, transient programs can use this address to size memory. The remaining information placed into the base page is data parsed out of the command line. This information is provided as a convenience to the programmer and is covered in detail in Section 2.

When the CLI initiates a transient program, it assigns a 20 byte stack area to the process. This stack is initialized in such a way that if the program returns to the system, its execution is terminated. A process can also terminate execution with a jump to BASE+OOOH, a BDOS System Reset call, or an XDOS Terminate Process call.

1.7 Resident System Processes

Resident System Processes (RSPs) are optional processes that can be included with MP/M II during system generation. There are two types of RSPs: standard and banked. A standard RSP is a page relocatable file that has a filetype of RSP. When integrated into MP/M II, a standard RSP resides in the common memory region. A banked RSP consists of two page-relocatable files, both of which have the same filename but have file type fields of RSP and BRS respectively. When a banked RSP is included in MP/M II, the RSP file loads into common memory, whereas the BRS file loads into memory segment 0 in bank 0. Because all Process Descriptors and queues must reside in common memory, the common module of a banked RSP contains its Process Descriptor and any additional Process Descriptors and queues.

The memory segment field of an RSPs Process Descriptor designates whether the RSP is standard or banked. Standard RSPs set the memory segment field to FFH; banked RSPs set the field to zero. When a RSP is selected during the system generation process, GENSYS checks this field, and if set to 0, includes the BRS file in memory segment 0.

RSPs load into memory as part of the MPMLDR operation, and are initiated following the XIOS System Initialization call and prior to the initialization of the TMPs. Once initiated, an RSP runs like any other process in the system, competing for the CPU and other system resources on a priority basis.

If a RSP is to be invoked as a built-in command from the console command line, it must perform the following steps:

1) Make a queue with a message length sufficiently large to accept the command tail. The name of the queue is the command name of the RSP. Because the CLI converts command lines to upper-case, RSP queue names must be upper-case. If the CLI is to assign the console to the RSP, the RSP's Process Descriptor name must be the same as its queue name.

2) Make an unconditional Read Queue call to the queue. This suspends the RSP on its queue's Enqueue List until the CLI writes it a command line message.

3) Perform its task by making BDOS and XDOS function calls using the command line message containing the current drive, user, list device and system console number obtained from the queue read. Note: An RSP does not make system calls by calling location 5. The system initializes the first two bytes of a standard RSP and the first two bytes of the common module of an extended RSP to contain the system entry point address. The system sets the first two bytes of the bank-zero module of an banked RSP to the beginning address of its corresponding common module. RSP's must use these addresses to make system calls.

4) After performing its task, the RSP must make an XDOS Detach Console call and an XDOS Detach List call if it is assigned the console by the CLI. It then returns to step 2 and awaits another command line.

Another special type of RSP is the Resident System Procedure. A Resident System Procedure provides a method of serially utilizing a block of code as a system resource. A Resident System Procedure is set up by a RSP. The process creates a queue with the name of the Resident System Procedure and sends it a single two-byte message containing the address of the procedure to be accessed serially. once this is accomplished, the RSP terminates itself.

The Resident System Procedure is accessed by opening the queue and reading the two byte message to obtain the actual memory address of the procedure. Because only one message resides in the queue, only one process can gain access to the procedure. When the process leaves the procedure, it writes the message containing the procedure address back to the queue. This enables the next waiting process to use the Resident System Procedure.

1.8 BDOS and XDOS Calling Conventions

MP/M II's BDOS and XDOS system functions can be accessed by both transient programs and RSPs. Transient programs make system calls via the primary entry point at location BASE+005H, where BASE equals the base address of the transient program's memory segment. Standard RSPs obtain the system entry point address from the first two bytes of the program. For banked RSPs, the first two bytes of the common module contain the system entry point address. The first two bytes of the bank-zero module contain the address of the common module.

MP/M II uses a standard protocol for system function calls. It is the same protocol used by CP/M. In general, when making a system call, register C contains the function number, and register pair DE contains the information address. Functions return single-byte values in register A, and double-byte values in register pair HL. Any system call made with an out-of-range or non-supported function number, returns a 0FFFFH in register pair HL. Note: CP/M returns with HL set to 0 on invalid function calls. For compatibility, register A equals L and register B equals H upon return in all cases. The register passing conventions of MP/M II agree with those of Intel's PL/M systems programming language.

When entering a transient program, the system sets the stack pointer to a 10-level stack, with the address contained in BASE+001H pushed onto the stack. Thus, a return to the system is equivalent to a jump to BASE+OOOH. Typically, this stack is sufficiently large to handle system calls. However, most transient programs set up their own stack and return to the system via a jump to location BASE+OOOH. Because of the way RSPs are integrated into the system, they must set up and initialize their own stack.

The programmer should be aware that BDOS and XDOS function calls do not restore registers to their entry values upon return to the calling program. The responsibility for saving and restoring any critical register values rests with the calling process.

As an example, the following transient program illustrates how to make system calls. This program reads characters continuously until it encounters an asterisk, at which time it terminates execution by returning to the system.

```
            ORG   OOOOH
BASE        EQU   $            ;BEGINNING OF MEMORY SEGMENT
BDOS        EQU   BASE+0005H   ;MP/M II SYSTEM ENTRY POINT
CONIN       EQU   1            ;CONSOLE INPUT FUNCTION
;
            ORG   100H         ;BASE OF TRANSIENT PROGRAM AREA
NEXTC       MVI   C,CONIN      ;READ NEXT CHARACTER FUNCTION #
            CALL  BDOS         ;RETURN CHARACTER IN A
            CPI   1*1          ;END OF PROCESSING
            JNZ   NEXTC        ;LOOP IF NOT
            RET                ;TERMINATE PROGRAM
                                 END
```

Section 2
The BDOS Interface

2.1 BDOS Console and List I/O Interface

A primary design objective of MP/M II has been to achieve CP/M compatibility. Thus, from the perspective of the applications program there are only minor differences between CP/M and MP/M II with regard to BDOS console and list I/O functions. These differences are described in Section 2.4, BDOS Function Calls.

Each program executing under MP/M II has a data structure called a Process Descriptor which defines the characteristics of the process. One byte of the Process Descriptor identifies the console and list I/O device numbers currently being used by the process. The high-order 4 bits of this byte, labeled the CONSOLE/LIST field, contain the list device number. The low-order 4 bits contain the console device number. The BDOS console and list I/O functions obtain the appropriate device number from the CONSOLE/LIST field of the Process Descriptor to call the XIOS console or list subroutine.

A process must be attached to a console or list device to access the device. This applies to both BDOS and direct XIOS function calls. MP/M II intercepts all BDOS and direct XIOS function calls for the console and list devices to determine if the specified device is attached to the calling process. The function call is permitted only if the device is currently unattached, or attached to the calling process. If the device is attached to some other process, MP/M II performs an XDOS Attach call for the specified device. The calling process then blocks, suspending execution, until the device is detached from the process owning the device and the calling process is the highest priority process requiring the device. Attaching a specific device to a process can be done explicitly by making XDOS Attach Console or Attach List calls, or implicitly by making BDOS and direct XIOS function calls which in turn force device attachment.

MP/M II maintains tables of processes currently owning the console and list devices. These tables contain Process Descriptor addresses. It is thus possible for one process to own several console or list devices by having its Process Descriptor address in several table entries. Multiple devices can be attached by repeatedly using the XDOS Set Console or Set List Device function call followed by an XDOS Attach call. Later, when actual I/O is to be performed, the specific console or list device must be set in the Process Descriptor by making an appropriate XDOS Set Console or Set List Device function call.

All console and list devices are detached from a process when it terminates, allowing processes that were waiting for the devices to resume execution.

While performing BDOS console I/O functions, there are several ASCII control characters that cause MP/M II to take specific actions. The TC character can abort the process owning the console. The TD character forces the process owning the console to detach from the console, allowing another waiting process to gain access to the console, and then attaches the console again before continuing. The TS and TQ characters are used to stop and re-start console display output. The TS character will cause console display output to be suspended. At that point a TQ can be typed to resume console display output or a TC can be typed to abort the process owning the console. Typing any other key when output has been suspended will cause MP/M II to send the ASCII Bell character (TG) to the console.

2.2 BDOS File System

The Basic Disk Operating System (BDOS) supports a named file system on one to sixteen logical drives. Each logical drive is divided into two regions: a directory area and a data area. The directory area defines the files that exist on the drive and identifies the data area space that belongs to each file. The data area contains the file data defined by the directory. The directory area is subdivided into sixteen logically independent directories, each identified by user numbers 0 through 15. In general, only files belonging to the current user number are "visible" in the directory. For example, the MP/M II DIR utility only displays files belonging to the current user number.

The BDOS file system automatically allocates directory and data area space when a file is created or extended and returns previously allocated space to free space when a file is deleted. If no directory or data space is available for a requested operation, the BDOS returns an error to the calling process. These actions are transparent to the calling process. As a result, the user does not need to be concerned with directory and drive organization when using the file system functions.

An eight-character filename field and a three character type field identifies each file in a directory. An eight character password can also be assigned to a file to protect it from unauthorized access. All BDOS functions that involve file operations specify the requested file by the filename and type fields. Multiple files can be specified by an ambiguous reference. An ambiguous reference uses one or more I'?" marks in the name or type field to indicate that any character matches that position. Thus, a name and type specification of all "?'"s (equivalent to a command line file specification of "*.*") matches all files in the directory that belong to the current user number.

The BDOS file system supports four categories of functions: file access functions, directory functions, drive related functions, and miscellaneous functions. The file access category includes functions to make (create) a new file, open an existing file and close an existing file. Both the make and open functions activate the file for subsequent access by read and write functions. After a file has been opened, subsequent BDOS functions can read or write to the file, either sequentially or randomly by record position. BDOS read and write commands transfer data in 128 byte logical units, which is the basic record size of the file system. The close function performs two steps to terminate access to a file. First, it indicates to the file system that the calling process has finished accessing the file. The file then becomes available to other processes. In addition, the function makes any necessary updates to the directory to permanently record the current status of the file.

BDOS directory functions operate on existing file entries in a drive's directory. This category includes functions to search for one or more files, delete one or more files, rename a file, set file attributes, assign a password to a file, and compute the size of a file. The BDOS search and delete functions are the only functions that allow ambiguous file references. All other directory and file related functions require a specific file reference. The BDOS file system does not allow a process to delete, rename, or set the attributes of a file that is currently opened by another process.

BDOS drive-related functions include those which select a drive as the default drive, compute a drive's free space, interrogate drive status and assign a directory label to a drive. The directory label for a drive controls whether file passwords are to be honored, and the type of date and time stamping to be performed for files on the drive. Also included in this category are functions to reset specified drives and to control whether other processes can reset particular drives. When a drive is reset, the next operation on the drive reactivates it by logging it in. The function of the log-in operation is to initialize the drive for file and directory operations. Under MP/M II, a successful drive reset operation must be performed on drives that support removeable media before changing disks.

Miscellaneous functions include those that set the current DMA address, access and update the current user number, chain to a new program, and flush the internal blocking/deblocking buffer. Also included are functions to set the BDOS multi-sector count and the BDOS error mode. The BDOS multi-sector count determines the number of 128-byte records to be processed by BDOS read, write, record lock, and record unlock functions. It can range from one to sixteen 128-byte records; the default value is one. The BDOS error mode determines whether the BDOS file system intercepts errors or returns all errors to the calling process.

The following list summarizes the operations performed by the BDOS file system:

Disk System Reset
Drive Selection
File Creation
File Open
File Close
Directory Search
File Delete
File Rename
Random or Sequential Read
Random or Sequential Write
Interrogate Selected Disks
Set DMA Address
Set/Reset File Indicators
Reset Drive
Access/Free Drive
Random Write With Zero Fill
Lock and Unlock Record
Set Multi-Sector Count
Set BDOS Error Mode
Get Disk Free Space
Chain To Program
Flush Buffers
Set Directory Label
Return Directory Label
Read and Write File XFCB
Set/Get Date and Time
Set Default Password
Return BDOS Serial Number

The following sections contain information on important topics related to the BDOS file system. The reader should be familiar with the content of these sections before attempting to use the BDOS functions described individually in Section 2.4.

2.2.1 File Naming Conventions

Under MP/M II, filenames consist of four parts: the drive select code (d), the filename field, the file type field, and the file password field. The general format for a command line file specification is shown below:

{d:}filename{.typ} {;password}

The drive select code field specifies the drive where the file is located. The filename and type fields identify the file. The password field specifies the password if a file is password protected.

The drive, type, and password fields are optional and the delimiters " : . ; " are required only when specifying their associated field. The drive select code can be assigned a value from "A" to "P" where the actual drive codes supported on a given system is determined by the XIOS implementation. When the drive code is not specified, the current default drive is indicated. The filename field can contain one to eight non-delimiter characters, the file type field, one to three non-delimiter characters, and the password field, one to eight non-delimiter characters. All alphabetic characters must be in uppercase. In addition, all three fields are padded with blanks, if necessary. Omitting the optional type or password fields implies a field specification of all blanks.

The MP/M II Parse Filename function recognizes certain ASCII characters as valid delimiters when it parses a file from a command line. The valid characters are shown in Table 2-1.

Table 2-1. Valid Filename Delimiters

ASCII	HEX EQUIVALENT
:	3A
.	2E
;	3B
=	3D
,	2C
/	2F
[	5B
]	5D
<	3C
>	3E

The Parse Filename function also excludes all control characters from the file fields and translates all lower-case letters to upper case.

The characters "(" and ")" should be avoided in filename and type fields because they are commonly used delimiters. The characters "*" and "?" must not be used in filename and type fields unless they are used to make an ambiguous reference. If the Parse Filename function encounters a "*" in a file name or type field, it pads the remainder of the field with "?" marks. For example, a filename of "X*.*" is parsed to "X???????.???". The BDOS search and delete functions treat a "?" in the filename and type fields as follows: A "?" in any position matches the corresponding field of any directory entry belonging to the current user number. Thus, a search operation for "X???????.???" finds all the current user files on the directory beginning in "X". Most other file related BDOS functions treat the presence of a "?" in the filename or type field as an error.

It is not mandatory to follow the file naming conventions of MP/M II when creating or renaming a file with BDOS functions. However, the conventions must be used if the file is to be accessed from a command line. For example, the CLI cannot locate a command file in the directory if its filename or type field contains a lower-case letter.

As a general rule, the file type field names the generic category of a particular file, while the filename distinguishes individual files in each category. Although they are generally arbitrary, the file types listed below name some of the generic categories that have been established.

ASM	Assembler Source	PLI	PL/I Source File
PRN	Printer Listing	REL	Relocatable Module
HEX	Hex Machine Code	TEX	TEX Formatter Source
BAS	Basic Source File	BAK	ED Source Backup
INT	Intermediate File	SYM	SID Symbol File
COM	Command File	$$$	Temporary File
PRL	Page Relocatable	RSP	Resident Sys. Process
SPR	Sys. Page Reloc.	SYS	System File
DAT	Data File	BRS	Banked RSP File

2.2.2 Disk Drive and File Organization

The BDOS file system can support from one to sixteen logical drives. The maximum file size supported on a drive is 32 megabytes. The maximum capacity of a drive is determined by the data block size specified for the drive in the XIOS. The data block size is the basic unit in which the BDOS allocates disk space to files. Table 2-2 displays the relationship between data block size and drive capacity.

Table 2-2. Logical Drive Capacity

Data Block Size	Maximum Drive Capacity
1K	256 Kilobytes
2K	64 Megabytes
4K	128 Megabytes
8K	256 Megabytes
16K	512 Megabytes

Logical drives are divided into two regions: a directory area and a data area. The directory area contains from one to sixteen blocks located at the beginning of the drive. The actual number is set in the XIOS. This area contains entries that define which files exist on the drive. The directory entries corresponding to a particular file define which data blocks in the drive's data area belong to the file. These data blocks contain the file's records. The directory area is logically subdivided into sixteen independent directories identified as user 0 through 15. Each independent directory shares the actual directory area on the drive. However, a file's directory entries cannot exist under more than one user number. In general, only files belonging to the current user number are visible in the directory.

Each disk file consists of a set of up to 242,144 128-byte records. Each record in a file is identified by its position in the file. This position is called the record's random record number. If a file is created sequentially, the first record has a position of zero, while the last record has a position one less than the number of records in the file. Such a file can be read sequentially in record position order beginning at record zero, or randomly by record position. Conversely, if a file is created randomly, records are added to the file by specified position. A file created in this way is called "sparse" if positions exist within the file where a record has not been written.

The BDOS automatically allocates data blocks to a file to contain its records on the basis of the record positions consumed. Thus, a sparse file that contains two records, one at position zero, the other at position 242,143, would consume only two data blocks in the data area. Sparse files can only be created and accessed randomly, not sequentially. Note that any data block allocated to a file is permanently allocated to the file until the file is deleted. There is no other mechanism supported by the BDOS for releasing data blocks belonging to a file.

Source files under MP/M are treated as a sequence of ASCII characters, where each "line" of the source file is followed by a carriage-return line-feed sequence (ODH followed by OAH). Thus a single 128-byte record could contain several lines of source text. The end of an ASCII file is denoted by a control-Z character (1AH) or a real end of file, returned by the BDOS read operation. Control-Z characters embedded within machine code files such as COM or PRL files are ignored. The end of file condition returned by BDOS is used to terminate read operations.

2.2.3 File Control Block Definition

The File Control Block (FCB) is a data structure used with the BDOS file access and directory functions. All of these functions reference an FCB to determine the file or files to be operated on. Certain fields in the FCB are also used for invoking special options associated with some functions. other functions use the FCB to return data to the calling process. Most importantly, when a process opens a file and subsequently accesses it with read, write, lock, and unlock record functions, the BDOS file system maintains the current file state and position within the user's FCB. In addition, all BDOS random I/O functions specify the random record number with a 3-byte field at the end of the FCB.

When making a file access or directory BDOS function call, register pair DE must address a FCB. The length of the FCB data area depends on the BDOS function. For most functions, the required length is 33 bytes. For random I/O functions and the Compute File Size function, the FCB length must be 36 bytes. When either the BDOS Open or Make File functions specify a file is to be opened in unlocked mode, the FCB must be 35 bytes in length. The FCB format is shown on the next page.

```
---------------------------------------------------------------
:dr:fl:f2: ... :f8:tl:t2:t3:ex:sl:s2:rc:d0: ... :dn:cr:r0:rl:r2:
---------------------------------------------------------------
 00 01 02  ...  08 09 10 11 12 13 14 15 16  ...  31 32 33 34 35
```

where

```
dr           drive code (0 - 16)
             0 => use default drive for file
             1 => auto disk select drive A,
             2 => auto disk select drive B,
             . . .
             16 => auto disk select drive P.

fl . . .f8   contain the file name in ASCII
             upper case, with high bit = 0.
             f1'. . . f8' denote the high
             order bit of these positions,
             and are file attribute bits.

tl,t2,t3     contain the file type in ASCII
             upper case, with high bit = 0.
             t1', t2' and t3'denote the
             high bit of these positions,
             and are file attribute bits.
             tl' = 1 => Read/Only file
             t2' = I => System file
             t3' = 1 => File has been archived

ex           contains the current extent number,
             normally set to 0 by the calling process, but
             can range 0 - 31 during file I/O

cs           contains the FCB checksum value for
             open FCBs.

rs           reserved for internal system use, set
             to zero on call to OPEN, MAKE, SEARCH

rc           record count for extent "ex"
             takes on values from 0 - 128

dO    dn     filled-in by MP/M, reserved for
             system use

cr           current record to read or write in
             a sequential file operation, normally
             set to zero by the calling process when a
             file is opened or created

rO,rl,r2     optional random record number in the
             range 0-242,143 (0 - 3FFFFH).
             ro,rl,r2 constitute a 18 bit value
             with low byte rO, middle byte rl, and
             high byte r2.
```

Note : The 2-byte File ID is returned in bytes r0 and r1 when a file is successfully opened in unlocked mode (see Section 2.2.8).

For BDOS directory functions, the calling process must initialize bytes 0 through 11 of the FCB before issuing the function call. The Set Directory Label and Write File XFCB functions also require the calling process to initialize byte 12. The BDOS Rename File function requires the calling process to place the new file name and type in bytes 17 through 27.

BDOS open or make function calls require the calling process to initialize bytes 0 through 12 of the FCB before issuing a file open or make function call. Normally, byte 12 is set to zero. In addition, if the file is to be processed from the beginning using sequential read or write functions, byte 32 (cr) must be zeroed. After an FCB is activated by an open or make operation, the FCB should not be modified by the user. Open FCBs are checksum verified to protect the integrity of the file system. In general, if a process modifies an open FCB, the next read, write, or close function call will return with a checksum error. See Section 2.2.9 for more on FCB checksums. Normally, sequential read or write functions do not require initialization of an open FCB. However, random I/O functions require that a process set bytes 33 through 35 to the requested random record number prior to making the function call.

File directory elements maintained in the directory area of each disk drive have the same format as FCB's (excluding bytes 32 through 35), except for byte 0 which contains the file's user number. Both the Open File and Make File functions bring these elements (excluding byte 0) into memory in the FCB specified by the calling process. All read and write operations on a file must specify an FCB activated in this manner. Otherwise, a checksum error is returned. The BDOS updates the memory copy of the FCB during file processing to maintain the current position within the file. During file write operations, the BDOS updates the memory copy of the FCB to record the allocation of data to the file, and at the termination of file processing, the Close File function permanently records this information on disk. Note that data allocated to a file during file write operations is not completely recorded in the directory until the calling process issues a Close File call. Therefore, it is mandatory that a process which creates or modifies files, close the files at the termination of any write processing, otherwise, data may be lost.

As a general rule under MP/M II, a process should close files as soon as they are no longer needed, even if they have not been modified. The BDOS file system maintains an entry in the system lock list (LCKLSTS.DAT memory segment) for each file opened by each process on the system. This entry is not removed from the system lock list until the file is closed or the process owning the entry terminates. The BDOS file system uses this entry to prevent other processes from accessing the file unless the file was opened in a mode that supports shared access. Normally, a process must close a file before other processes on the system can access the file.

Keep in mind that the space in the system lock list is a limited resource under MP/M II. If a process attempts to open a file and no space exists in the system lock list, or the process exceeds the process open file limit (specified during system generation), the BDOS denies the open operation and usually aborts the calling process.

The high-order bits of the FCB filename (fl',...,f8’) and type (tl',t2',t3') fields are called attribute bits. Attributes bits are 1 bit boolean fields where 1 indicates on or true, and 0 indicates off or false. Attribute bits have two functions within the file system: as file attributes and interface attributes.

The file attributes (fl',...,f4' and tl',t2',t3') are used to indicate that a file has a defined attribute. These bits are recorded in a file's directory FCBs. File Attributes can only be set or reset by the BDOS Set File Attributes function. When the BDOS Make File function creates a file, it initializes all file attributes to zero. A process can interrogate file attributes in an FCB activated by the BDOS Open File function or in directory FCBs returned by the BDOS Search For First and Search For Next functions. Note: the BDOS file system ignores the file attribute bits when it attempts to locate a file in the directory.

The file attributes (tl’,t2',t3') are defined by the file system as follows:

t1’: Read/Only attribute - The file system prevents write operations to a file with the read/only attribute set.

t2’: System Attribute - This attribute, if set, identifies the file as a MP/M II system file. System files are not normally displayed by the MP/M II DIR utility. In addition, user zero system files can be accessed on a read/only basis from other user numbers (see Section 2.2.8).

t3’: Archive Attribute - This attribute is designed for user written archive programs. When a archive program copies a file to backup storage, it sets the archive attribute of the copied files. The file system automatically resets the archive attribute of a directory FCB that has been issued a write command. The archive program can test this attribute in each of the file's directory FCBs via the BDOS Search and Searchn functions. If all directory FCBs have the archive attribute set, it indicates that the file has not been modified since the previous archive. Note that the MP/M II PIP utility supports file archival.

Attributes fl' through f8’ are available for definition by the user.

The interface attributes are f5' through f8'. These attributes cannot be used as file attributes. Interface attributes f5’ and f6’ are used to request options for BDOS calls requiring an FCB address in register pair DE. They are used by the BDOS Open, Make, Close, and Delete File functions. Table 2-3 shows the f5' and f6' interface attribute definitions for these functions.

Table 2-3. BDOS Interface Attributes

open function	f5' = 1 : Open in unlocked mode f6' = 1 : Open in read/only mode
Make function	f5' = 1 : Open in unlocked mode f6' = 1 : Assign password to file
Close function	f5' = 1 : Partial Close
Delete function	f5' = 1 : Delete file XFCBs only

The interface attributes are discussed in detail for each of the above functions in Section 2.4. Attributes f5' and f6' are always reset when control is returned to the calling process. Interface attributes f7' and f8' are reserved for internal use by the BDOS file system.

The BDOS search and delete functions allow multiple file (ambiguous) reference. In general, a ? mark in the filename, type, or extent field matches any value in the corresponding positions of directory FCBs during a directory search operation. The BDOS search functions also recognize a ? mark in the drive code field, and if specified, they return all directory entries on the disk regardless of user number including empty entries. A directory FCB beginning with E5H is an empty directory entry.

2.2.4 User Number Conventions

The MP/M II User facility divides each drive directory into sixteen logically independent directories, designated as user 0 through user 15. Physically, all user directories share the directory area of a drive. In most other aspects, however, they are independent. For example, files with the same name can exist on different user numbers of the same drive with no conflict. However, a single file cannot reside under more than one user number.

Only one user number is active for a process at one time, and the current user number applies to all drives on the system. Furthermore, the FCB format does not contain any field that can be used to override the current user number. As a result, all file and directory operations reference directories associated with the current user number. However, it is possible for a process to access files on different user numbers by setting the user number to the file's user number with the BDOS Set User command prior to issuing the desired BDOS function call for the file. Note that this technique must be used carefully. If a process attempts to read or write to a file under a user number that is not the same as the user number that was active when the file was opened, the BDOS file system returns a FCB checksum error.

When the CLI initiates a transient program or RSP, its user number is set to the value established by the process issuing the XDOS Send Cli Command. Normally, the sending process is the TMP. However, the sending process may be another process such as a transient program that makes a BDOS Chain Program call. A transient program can change its user number by making a BDOS set user call. Changing the user number in this way does not affect the command line user number displayed by the TMP. Thus, when a transient program that has changed its user number terminates, the original user number for the console is restored when the TMP regains control.

User 0 has special properties under MP/M II. With some restrictions, the file system automatically opens a file under user zero, if it is not present under the current user number. Of course, this action is only performed when the current user number is not zero. In addition, a file on user zero must have the system attribute (t2') set to be eligible for this operation. This procedure allows utilities that may include overlays and any other commonly accessed files to be placed on user zero, but be available for access from other user numbers. As a result, it eliminates the need for copying commonly needed utilities to all user numbers on a directory, and gives the MP/M II user control over which user zero files are directly accessible from other user numbers. Refer to Section 2.2.8 for more information on this topic.

2.2.5 Directory Labels and XFCBs

The BDOS file system includes two special types of FCB's, the XFCB and the Directory Label. The XFCB is an "extended" FCB that can optionally be associated with a file in the directory. if present, it contains the file's password field and date and time stamp information. The format of the XFCB is shown below:

```
XFCB FORMAT
-----------------------------------------------------
:dr: file : type :pm:sl:s2:rc: password : tsl : ts2 :
:---------------------------------------------------:
:00  01. .  09. . 12 13 14 15  16. . .    25.   29. :
-----------------------------------------------------

      dr    drive code (0 - 16)
            file  file name field
            type  file type field
      pm    password mode
            bit 7 - Read mode
            bit 6 - Write mode
            bit 5 - Delete mode
            **    - bit references are right to left,
                    relative to 0
      sl,s2,rc    - reserved for system use
      password    - 8 byte password field (encrypted)
      tsl         - 4 byte creation or access time stamp field
      ts2         - 4 byte update time stamp field
```

An XFCB can be created for a file in two ways: automatically, as part of the BDOS Make function or explicitly, by the BDOS function, Write File XFCB. The BDOS file system does not automatically create an XFCB for a file unless a Directory Label is present on the file's drive. The BDOS Read File XFCB function returns a file's XFCB if it exists in the directory. Note that in the directory, an XFCB is identified by a drive byte value (byte 0 in the FCB) equal to 16 + N, where N equals the user number.

The Directory Label specifies for a drive if passwords for password protected files are to be required, if date and time stamping for files is to be performed, and if XFCBs are to be created automatically for files by the Make function. The format of the Directory Label is similar to that of the XFCB as shown below:

DIRECTORY LABEL FORMAT

```
 ----------------------------------------------------
 :dr: name : type :dl:sl:s2:rc: password : ts : ts2 :
 :--------------------------------------------------:
 :00   01. . 09 .. 12 13 14  15  16....    25.  29.  :
 :--------------------------------------------------:
```

```
dr        -   drive code (0 - 16)
name      -   Directory Label name
type      -   Directory Label type
dl        -   Directory Label data byte
              bit 7 - require passwords for files
              bit 6 - perform access time stamping
              bit 5 - perform update time stamping
              bit 4 - Make creates XFCBs
              bit 0 - Directory Label exists
              **    - bit references are right to left,
                      relative to 0
sl,s2,rc -   n/a
password -   8 byte password field (encrypted)
tsl       -  4 byte creation time stamp field
ts2       -  4 byte update time stamp field
```

Only one Directory Label can exist in a drive's directory. The Directory Label name and type fields are not used to search for a Directory Label in the directory; they can be used to identify a diskette or a drive. A Directory Label can be created or its fields can be updated by the BDOS function, Set Directory Label. This function can also assign a Directory Label a password. The Directory Label password, if assigned, cannot be circumvented, whereas file password protection is an option controlled by the Directory Label. Thus, access to the Directory Label password provides a kind of super-user status for that drive.

<u>Note:</u> The BDOS file system provides no function to read the Directory Label FCB directly. However, the Directory Label data byte can be read directly with the BDOS function, Return Directory Label. In addition, the BDOS Search functions (‘?' in FCB drive

byte) can be used to find the Directory Label on the default drive. In the directory, the Directory Label is identified by a drive byte value (byte 0 in the FCB) equal to 32 (20H).

2.2.6 File Passwords

Files may be assigned passwords in two ways: by the Make File function if the Directory Label specifies automatic creation of XFCBs or by the Write File XFCB function. A file's password can also be changed by the Write File XFCB function if the original password is supplied. However, a file's password cannot be changed without the original password even when password protection for the drive is disabled by the Directory Label.

Password protection is provided in one of three modes. Table 2-4 shows the difference in access level allowed to BDOS functions when the password is not supplied.

Table 2-4. Password Protection Modes

Password Mode	Access level allowed when the password is not supplied
1. Read	The file cannot be read
2. Write	The file can be read but not modified.
3. Delete	The file can be modified but not deleted.

If a file is password protected in Read mode, the password must be supplied to open the file. A file protected in Write mode cannot be written to without the password. A file protected in Delete mode allows read and write access, but the user must specify the password to delete the file, rename the file, or to modify the file's attributes. Thus, password protection in mode 1 implies mode 2 and 3 protection, and mode 2 protection implies mode 3 protection. All three modes require the user to specify the password to delete the file, rename the file, or to modify the file's attributes.

If the correct password is supplied, or if password protection is disabled by the Directory Label, then access to the BDOS functions is the same as for a file that is not password protected. In addition, the Search For First and Search For Next functions are not affected by file passwords.

Table 2-5 lists the BDOS functions that test for password.

Table 2-5. BDOS Functions That Test For Password

15.	Open File
19.	Delete File
23.	Rename File
30.	Set File Attributes
100.	Set Directory Label
103.	Write File XFCB

File passwords are eight bytes in length. They are maintained in the XFCB and Directory Label in encrypted form. To make a BDOS function call for a file that requires a password, a process must place the password in the first eight bytes of the current DMA or specify it with the BDOS function, Set Default Password, prior to making the function call. Note: the BDOS maintains the assigned default password on a system console basis and retains it across process termination.

2.2.7 File Date and Time Stamps

The BDOS file system can record when a file was created or last accessed, and/or last updated. It records the creation stamp only when an XFCB is automatically created by the Make File function. If an XFCB is created by the Make File XFCB function, the creation stamp is set to zero. The Close File function makes the update stamp if a write operation is made to the file while the file is open. The Open File function makes the access stamp if the file is successfully opened. The creation date stamp is overwritten when access stamping is performed because only two date and time fields reside in the XFCB and the access and creation time stamps share the same field.

The drive's Directory Label determines the type of date and time stamping supported for files on a drive. If a drive does not have a Directory Label, or if it is read/only, or if the drive's directory label does not specify date and time stamping, then no date and time stamping for files is performed. In addition, a file must have an XFCB to be eligible for date and time stamping. For the Directory Label itself, time stamps record when it was created and last updated. No access stamping for Directory Labels is supported.

A process can directly access the date and time stamps for a file by using the Read File XFCB function. No mechanism is provided to directly update XFCB date and time fields.

The BDOS file system uses the MP/M internal date and time when it records a date and time stamp. On MP/M II systems that do not support a clock, date and time stamps record the last initialized value for the system date and time. The MP/M II TOD utility can be used to set the system date and time.

2.2.8 File Open Modes

The BDOS file system provides three different modes of opening files. They are defined as follows:

locked mode:

A process can open a file in locked mode only if the file is not currently opened by another process. once open in locked mode, no other process can open the file until it is closed. Thus, if a process successfully opens a file in locked mode, that process in effect owns the file until the file is closed or the process terminates. Files opened in locked mode support read and write operations unless the file is a read/only file (attribute tl' set) or the file is password protected in Write mode and the password is not supplied with the BDOS Open File call. In both of these cases, only read operations to the file are allowed. Note: locked mode is the default mode for opening files under MP/M II.

unlocked mode:

A process can open a file in unlocked mode if the file is not currently open, or if the file has been opened by another process in unlocked mode. This mode allows more than one process to open the same file. Files opened in unlocked mode support read and write operations unless the file is a read/only file (attribute tl' set) or the file is password protected in Write mode and the password is not supplied with the BDOS Open File call. However, when a file opened in unlocked mode is extended by a write operation, the BDOS allocates space to the file in data block units, not in 128 byte record units as is normally the case. The BDOS record locking and unlocking functions are only supported for files opened in unlocked mode.

When opening a file in unlocked mode, a process must reserve 35 bytes in the FCB, because the Open File function returns a 2-byte value called the File ID in the rO and rl bytes of the FCB. The File ID is a required parameter for the BDOS record lock and record unlock commands.

read/only mode:

A process can open a file in read/only mode if the file is not currently opened by another process, or the file has been opened by another process in read/only mode. This mode allows more than one process to open the same file for read/only access.

The open function performs the following steps for files opened in locked or read/only mode. If the current user is non-zero, and the file to be opened does not exist under the current user number, the open function searches user zero for the file. If the file exist, under user zero and the file has the system attribute (t2') set, the file is opened under user zero. The open mode is automatically forced to read/only when this is done. For more information on this, refer to Section 2.2.4.

The open function also performs the following action for files opened in locked mode when the current user number is zero. If the file exists under user zero and has the system (t2’) and read/only (tl') attributes set, the open mode is automatically set to read/only. Thus, the read/only attribute controls whether a user zero system file can be concurrently opened by a user-zero process and processes on other user numbers when each process opens the file in the default locked mode. If the read/only attribute is set, all processes open the file in read/only mode and concurrent access of the file is allowed. However, if the read/only attribute is reset, the user-zero process opens the file in locked mode. if it successfully opens the file, no other process can open it. if another process has the file open, its open operation is denied.

Table 2-6 shows the definition of the FCB interface attributes f5' and f6’ for the BDOS Open File function.

Table 2-6. FCB Interface Attributes F5’ F6’
Open File Function

f5' = 0,	f6’ = 0	- open in locked mode (default mode)
f5' = 1,	f6’ = 0	- open in unlocked mode
f5' = 0 or 1,	f6’ = 1	- open in read/only

Interface attribute f5’ designates the open mode for the BDOS Make File function. Table 2-7 shows the definition of the FCB interface attribute f5' for the Make File function.

Table 2-7. FCB Interface Attribute F6’
Make Function

f5' = 0	- open in locked mode (default mode)
f5’ = 1	- open in unlocked mode

Note: the Make File function does not allow opening the file in read/only mode.

2.2.9 File Security

In general, the security measures implemented in the BDOS file system are intended to prevent accidental collisions between running processes. It is not possible to provide total security under MP/M II because the BDOS file system maintains file allocation information in open FCBs in the user's memory region, and MP/M II does not support memory protection. In the worst case, a program that "crashes" on MP/M II can take down the entire system. Therefore, MP/M II requires that all processes running on the system be "friendly." However, the BDOS file system is designed to ensure that multiple processes can share the same file system without interfering with each other. It does this in two ways:

- it performs checksum verification of open FCB's.
- it monitors all open files and locked records via the system lock list (LCKLSTS.DAT).

User FCBs are checksum validated before I/O operations to protect the integrity of the file system from corrupted FCBs. The Open File and Make File functions compute and assign checksums to FCBs. The Read, Write, Lock Record, Unlock Record and Close File functions subsequently verify and recompute the checksums when the FCB changes. If the BDOS detects an FCB checksum error, it does not perform the requested command. Instead, it either terminates the calling process with an error, or if the process is in BDOS return error mode (see Section 2.2.13), it returns to the process with an error code.

The system lock list is established during the system generation process at which time the user can establish the size of the list and also define limits for the number of files a single process can open and the number of records a single process can lock. Each time a process opens a file or locks a record successfully, the BDOS file system allocates an entry in the system lock list to record the fact. The file system uses this information to:

- prevent a process from deleting, renaming, or updating the attributes of another process's open file.
- prevent a process from opening a file currently opened by another process unless both processes open the file in locked or read/only mode.
- prevent a process from resetting a drive on which another process has an open file.
- prevent a process from locking or updating a record currently locked by another process. Refer to Section 2.2.10 for more information on record locking and unlocking.

For reasons of efficiency, the file system verifies only for certain functions whether another process has the FCB specified file open. These functions are: Open File, Make File, Delete File, Rename File, and Set File Attributes. For open FCBs, the FCB checksum controls whether the process can use the FCB. By definition, a valid FCB checksum implies that the file has been successfully opened and an entry for the file resides in the system lock list. When a process closes a file permanently, the file system removes the file from the system lock list and invalidates its FCB checksum field.

There are several other situations where the file system removes open file entries from the system lock list for a process. For example, if a process makes a delete call for a file that it has open in locked mode, the file system deletes the file and also removes the file's entry from the system lock list. Deleting an open file is not recommended practice under MP/M but is supported for files opened in locked mode (the default open mode) , to provide compatibility with software written under earlier releases of MP/M and CP/M. Note that the file system does not delete a file opened in unlocked or read/only mode.

To ensure that the process does not use the FCB corresponding to the deleted file, the file system subsequently checks all open FCBs for the process to ensure that a lock list item exists for the FCB. Each open FCB is checked the next time it is used. If a lock list entry exists for the file, the operation is allowed to proceed. Otherwise, a FCB checksum error is returned.

The file system performs this verification of open FCBs for all situations where it purges an open file entry from the system lock list. The following list describes these situations:

- A process deletes a file it has open in locked mode.
- A process renames a file it has open in locked mode.
- A process updates the attributes via the BDOS Set File Attributes command of a file it has open in locked mode.
- A process issues a Free Drive call for a drive on which it has an open file.
- A change in media is detected on a drive that has open files. This situation is a special case because a process cannot control whether it occurs and it can impact more than one process. Refer to Section 2.2.13 for more information on this situation.

The automatic verification of open FCBs by the file system after it purges a file entry from the system lock list can affect performance. Each verification requires a directory search operation. Therefore, it is strong y recommended that these situations be avoided in new programs developed for MP/M II.

2.2.10 Concurrent File Access

More than one process can access the same file if each process opens the file in the same shared access mode. BDOS supports two shared access modes, unlocked and read/only. Read/only mode is functionally identical to the default locked mode except that more than one process can access the file and no process can change it.

Files opened in unlocked mode present a more complex situation because a file opened in this mode can be modified by multiple processes concurrently. As a result, unlocked mode differs in some important ways from the other open modes.

When a process opens a file in unlocked mode, the file system returns a 2-byte field called the File ID in the r0 and rl bytes of the FCB. The File ID is a required parameter of the BDOS Lock Record and Unlock Record functions.

The file system supports two mechanisms that allow processes to coordinate update operations on files open in unlocked mode. The record locking and unlocking functions allow a process to establish and relinquish temporary ownership of particular records. A record lock does not prevent another process from reading the locked record; only write and lock operations for other processes are intercepted. As an alternative, the Test and Write function verifies the current contents of a record before allowing the write operation to proceed.

The Record locking and unlocking functions and the Test and Write function provide two fundamentally different approaches to record update coordination. When a record is locked, the file system allocates an entry in the system lock list, identifying the locked record and associating it with the calling process. The Unlock Record function removes the locked entry from the list. While the locked record's entry exists in the system lock list, no other process can lock or write to that record. Because the system lock list is a limited resource under MP/M, a process is restricted regarding the number of records it can lock.

The Test and Write function, on the other hand, performs its verification at the I/O level. In a single indivisible operation, it verifies that the user's current version of the record matches the version on disk before allowing the write operation to proceed. As a result, it is not restricted like the Record Lock function. However, record update coordination can usually be performed more efficiently with the lock functions.

The BDOS file system performs additional steps for read and write operations to a file open in unlocked mode. These added steps are required because the BDOS file system maintains the current state of an open file in the user's FCB. When multiple processes have the same file open, FCBs for the same file exist in each processes' memory. To ensure that all process' have current information, the file system updates the directory immediately when an FCB for an unlocked file is changed. In addition, the file system verifies error situations such as end of file or reading unwritten data with the directory before returning an error. As a result, read and write operations are less efficient for files open in unlocked mode when compared to equivalent operations for files opened in the default locked mode.

Extending a file is also a special situation for files opened in unlocked mode. Normally, when a file is extended, the size of the file is set to the random record number of the last record + 1. However, when a file opened in unlocked mode is extended, the size of the file is set to the random record number + 1 of the last 128 byte record in the file's last data block. A process must keep track of the actual last record of a file extended while open in unlocked mode, if that is required.

2.2.11 Multi-Sector I/O

The BDOS file system provides the capability to read or write multiple 128-byte records in a single BDOS function call. This multi-sector facility can be visualized as a BDOS "burst" mode, enabling a process to complete multiple I/O operations without interference from other running processes. The use of this facility in an application program can improve its performance, and also enhance overall system throughput. For example, the PIP utility performs its sequential I/O with a multi-sector count of 8. Multi sector I/O has its greatest impact, however, in the performance of sequential I/O processing on MP/M II systems that support record blocking/deblocking in their XIOS. Improved performance is achieved by eliminating the need for a large percentage of XIOS physical record pre-read operations.

The number of records that can be supported with multi-sector I/0 ranges from one to sixteen. For transient programs, the default value is one because the CLI initializes the multi-sector count of a transient program to one when it initiates the program. The BDOS Set Multi-Sector Count function can be used to set the count to another value.

The multi-sector count determines the number of operations to be performed by the following BDOS functions:

- o Sequential Read and Write functions
- o Random Read and Write functions including Write with Zero Fill and Test and Write
- o Record Lock and Record Unlock functions

If the multi-sector count is N, calling one of the above functions is equivalent to making N function calls. If a multi-sector I/O operation is interrupted with an error, the file system returns the number of 128-byte records successfully processed in the high-order nibble of register H.

2.2.12 XIOS Blocking and Deblocking

An optional physical record blocking and deblocking facility can be implemented as part of the XIOS when it is necessary to maintain physical records on disk in units greater than 128-bytes.

In general, record blocking and deblocking in the XIOS is transparent to the BDOS file system as well as to programs that make BDOS file system calls.

If this facility is implemented, then the XIOS sends data to or receives data from the BDOS file system in logical 128-byte records, but accesses the disk with a larger physical record size. The XIOS uses an internal physical record buffer equal in size to the physical record size to buffer logical records. The process of building up physical records from 128-byte logical records is called blocking, and it is required for BDOS write operations. The reverse process is called deblocking and it is required for BDOS read operations. For BDOS write operations, the XIOS postpones the physical write operation for permanent drives (see Section 2.2.13) if the write operation is not to the directory. For BDOS read operations, the XIOS performs a physical read only if the current physical record buffer does not contain the requested logical record. In addition, if the physical record is "pending" as the result of a previous write operation, the XIOS performs a physical write operation prior to the read operation.

Postponing physical record write operations has implications for some application programs. For those programs that involve file updating, it is often critical to guarantee that the state of a file on disk parallels the state of the file in memory after updating the file. This is only an issue for systems that implement blocking and deblocking because of the postponement of physical write operations. If the system should crash while the physical buffer is pending, data would be lost. To prevent this, the BDOS Flush Buffers function can be invoked to force the write of any pending physical buffers in the XIOS.

Note: The XDOS automatically calls this function when a process terminates. In addition, the BDOS file system automatically makes a Flush Buffers call in the Close File function.

2.2.13 Reset, Access and Free Drive

The BDOS functions Disk System Reset, Reset Drive, Access Drive, and Free Drive allow a process to control when a drive's directory is to be reinitialized for file operations. When MP/M II is initiated by MPMLDR, all drives are initialized to the reset state . Subsequently, as drives are referenced, they are automatically logged-in by the file system. The log-in operation initializes the drive for BDOS file operations. In general, once a drive is logged-in, it is not necessary to relog the drive unless a disk media change is to be made. However, MP/M II requires that a successful drive reset be performed for a drive before a media change. If a drive is in the reset state when the media is changed, the next access to the drive logs in the drive. Note that the Disk System Reset and Reset Drive functions have similar effects except that the Disk System Reset function is directed to all drives on the system. The user can specify any combination of drives to be reset with the Reset Drive function.

Under MP/M II, the drive reset operation is conditional in nature. Generally speaking, the file system cannot reset a drive for a process if another process has an open file on the drive. However, the exact action taken by a drive reset operation depends on whether the drive to be reset is permanent or removeable. MP/M II determines whether a drive is permanent or removeable by interrogating a bit in the drive's disk parameter block (DPB) in the XIOS (refer to the MP/M II System's Guide for a detailed discussion of the DPB) . A high-order bit of 1 in the DPB checksum vector size field designates the drive as permanent. Under MP/M II, a drive's designation is critical to the reset operation, which is described below.

The BDOS first determines if there are any files currently open on the drive to be reset. If there are none, the reset takes place. otherwise, if the drive is a permanent drive and if the drive is not read/only, the reset operation is not performed but a successful result is returned to the calling process. However, if the drive is removeable or read/only, the file system determines whether other processes have open files on the drive. If they do, the drive reset operation is denied and an error code is returned to the calling process. If all the files open on the drive belong to the calling process, the file system performs a "qualified" reset operation for the drive and returns a successful result to the calling process. This means that the next time the drive is accessed, the log-in operation is only performed if a media change is detected on the drive. The logic flow of the drive reset operation is shown in Figure 2-1.

If the file system detects a media change on a drive after a qualified reset, it purges all open files on the drive from the system lock list and subsequently verifies all open FCBS in file operations for the owning process (see Section 2.2.9) . The drive is also relogged-in. In all other cases where a media change is detected on a drive, the file system performs the following steps: All open files on the drive are purged from the system lock list, and all process owning a purged file are flagged for automatic open FCB verification. The drive is then placed in read/only status. It is not relogged-in until a drive reset is issued for the drive. Note: If a process references a file purged from the system lock list in a BDOS command that requires an open FCB, it is returned an FCB checksum error by the BDOS file system.

The Access Drive and Free Drive functions perform special actions under MP/M II. The Access Drive function inserts a "dummy" open file item into the system lock list for each specified drive. While that item exists in the system lock list, the drive cannot be reset by another process. The Free Drive function purges the open lock list of all items including open file items belonging to the calling process on the specified drives. Any subsequent reference to those files by a BDOS function call requiring an open FCB results in a FCB checksum error return.

The BDOS function Write Protect Disk function has special properties under MP/M II. This function can be used to set the specified drive to read/only. However, MP/M II does not allow a process to set a drive read/only if another process has an open file on the drive. This applies to both removeable and permanent drives. If a process has successfully set a drive read/only, it can prevent other processes from resetting the drive by either opening a file on the drive or issuing an Access Drive call for the drive. While the open file or "dummy" item belonging to the process resides in the system lock list, no other process can reset the drive to take it out of read/only status.

```
+------------+
: open files : yes
: on drive ? :--------+
+------------+        :
       :              :
       : no           :
       :              :
       :              :
       :      +---------------+
       :      : Drive         : yes
       :      : removeable ?  :----------+
       :      +---------------+          :
       :             : no                :
       :      +---------------+ yes      :
       :      : Drive R/O ?   :----------+
       :      +---------------+          :
       :             : no                :
       :             :                   :
+----------+  +---------------+    +-------------+
:   Reset  :  : don't reset   :    : Open files  :  yes
:   drive  :  : drive         :    : belong to   :---------+
+----------+  +---------------+    : another     :         :
      :              :             : process ?   :         :
      :              :             +-------------+         :
      :              : no                : no              :
      :              :             +-------------+         :
      :              :             : Qualified   :         :
      :              :             : reset       :         :
      :              :             : performed   :         :
      :              :             +-------------+         :
      :              :                   :                 :
+----------+         :                   :           +----------+
:  Disk    :         :                   :           : Disk     :
: Reset    :---------+-------------------+           : Reset    :
: success  :                                         : Denied   :
+----------+                                         +----------+
```

Figure 2-1. Disk System Reset

2.2.14 BDOS Error Handling

The BDOS file system has an extensive error handling capability. When it detects an error, it can respond in three ways:

1) It can return to the calling process with return codes in register A, H, and L identifying the error.

2) It can display an error message on the console and abort the process.

3) It can display an error message on the console and return to the calling process as in method 1.

The file system handles the majority of errors it detects via method 1. The kinds of errors the file system handles via methods 2 and 3 are called "physical" and "extended" errors. The BDOS Set Error Mode function determines how the file system handles physical and extended errors. The BDOS Error Mode can exist in three states. In the default state, the BDOS displays the error message and terminates the calling process (method 2). In return error mode, the BDOS returns control to the calling process with the error identified in registers A, H, and L (method 1) . In return and display mode, the BDOS returns control to the calling process with the error identified in registers A, H, and L, and also displays the error message at the console (method 3). Both the return modes ensure that MP/M II does not terminate the process because of a physical or extended error. The return and display mode also allows the calling process to take advantage of the built-in error reporting of the BDOS file system. Physical and extended errors are displayed on the console in the following format:

```
BDOS Err on d: error message
BDOS function: nn File: filename.type
```

where "d" is the name of the drive selected when the error condition is detected; "error message" identifies the error; "nn" is the BDOS function number, and "filename.type" identifies the file specified by the BDOS function. If the BDOS function did not involve a FCB, the file information is omitted.

The BDOS physical errors are identified by the following error messages:

- Bad Sector
- Select
- File R/O
- R/O

The "Bad Sector" error results from an error condition returned to the BDOS from the XIOS module. The file system makes XIOS read and write calls to execute file related BDOS calls. If the XIOS read or write routine detects an error, it returns an error code to the BDOS resulting in this error.

The "Select" error also results from an error condition returned to the BDOS from the XIOS module. The BDOS makes an XIOS Select Disk call prior to accessing a drive to perform a requested BDOS function. If the XIOS does not support the selected disk, it returns an error code resulting in this error.

The "File R/O" error is returned whenever a process makes a write operation to a file with the R/O attribute set.

The 11R/01' error is returned whenever a process makes a write operation to a disk that is in read/only status. A drive can be placed in read/only status explicitly with the BDOS Write Protect Disk function, or implicitly if the file system detects a change in media on the drive.

The BDOS extended errors are identified by the following error messages:

- File opened in Read/only Mode
- File Currently Opened
- . Close Checksum Error
- Password Error
- File Already Exists
- Illegal ? in FCB
- Open File Limit Exceeded
- No Room in System Lock List

The "File Opened in Read/only Mode" error is returned when a process attempts to write to a file opened in read/only mode. A file can be opened in read/only mode explicitly, or opened in read/only mode implicitly in two ways. If a file is opened from user zero when the current user number is non-zero, the file is opened in read/only mode. In addition, if a file is password protected in write mode and the password is not supplied with the open call, this error is returned if an attempt is made to write to the file.

The "File Currently open" error is returned when a process attempts to delete, rename, or modify the attributes of a file opened by another process. This error is also returned when a process attempts to open a file in a mode incompatible with the mode in which the file was opened by another process.

The "Close Checksum Error" message is returned when the BDOS detects a checksum error in the FCB passed to the file system with a BDOS Close File call.

The "File Password" error is returned when the file password is not supplied, or it is incorrect.

The "File Already Exists" error is returned for the BDOS Make File and Rename File functions when the BDOS detects a conflict on file name and type.

The "Illegal ? in FCB" error is returned whenever the BDOS detects a "?" in the file name or type field of the passed FCB for the BDOS Rename File, Set File Attributes, Open File, and Make File functions.

The "Open File Limit Exceeded" error is returned when a process exceeds the file lock limit specified in the system lock table during system generation. The Open File, Make File, and Access Drive functions can return this error.

The "No Room in System Lock List" error is returned when no room for new entries exists within the system lock list. The capacity of the system lock list is a system generation parameter. The Open File, Make File, and Access Drive functions can return this error.

The following paragraphs describe the error return code conventions of the BDOS file system functions. Most BDOS file system functions fall into three categories in regard to return codes; they return an Error Code, a Directory Code, or an Error Flag. The error conventions are designed to allow programs written for earlier versions of CP/M and MP/M to run without modification

The following BDOS functions return an Error Code in register A.

20. Read Sequential
21. Write Sequential
33. Read Random
34. Write Random
40. Write Random w/Zero Fill
41. Test and Write Record
42. Lock Record
43. Unlock Record

The Error Code definitions for register A are shown in Table 2-8.

Table 2-8. BDOS Error Codes

```
  00 : Function successful
 255 : Physical error : refer to register H
  01 : Reading unwritten data
       No available directory space (Write Sequential)
  02 : No available data block
  03 : Cannot close current extent
  04 : Seek to unwritten extent
  05 : No available directory space
  06 : Random record number out of range
  07 : Record match error (Test and Write)
* 08 : Record locked by another process
       (restricted to files opened in unlocked mode)
  09 : Invalid FCB (previous BDOS read or write call
       returned an error code and invalidated the FCB)
  10 : FCB checksum error
* 11 : Unlocked file unallocated block verify error
  12 : Process record lock limit exceeded
  13 : Invalid File ID
  14 : No room in BDOS internal lock table

*  -  returned only for files opened in unlocked mode
** -  returned only by the Lock Record function
      for files opened in unlocked mode
```

The following BDOS functions return a Directory Code in register A:

15. Open File
16. Close File
17. Search For First
18. Search For Next
19. Delete File
22. Make File
23. Rename File
30. Set File Attributes
100. Set Directory Label
101. Read File XFCB
102. Write File XFCB

The Directory Code definitions for register A are shown in Table 2-9

Table 2-9. BDOS Directory Codes

```
00 - 03  : successful function
     255 : unsuccessful function
```

With the exception of the BDOS search functions, Directory Code values (0-3) have no significance other than to indicate a successful result. However, for the search functions, a successful Directory Code identifies the relative starting position of the directory element in the calling process' current DMA buffer.

If the Set BDOS Error Mode function is used to place the BDOS in return error mode, the following functions return an Error Flag on physical errors:

14. Select Disk
35. Compute File Size
38. Access Drive
46. Get Disk Free Space
48. Flush Buffers
101. Return Directory Label Data

The Error Flag definition for register A is shown in Table 2-9.

Table 2-10. BDOS Error Flags

00	: successful function
255	: physical error : refer to register H

The BDOS returns register H values for all three of the above categories in the following format:

Nl	N2

where Nl denotes the high order nibble and N2 denotes the low order nibble. The following rules govern the assignment of values to Nl and N2.

Nl For functions that return Error Codes, the BDOS sets Nl to the number of sectors successfully read or written before the error is encountered. This information is returned only when a process uses the Set Multi-Sector Count function to set the BDOS logical sector count to a value other than one; otherwise the BDOS sets Nl to zero. Successful read and write functions also set Nl to zero.

Functions that return a Directory Code or an Error Flag set NI to zero.

N2 The values contained in N2 identify BDOS physical and extended errors. The BDOS returns values in N2 only if it is in one of the return error modes; otherwise, it sets N2 to zero. Table 2-10 lists the physical and extended error codes returned in N2.

Table 2-11. BDOS Physical and Extended Errors

```
00 - no error or not a register H error
01 - Bad Sector : permanent error
02 - R/O : read/only disk
03 - R/O File : read/only file
   - File Opened in Read/only Mode
04 - Select : drive select error
05 - File Currently Open
06 - Close Checksum Error
07 - Password Error
08 - File Already Exists
09 - Illegal ? in FCB
10 - Open File Limit Exceeded
11 - No Room in System Lock List
```

Note: Register H is equal to zero if the called function is successful. In addition, the BDOS sets N2 to zero when register A returns a value other than 255. Except for functions that return Directory Codes, if register A contains a value of 255 upon return, N2 identifies the error when the BDOS is in return error mode.

The following two functions represent a special case because they return an address in registers H and L.

```
27.         Get Addr(Alloc)
31.         Get Addr(Disk Parms)
```

When the BDOS is in return error mode and it detects a physical error for these functions, it returns to the calling process with registers A, H, and L all set to 255. otherwise, they return no error code.

Under MP/M II, the following functions also represent a special case.

```
13.         Reset Disk System
28.         Write Protect Disk
37.         Reset Drive
```

These functions return to the calling process with registers A, H, and L all set to 255 if another process has an open file or has made a BDOS Access Drive call that prevents the reset or write protect operation (see Section 2.2.13). If the BDOS is not in return error mode, these functions also display an error message identifying the process that prevented the requested operation.

2.3 Base Page Initialization

The region of memory located from BASE+OOOOH to BASE+OOFFH is called the base page of a memory segment (BASE = memory segment base address) . The base page contains several segments of code and data that are used by transient programs while running under MP/M II. The code and data areas are shown below for reference. All addresses are relative to the beginning of the memory segment.

Table 2-12. Base Page Areas

Locations from to	Contents
OOOOH - 0002H	Contains a jump instruction to the XIOS process termination entry point at XIOS BASE + 3. This entry point may also be used for direct XIOS calls to the XIOS console status, console input, console output, and list output primitive functions.
0003H - 0004H	(Reserved)
0005H - 0007H	Contains a jump instruction to the BDOS and XDOS, and serves two purposes: JMP 0005H provides the primary entry point to the BDOS and XDOS, and LHLD 0006H places the address field of the jump instruction in the HL register pair. This value (-l) is the highest address of the memory segment available to the transient program. Note: The RDT program changes the address field to reflect the reduced memory size in debug mode.
0008H - 003AH	Reserved interrupt locations for Restarts 1 - 7
003BH - 004FH	(not currently used - reserved)
0050H	Identifies the drive from which the transient program is read. A value of zero designates the default drive, a value of one to sixteen identifies drives A through P.
0051H - 0052H	Contains the address of the password- field of the first command-tail operand in the default DMA buffer beginning at 0080H. The CLI sets this field to zero if no password for the first command-tail operand is specified.

Table 2-12. (continued)

Location from to	Contents
0053H	Contains the length of the password field for the first command-tail operand. The CLI also sets this field to zero if no password for the first command-tail is specified.
0054H 0055H	Contains the address of the password field of the second command-tail operand in the default DMA buffer beginning at 0080H. The CLI sets this field to zero if no password for the second command-tail operand is specified.
0056H	Contains the length of the password field for the second command-tail operand. The CLI also sets this field to zero if no password for the second command-tail is specified.
0057H - 005BH	(not currently used - reserved)
005CH - 007BH	Default File Control Block (FCB) area 1 initialized by the CLI for a transient program from the first command-tail operand of the command line (if it exists).
006CH - 007BH	Default File Control Block (FCB) area 2 initialized by the CLI for a transient program from the second command-tail operand of the command line (if it exists). Note: this area overlays the last 16 bytes of default FCB area 1. To use the information in this area, the transient program must copy it to another location before using FCB area 1.
007CH - 007CH	Current record position of default FCB area 1. This field is used with default FCB area 1 in sequential record processing.
007DH - 007FH	optional default random record position. This field is an extension of default FCB area 1 used in random record processing.
0080H - OOFFH	Default 128-byte disk buffer (also filled with the command tail when the CLI loads a transient program).

The CLI initializes the base page prior to initiating a transient program. The fields at BASE+0050H and above are initialized from the command line invoking the transient program. The command line format of a transient program usually takes the form:

```
                     <command> <command tail>
```

where

```
<command>          => {d:}filename{;password}

<command tail>     => (no command tail)
                   => <file spec>
                   => <file spec><delimiter><file spec>

<file spec>        => {d:}filename{.type}{;password}
```

If a drive {d:} is specified in the <command> field, the CLI initializes the command drive field at 0050H to the drive index (A = 1, ... , P = 16). Otherwise, it sets the field to zero.

The default FCB at 005CH is defined if a command tail is entered. otherwise, the fields at 5CH, 68H to 6BH are set to binary zeros, the fields from 5DH to 67H are set to blanks. The fields at 51H through 53H are set if a password is specified for the first <file spec> of the command tail. If not, these fields are set to zero.

The default FCB at 006Ch is defined if a second <file spec> exists in the command tail. otherwise, the fields at 6CH, 78H to 7Bh are set to binary zeros, the fields from 5DH to 67H are set to blanks. The fields at 54H through 56H are set if a password is specified for the second <file spec> of the command tail. If not, these fields are set to zero.

Transient programs often use the default FCB at 005CH for file operations. This FCB may even be used for random file access because the three bytes starting at 007DH are available for this purpose. However, a transient program must copy the contents of the default FCB at 006CH to another area before using the default FCB at 005CH, because an open operation for the default FCB at 005CH overwrites the FCB data at 006CH.

The default DMA address for transient programs is BASE+0080H. The CLI also initializes this area to contain the command tail of the command line. The first position contains the number of characters in the command line, followed by the command line characters. The command line characters are preceded by a leading blank and are translated to ASCII upper-case. Because the 128-byte region beginning at BASE+0080H is the default DMA, the BDOS file system moves 128-byte records to this area with read operations and accesses 128-byte records from this area with write operations. The transient program must extract the command tail information from this buffer before performing file operations unless it explicitly changes the DMA address with the BDOS Set DMA Address function. The base page fields of 0051H through 0056H locate the password fields of the first two file specifications in the command tail if they exist. These fields are provided so that transient programs are not required to parse the command tail for password fields. However, the transient program must save the password, or change the DMA address before performing file operations.

The following example illustrates the initialization of the command line fields of the base page. Assuming the following command line is typed at the console:

```
A:PROGRAM B:FILE.TYP;PASS C:FILE.TYP;PASSWORD
```

A hexadecimal dump of BASE+0050H to BASE+OOA5H would show the base page initialization performed by the CLI.

```
0050H: 01 8D 00 04 9D 00 08 00 00 00 00 00 02 46 49 4C . . . . . . .FIL
0060H: 45 20 20 20 20 54 59 50 00 00 00 00 03 46 49 4C E . .TYP. .FIL
0070H: 45 20 20 20 20 54 59 50 00 00 00 00 00 00 00 00 E . .TYP. .
0080H: 24 20 42 3A 46 49 4C 45 2E 54 59 50 3B 50 41 53 . B:FILE.TYP;PAS
0090H: 53 20 43 3A 46 49 4C 45 2E 54 59 50 3B 50 41 53 S C:FILE.TYP;PAS
OOAOH: 53 57 4F 52 44 00                                SWORD.
```

2.4 BDOS Function Calls

FUNCTION 0: SYSTEM RESET
Entry Parameters: Register C: OOH

The System Reset function terminates the calling process, releasing all system resources owned by the process. In general, a process can own one or more of the following resources: memory segments, consoles, printers, mutual exclusion messages, and system lock list entries that record open files and locked records. All released resources become available to other processes on the system. For example, if a system console is released by a terminating process, it is usually given back to the console's TMP. This occurs when the TMP is the highest priority process waiting for the console.

Normally, the System Reset function operates the same way under MP/M II as it does under CP/M: the calling program terminates and the user receives the command prompt. Note that the disk subsystem is not reset by System Reset under MP/M II.

For transient programs, System Reset is equivalent to a jump to BASE+O.

FUNCTION 1: CONSOLE INPUT
Entry Parameters: Register C: 01H Returned Value: Register A: ASCII Character

The Console Input function reads the next character from the console device to register A. Most graphic characters, including carriage return, line feed, and backspace (CONTROL-H) are echoed to the console. Tab characters (CONTROL-I) are expanded in columns of 8 characters. However, the terminate process (CONTROL-C) and detach process (CONTROL-D) characters are intercepted by the BDOS (see Section 2.1). The BDOS does not return control to the calling process until a character is typed, thus suspending execution if a character is not ready.

MP/M II performs an XDOS Attach Console call (function 146) for the calling process if it does not own the console (see Section 2.1).

FUNCTION 2: CONSOLE OUTPUT
Entry Parameters: Register   C: 02H Register   E: ASCII Character

The Console Output function sends the ASCII character from register E to the console device. It expands tab characters (CONTROL-I) in columns of 8 characters, and checks for start scroll (CONTROL-S) , stop scroll (CONTROL-Q) , terminate process (CONTROL-C) , and detach process (CONTROL-D) (see Section 2.1).

MP/M II performs an XDOS Attach Console call (function 146) for the calling process if it does not own the console (see Section 2.1).

FUNCTION 3: RAW CONSOLE INPUT
Entry Parameters: Register   C: 03H Returned   Value: Register   A: ASCII Character

The Raw Console Input function reads the next console character to register A. It reads all characters including control characters, without any testing or interpretation.

MP/M II performs an XDOS Attach Console call (Function 146) for the calling process if it does not own the console (see Section 2.1).

MP/M II does not support the CP/M Reader Input function because the system treats all character I/O devices such as the reader/punch as consoles.

FUNCTION 4: RAW CONSOLE OUTPUT
Entry Parameters: Register C: 04H Register E: ASCII Character

The Raw Console Output function sends the ASCII character from register E to the console device. It does not test the output character; that is, tabs are not expanded and no checks are made for control characters.

MP/M II performs an XDOS Attach Console call (function 146) for the calling process if it does not own the console (see Section 2.1).

MP/M II does not support the CP/M Punch Output function.

FUNCTION 5: LIST OUTPUT
Entry Parameters: Register C: 05H Register E: ASCII Character

The List Output function sends the ASCII character in register E to the list device.

MP/M II performs an XDOS Attach List call (function 158) for the calling process if it does not own the list device (see Section 2.1).

FUNCTION 6: DIRECT CONSOLE I/O			
Entry Parameters:			
Register	C:	06H	
Register	E:	OFFH	(input/ status) or
		OFEH	(status)or
		OFDH	(input)
		char	(output)
Returned	Value:		
Register	A:	char or status (no value)	

MP/M II supports direct console I/O for those specialized applications where unadorned console input and output is required. The programmer should use direct console I/O carefully because it bypasses all the normal control character functions. Programs that perform direct I/O through the BIOS under previous releases of CP/M should be changed to use direct I/O under the new BDOS so that they can be fully supported under future releases of MP/M and CP/M.

A Process calls Function 6 by passing one of four different values in register E. These are summarized in Table 2-13, below.

Table 2-13. Function 6 Entry Parameters

Register E value	Meaning
OFFH	console input/status command, returns an input character; if no character is ready, a value of zero is returned.
OFEH	console status command (On return, register A contains 00 if no character is ready; otherwise it contains FFH.)
OFDH	console input command, returns an input character; this function will suspend the calling process until a character is ready.
ASCII Character	Function 6 assumes register E contains a valid ASCII character and sends it to the console.

Note: MP/M II is not compatible with MP/M 1.1 in regard to Function 6 with a parameter of E=FFH. Under MP/M 1.1 the direct console input command (E=FFH) suspends the calling process until a character is typed, whereas MP/M II returns immediately with a zero if no character is available. To upgrade programs using Function 6 with E=FFH under MP/M 1.1 to MP/M II, the direct input command (E=FDH) must be used. The change from MP/M 1.1 was required to achieve consistent direct console I/O handling between CP/M, MP/M II, CP/M-86 and MP/M-86.

MP/M II performs an XDOS Attach Console call (Function 146) for the calling process if it does not own the console (see Section 2.1) . MP/M II performs a dispatch if a direct console input/status command (E=FFH) is made which returns a zero indicating that a character is not ready.

FUNCTION 7: GET I/O BYTE

MP/M II does not support the Get I/O Byte function.

FUNCTION 8: SET I/O BYTE

MP/M II does not support the Set I/O Byte function.

FUNCTION 9: PRINT STRING
Entry Parameters: Register C: 09H Registers DE: String Address

The Print String function sends the character string stored in memory at the location addressed by register pair DE to the console until it encounters a "$" in the string. Function 9 expands tab characters (CONTROL-I) in columns of 8 characters. It also checks for start scroll (CONTROL-S) , stop scroll (CONTROL-Q) , terminate process (CONTROL-C) and detach process (CONTROL-D) (see Section 2.1).

MP/M II performs an XDOS Attach Console call (Function 146) for the calling process if it does not own the console (see Section 2.1).

```
FUNCTION 10: READ CONSOLE BUFFER

Entry Parameters:
Register    C:    OAH
Registers DE: Buffer Address

Returned Value:
Console Characters in Buffer
```

The Read Console Buffer function reads a line of edited console input to a buffer addressed by register pair DE. It terminates input when the buffer is filled or when it encounters a return (CONTROL-M) or a line feed (CONTROL-J) character. The input buffer addressed by DE has the following format:

```
   DE: +0 +1 +2 +3 +4 +5 +6 +7 +8 . . . .        +n
       +---------------------------------------------
       :mx:nc:cl:c2:c3:c4:c5:c6:c7: . . .        :??:
       +---------------------------------------------
```

where "mx" is the maximum number of characters which the buffer holds, and "nc" is the number of characters placed in the buffer. The characters entered by the operator follow the "nc" value. The value "mx" must be set, prior to making a Function 10 call and may range in value from 1 to 255. Setting "mx" to zero is equivalent to setting "mx" to one. The value "nc" is returned to the calling process and may range from zero to "mx". If nc < mx, then uninitialized positions follow the last character, denoted by "??" in the above figure. Note that a terminating return or line feed character is not placed in the buffer and not included in the count "nc".

Function 10 recognizes the edit control characters summarized in Table 2-14, below.

Table 2-14. Console Buffer Edit Control Characters

Character	Edit Control Function
rub/del	removes and echoes the last character
CONTROL-C	reboots when at the beginning of line
CONTROL-E	causes physical end of line
CONTROL-H	backspaces one character position
CONTROL-J	(line feed) terminates input line
CONTROL-M	(return) terminates input line
CONTROL-P	echoes console output to the list device
CONTROL-R	retypes the current line after new line
CONTROL-U	removes current line after new line
CONTROL-X	backspaces to beginning of current line

The control functions that return the cursor to the leftmost position (e.g., CONTROL-X) do so only to the column position where the prompt ended (in earlier releases, the cursor returned to the extreme left margin). This convention simplifies data input and line correction.

MP/M II performs an XDOS Attach Console call (Function 146) for the calling process if it does not own the console (see Section 2.1).

FUNCTION 11: GET CONSOLE STATUS
Entry Parameters: Register   C: OBH Returned Value: Register   A: Console Status

The Get Console Status function checks to see if a character has been typed at the console. If a character is ready, Function 11 returns the value 01H in register A. If a character is not ready, it returns a value of OOH.

MP/M II performs an XDOS Attach Console call (Function 146) for the calling process if it does not own the console (see Section 2.1).

FUNCTION 12: RETURN VERSION NUMBER
Entry Parameters: Register   C: OCH Returned Value: Registers HL: Version Number

The Return Version Number function provides information that allows version independent programming. It returns a two-byte value in register pair HL: H contains 01H for MP/M and L contains 30H, the BDOS file system version number. Function 12 is useful for writing applications programs that provide both random and sequential file access, and disabling the random access when operating under early versions of CP/M.

XDOS Function 163 can be called to obtain the MP/M version number.

FUNCTION 13: RESET DISK SYSTEM		
Entry Parameters:		
Register	C:	ODH
Returned Value:		
Register	A:	Return Code

The Reset Disk System function restores the file system to a reset state where all the disk drives are set to read/write (see Functions 28 and 29), the default disk is set to drive A, and the default DMA address is reset to BASE+0080H. This function can be used, for example, by an application program that requires disk changes during operation. Function 37 (Reset Drive) can also be used for this purpose.

This function is conditional under MP/M II. If another process has an open file on a removeable or read/only drive, the disk reset is denied and no drives are reset.

Upon return, if the reset operation is successful, register A is set to zero. Otherwise, register A is set to OFFH (255 decimal) . If the BDOS error mode is not Return Error mode (see Function 45), then an error message is displayed at the console, identifying a process owning an open file.

FUNCTION 14: SELECT DISK		
Entry Parameters:		
Register	C:	OEH
Register	E:	Selected Disk
Returned Value:		
Register	A:	Error Flag
Register	H:	Physical Error

The Select Disk function designates the disk drive named in register E as the default disk for subsequent BDOS file operations. Register E is set to 0 for drive A, 1 for drive B, and so-forth through 15 for drive P in a full 16 drive system. In addition, function 14 logs in the designated drive if it is currently in the reset state. Logging-in a drive activates the drive's directory until the next disk system reset or drive reset operation.

FCBs that specify drive code zero (dr = OOH) automatically reference the currently selected default drive. FCBs with drive code values between 1 and 16, however, ignore the selected default drive and directly reference drives A through P.

Upon return, register A contains a zero if the select operation was successful. If a physical error was encountered, the select function performs different actions depending on the BDOS error mode (see Function 45) . If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the calling process is terminated. Otherwise, the select function returns to the calling process with register A set to OFFH and register H set to one of the following physical error codes:

01 : Permanent error
04 : Select error

```
FUNCTION 15: OPEN FILE

Entry Parameters:
Register    C: OFH
Registers DE: FCB Address

Returned    Value:
Register    A: Directory Code
Register    H: Physical or
               Extended Error
```

The Open File function activates the FCB for a file that exists in the disk directory under the currently active user number or user zero. The calling process passes the address of the FCB in register pair DE, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and type, and byte 12 specifying the extent. Normally, byte 12 of the FCB is initialized to zero. Interface attributes f5' and W of the FCB specify the mode in which the file is to be opened as shown below:

```
f5' = 0,      f6' = 0 - Open in locked mode (default)
f5' = 1,      f6’ = 0 - open in unlocked mode
f5' = 0 or 1, f6’ = 1 - Open in read/only mode
```

If the file is password protected in Read mode, the correct password must be placed in the first eight bytes of the current DMA or have been previously established as the default password (see Function 106). Note that the current record field of the FCB ("cr") must be zeroed by the calling process if the file is to be accessed sequentially from the first record.

The Open File function performs the following steps for files opened in locked or read/only mode. If the current user is non zero, and the file to be opened does not exist under the current user number, the open function searches user zero for the file. If the file exists under user zero, and has the system attribute (t2') set, the file is opened under user zero. The open mode is automatically set to read/only when this is done.

The Open File function also performs the following action for files opened in locked mode when the current user number is zero. If the file exists in the directory under user zero, and has both the system attribute (t2') set and the read/only attribute (tl') set, the open mode is automatically set to read/only. Note that read/only mode implies the file can be concurrently accessed by other processes if they open the file in read/only mode.

If the open operation is successful, the user's FCB is activated for read and write operations as follows. The relevant directory information is copied from the matching directory FCB into bytes dO through dn of the FCB. A checksum is computed and assigned to the FCB. BDOS functions that require an open FCB (e.g. Read Sequential) verify that the FCB checksum is valid before performing their operation. if the file is opened in unlocked mode, bytes rO and rl of the FCB are set to a two byte value called the File ID. The File ID is a required parameter for the BDOS Lock Record and Unlock Record functions. If the open mode is forced to read/only mode, interface attribute f8' is set to one in the user's FCB. In addition, if the referenced file is password protected in Write mode and the correct password was not passed in the DMA or did not match the default password, interface attribute f7' is set to one. Write operations are not supported for an activated FCB if interface attribute f7' or f8' is true.

The BDOS file system also creates an open file item in the system lock list to record a successful open file operation. While this item exists, no other process can delete, rename, or modify the file's attributes. In addition, this item prevents other processes from opening the file if the file was opened in locked mode. It also requires that other processes match the file's open mode if the file was opened in unlocked or read/only mode. Normally, this item remains in the system lock list until the file is permanently closed or the process that opened the file terminates.

When the open operation is successful, the open function also makes an Access date and time stamp for the opened file when the following conditions are satisfied: the reference drive has a directory label that requests Access date and time stamping, the opened file has an XFCB, and the referenced drive is read/write.

Upon return, the open function returns a directory code in register A with the value 0 through 3 if the open was successful, or FFH (255 decimal) if the file was not found. Register H is set to zero in both of these cases. If a physical or extended error was encountered, the open function performs different actions depending on the BDOS error mode (see Function 45) . If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the process is terminated. otherwise, the open function returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

```
01 :  Permanent error
04 :  Select error
05 :  File is open by another process or by the
      current process in an incompatible mode
07 :  File password error
09 :  ? in the FCB file name or type field
10 :  Process open file limit exceeded
11 :  No room in the system lock list
```

```
FUNCTION 16: CLOSE FILE

Entry Parameters:
Register    C: 10H
Registers  DE: FCB Address

Returned Value:
Register     A: Directory Code
Register     H: Physical or
               Extended Error
```

The Close File function performs the inverse of the open file function. The calling process passes the address of an FCB in the register pair DE. The referenced FCB must have been previously activated by a successful open or make function call (see functions 15 and 22). Interface attribute f5l specifies how the file is to be closed as shown below:

f5l = 0 - Permanent close (default mode)
f5l = 1 - Partial close

The close function first verifies that the referenced FCB has a valid checksum. If the checksum is valid and the referenced FCB contains new information because of write operations to the FCB, the close function permanently records the new information in the referenced disk directory. Note that the FCB does not contain new information and the directory update step is bypassed if only read and/or update operations have been made to the referenced FCB. However, the close function always attempts to locate the FCB's corresponding entry in the directory, and returns an error if the directory entry is not found.

If the close function successfully performs the above steps, and if interface attribute f5’ indicates that the close is permanent, the close function removes the file's item from the system lock list. If the FCB was opened in unlocked mode, it also purges all record lock items belonging to the file from the system lock list. Because the file's lock list item is removed, the close function invalidates the FCB's checksum to ensure the referenced FCB is not subsequently used with BDOS functions that require an open FCB (e.g. Write Sequential).

The close function also makes an Update date and time stamp for the closed file when the following conditions are satisfied: the reference drive has a directory label that requests Update date and time stamping, the referenced file has an XFCB, the referenced drive is read/write, and a write operation to the file was made since the FCB was opened. None of these steps are performed for partial close operations (f5’ = 1).

Upon return, the close function returns a Directory Code in register A with the value 0 to 3 if the close was successful, or FFH (255 Decimal) if the file was not found. Register H is set to zero in both of these cases. If a physical or extended error was encountered, the close function performs different actions depending on the BDOS error mode (see Function 45) . If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the calling process is terminated. Otherwise, the close function returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

01 : Permanent error
02 : Read/only disk
04 : Select error
06 : FCB checksum error

FUNCTION 17: SEARCH FOR FIRST
Entry Parameters: Register C: 11H Registers DE: FCB Address Returned Value: Register A: Directory Code Register H: Physical Error

Search For First scans the directory for a match with the FCB addressed by register pair DE. Two types of searches can be performed. For standard searches, the calling process initializes bytes 0 through 12 of the referenced FCB, with byte 0 specifying the drive directory to be searched, bytes 1 through 11 specifying the file or files to be searched for, and byte 12 specifying the extent. Normally byte 12 is set to zero. An ASCII question mark (63 decimal, 3F hex) in any of the bytes 1 through 12 matches all entries on the directory in the corresponding position. This facility, called ambiguous reference, can be used to search for multiple files on the directory. When called in the standard mode, the search function scans for the first file entry in the specified directory that matches the FCB and belongs to the current user number.

The search function also initializes the Search For Next function. After the search function has located the first directory entry matching the referenced FCB, the Search For Next function can be called repeatedly to locate all remaining matching entries. In terms of execution sequence, however, the Search For Next call must either follow a Search For First or Search For Next call with no other intervening BDOS disk-related function calls.

If byte 0 of the referenced FCB is set to a question mark, the search function ignores the remainder of the referenced FCB and locates the first directory entry residing on the current default drive. All remaining directory entries can be located by making multiple Search For Next calls. This type of search operation is not normally made by application programs, but it does provide complete flexibility to scan all current directory values. Note that this type of search operation must be performed to access a drive's Directory Label (see Section 2.2.5).

Upon return, the search function returns a Directory Code in register A with the value 0 to 3 if the search was successful, or OFFH (255 Decimal) if a matching directory entry was not found. Register H is set to zero in both of these cases. For successful searches, the current DMA is also filled with the directory record containing the matching entry, and the relative starting position is A * 32 (i.e. rotate the A register left 5 bits, or ADD A five times). Although not normally required for application programs, the directory information can be extracted from the buffer at this position.

If a physical error was encountered, the search function performs different actions depending on the BDOS error mode (See function 45). If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the calling process is terminated. Otherwise, the search function returns to the calling process with register A set to OFFH and register H set to one of the following physical error codes:

01	Permanent error
04	Select error

FUNCTION 18: SEARCH FOR NEXT

Entry Parameters:
Register C: 12H

Returned Value:
Register A: Directory Code
Register H: Physical Error

The Search For Next function is identical to the Search For First function, except that the directory scan continues from the last entry that was matched. Function 18 returns a Directory code in register A, analogous to Function 17. Note: In execution sequence, a Function 18 call must follow either a Function 17 or another Function 18 call with no other intervening BDOS disk-related function calls.

```
FUNCTION 19: DELETE FILE

Entry Parameters:
Register    C: 13H
Registers  DE: FCB Address

Returned Value:
Register     A:   Directory Code
Register     H:   Extended or
            Physical Error
```

The Delete File function removes files and/or XFCBs that match the FCB addressed in register pair DE. The filename and type may contain ambiguous references (i.e., question marks in bytes fl through t3) , but the "dr" byte cannot be ambiguous, as it can in the Search and Search Next functions. Interface attribute f5' specifies the type of delete operation to be performed as shown below:

f5' = 0 - Standard Delete (default mode)
f5' = 1 - Delete only XFCB's

If any of the files specified by the referenced FCB are password protected, the correct password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see Function 106).

For standard delete operations, the delete function removes all directory entries belonging to files that match the referenced FCB. All disk directory and data space owned by the deleted files is returned to free space, and becomes available for allocation to other files. Directory XFCBs that were owned by the deleted files are also removed from the directory. If interface attribute f5' of the FCB is set to 1, Function 19 deletes only the directory XFCBs matching the referenced FCB. Note: If any of the files matching the input FCB specification fail the password check, are read/only, or are currently open by another process, then the delete function deletes no files or XFCBs. This applies to both types of delete operations.

A process can delete a file that it currently has open if the file was opened in locked mode. However, a checksum error is returned if the process makes a subsequent reference to the file with a BDOS function requiring an open FCB. Files open in read/only or unlocked mode cannot be deleted by any process.

Upon return, the delete function returns a Directory Code in register A with the value 0 to 3 if the delete was successful, or OFFH (255 Decimal) if no file matching the referenced FCB was found. Register H is set to zero in both of these cases. If a physical or extended error was encountered, the delete function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the calling process is terminated. Otherwise, the delete function returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

```
01 :  Permanent error
02 :  Read/only disk
03 :  Read/only file
04 :  Select Error
05 :  File open by another process or open
      in read/only or unlocked mode
07 :  File password error
```

```
FUNCTION 20: READ SEQUENTIAL

Entry Parameters:
Register    C: 14H
Registers DE: FCB Address

Returned Value:
Register    A: Error Code
Register    H: Physical Error
```

The Read Sequential function reads the next one to sixteen 128-byte records from a file into memory beginning at the current DMA address. The BDOS Multi-Sector Count (see Function 44) determines the number of records to be read. The default is one record. The FCB addressed by register pair DE must have been previously activated by an Open or Make function call.

Function 20 reads each record from byte "cr" of the extent, then automatically increments the "cr" field to the next record position . If the "cr" field overflows then the function automatically opens the next logical extent and resets the "cr" field to 0 in preparation for the next read operation. The calling process must set the "cr" field to 0 following the open call if the intent is to read sequentially from the beginning of the file.

Upon return, the Read Sequential function sets register A to zero if the read operation was successful. otherwise, register A contains an error code identifying the error as shown below:

01 : Reading unwritten data (end of file)
09 : Invalid FCB
10 : FCB checksum error
11 : Unlocked file verification error
255 : Physical error; refer to register H

Error Code 01 is returned if no data exists at the next record position of the file. Normally, the no data situation is encountered at the end of a file. However, it can also occur if an attempt is made to read a data block which has not been previously written, or an extent which has not been created. These situations are usually restricted to files created or appended with the BDOS random write functions (see Functions 34 and 40).

Error Code 09 is returned if the FCB was invalidated by a previous BDOS random read or write call that returned an error. A Read Random call (Function 33) for an existing record in the file, can be made to revalidate the FCB.

Error Code 10 is returned if the referenced FCB failed the FCB checksum test.

Error Code 11 is returned if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. This error is only returned for files open in unlocked mode.

Error Code 255 is returned if a physical error was encountered and the BDOS error mode is Return Error mode or Return and Display Error mode (see Function 45) . If the error mode is the default mode, a message identifying the physical error is displayed at the console and the calling process is terminated. When a physical error is returned to the calling process, it is identified by the four low-order bits of register H as shown below:

01 : Permanent error
04 : Select error

The Read Sequential function also sets the four high-order bits of register H on all error returns when the BDOS Multi-Sector Count is greater than one. In this case, the four bits contain an integer set to the number of records successfully read before the error was encountered. This value can range from 0 to 15. The high-order four bits of register H are always zeroed when the Multi Sector Count is equal to one.

FUNCTION 21: WRITE SEQUENTIAL
Entry Parameters: Register C: 15H Registers DE: FCB Address Returned Value: Register A: Error Code Register H: Physical Error

The Write Sequential function writes one to sixteen 128-byte data records beginning at the current DMA address into the file named by the FCB addressed in register pair DE. The BDOS Multi Sector Count (see Function 44) determines the number of 128 byte records that are written. The default is one record. The referenced FCB must have been previously activated by a BDOS Open or Make function call.

Function 21 places the record into the file at the position indicated by the "cr" byte of the FCB, and then automatically increments the "cr" byte to the next record position. If the "cr" field overflows, the function automatically opens or creates the next logical extent and resets the "cr" field to 0 in preparation for the next write operation. If Function 21 is used to write to an existing file, then the newly written records overlay those already existing in the file. The calling process must set the "cr" field to 0 following an Open or Make call if the intent is to write sequentially from the beginning of the file.

Upon return, the Write Sequential function sets register A to zero if the write operation was successful. otherwise, register A contains an error code identifying the error as shown below:

01	No available directory space
02	No available data block
08	Record locked by another process
09	Invalid FCB
10	FCB checksum error
11	Unlocked file verification error
255	Physical error : refer to register H

Error Code 01 is returned when the write function attempts to create a new extent that requires a new directory entry and no available directory entries exist on the selected disk drive.

Error Code 02 is returned when the write command attempts to allocate a new data block to the file and no unallocated data blocks exist on the selected disk drive.

Error Code 08 is returned if the write function attempts to write to a record locked by another process. This error is only returned for files open in unlocked mode.

Error Code 09 is returned if the FCB was invalidated by a previous BDOS random read or write call that returned an error. A Read Random call (Function 33) for an existing record in the file can be made to revalidate the FCB.

Error Code 10 is returned if the referenced FCB failed the FCB checksum test.

Error Code 11 is returned if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. This error is only returned for files open in unlocked mode.

Error Code 255 is returned if a physical error was encountered and the BDOS error mode is Return Error mode or Return and Display Error mode (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console and the calling process is terminated. When a physical error is returned to the calling process, it is identified by the four low-order bits of register H as shown below:

```
01 :  Permanent error
02 :  Read/only disk
03 :  Read/only file or
      File open in read/only mode or
      File password protected in Write mode
04 :  Select error
```

The Write Sequential function also sets the four high-order bits of register H on all error returns when the BDOS Multi-Sector Count is greater than one. In this case, the four bits contain an integer set to the number of records successfully written before the error was encountered. This value can range from zero to 15. The high order four bits of register H are always zeroed when the Multi Sector Count is equal to one.

```
FUNCTION 22: MAKE FILE

Entry Parameters:
Register    C: 16H
Registers DE: FCB Address

Returned    Value:
Register    A:  Directory Code
Register    H:  Physical or
                Extended Error
```

The Make File function creates a new directory entry for a file under the current user number. It also creates an XFCB for the file if the referenced drive has a Directory Label that invokes automatic creation of XFCBs. The calling process passes the address of the FCB in the register pair DE, with byte 0 of the FCB specifying the drive, bytes 1 through 11 specifying the filename and type, and byte 12 set to the extent number. Normally, byte 12 is set to zero. Byte 32 of the FCB (the "cr" field) must be initialized to zero (before or after the Make call) if the intent is to write sequentially from the beginning of the file.

Interface attribute f5' specifies the mode in which the file is to be opened. Interface attribute f6l specifies whether a password is to be assigned to the created file. The interface attributes are summarized below:

f5’ = 0 – Open in locked mode (default mode)
f5’ = 1 – Open in unlocked mode
f6’ = 0 – Don't assign password (default)
f6’ = 1 – Assign password to created file

When attribute W is set to 1, the calling process must place the password in the first 8 bytes of the current DMA buffer and set byte 9 of the DMA buffer to the password mode (See Function 102).

The Make function returns with an error if the referenced FCB names a file that currently exists in the directory under the current user number. A preceding delete operation can be made if there is any possibility of duplication.

If the make operation is successful, it activates the referenced FCB for file operations (opens the FCB) and initializes both the directory entry and the referenced FCB to an empty file. A checksum is computed and assigned to the FCB. BDOS functions that require an open FCB (e.g. Write Random) verify that the FCB checksum is valid before performing their operation. If the open mode is unlocked, bytes rO and rl are set to a two byte value called the File ID. The File ID is a required parameter for the BDOS Lock Record and Unlock Record functions. Note that the Make File function initializes all file attributes to zero.

The BDOS file system also creates an open file item in the system lock list to record a successful make file operation. While this item exists, no other process can delete, rename, or modify the file's attributes.

If the referenced drive contains a Directory Label that invokes automatic creation of XFCBs, the make function creates an XFCB and makes a Creation date and time stamp for the created file. Note that the Creation time stamp is not made (the XFCB Creation time stamp field is set to zeroes) if an XFCB is assigned to a file by the BDOS Write File XFCB call. If interface attribute f6’ of the FCB is 1, the make function also assigns the password passed in the DMA to the file.

Upon return, the make function returns a directory code in register A with the value 0 through 3 if the make operation was successful, or OFFH (255 decimal) if no directory space was available. Register H is set to zero in both of these cases. If a physical or extended error was encountered, the make function performs different actions depending on the BDOS error mode (see Function 45) . If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the calling process is terminated. Otherwise, the make function returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

```
01 : Permanent error
02 : Read/only disk
04 : Select error
08 : File already exists
09 : ? in filename or type field
10 : Process open file limit exceeded
11 : No room in the system lock list
```

```
FUNCTION 23: RENAME FILE

Entry Parameters:
Register    C: 17H
Registers DE: FCB Address

Returned Value:
Register    A:  Return Code
Register    H:  Physical or
                Extended Error
```

The Rename function uses the FCB addressed by register pair DE to change all directory entries of the file specified by the filename in the first 16 bytes of the FCB to the filename in the second 16 bytes. If the file specified by the first filename is password protected, the correct password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see Function 106). The calling process must also ensure that the filenames specified in the FCB are valid and unambiguous, and that the new filename does not already exist on the drive. Function 23 uses the "dr" code at byte 0 of the FCB to select the drive. The drive code at byte 16 of the FCB is ignored.

A process can rename a file that it has open if the file was opened in locked mode. However, if the process subsequently references the file with a BDOS function requiring an open FCB, a checksum error is returned. A file open in read/only or unlocked mode cannot be renamed by any process.

Upon return, the rename function returns a Directory Code in register A with the value 0 to 3 if the rename was successful, or OFFH (255 Decimal) if the file named by the first filename in the FCB was not found. Register H is set to zero in both of these cases. If a physical or extended error was encountered, the rename function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the process is terminated. Otherwise, the rename function returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

```
01 : Permanent error
02 : Read/only disk
03 : Read/only file
04 : Select error
05 : File open by another process
07 : File password error
08 : File already exists
09 : ? in file name or type field
```

FUNCTION 24: RETURN LOGIN VECTOR
Entry Parameters: Register C: 18H Returned Value: Registers HL: Login Vector

Function 24 returns the login vector in register pair HL. The login vector is a 16-bit value with the least significant bit of L corresponding to drive A, and the high-order bit of H corresponding to the 16th drive, labeled P. A "0" bit indicates that the drive is not on-line, while a "1" bit indicates the drive is active. A drive is made active by either an explicit BDOS Select Disk call (number 14), or an implicit selection when a BDOS file operation specifies a non-zero "dr" byte in the FCB. Function 24 maintains compatibilty with earlier releases, since registers A and L contain the same values upon return.

FUNCTION 25: RETURN CURRENT DISK
Entry Parameters: Register C: 19H Returned Value: Register A: Current Disk

Function 25 returns the currently selected default disk number in register A. The disk numbers range from 0 through 15 corresponding to drives A through P.

FUNCTION 26: SET DMA ADDRESS
Entry Parameters: Register C: IAH Registers DE: DMA Address

"DMA" is an acronym for Direct Memory Address, which is often used in connection with disk controllers that directly access the memory of the computer to transfer data to and from the disk subsystem. Under MP/M II, the current DMA is usually defined as the buffer in memory where a record resides before a disk write and after a disk read operation. If the BDOS Multi-Sector Count is equal to one (see Function 44) , the size of the buffer is 128 bytes. However, if the BDOS Multi-Sector Count is greater than one, the size of the buffer must equal N * 128, where N equals the Multi Sector Count.

Some BDOS functions also use the current DMA to pass parameters and to return values. For example, BDOS functions that check and assign file passwords, require that the password be placed in the current DMA. As another example, Function 46 (Get Disk Free Space) returns its results in the first 3 bytes of the current DMA. When the current DMA is used in this context, the size of the buffer in memory is determined by the specific requirements of the called function.

When a transient program is initiated by the CLI, its DMA address is set to BASE+0080H. The BDOS Reset Disk System function (Function 13) also sets the DMA address to BASE+0080H. The Set DMA function can change this default value to another memory address. The DMA address is set to the value passed in the register pair DE. The DMA address remains at this value until it is changed by another Set DMA Address, or Reset Disk System call.

FUNCTION 27: GET ADDR(ALLOC)
Entry Parameters: Register C: 1BH Returned Value: Registers HL: ALLOC Address

MP/M II maintains an "allocation vector" in main memory for each active disk drive. Many programs commonly use the information provided by the allocation vector to determine the amount of free data space on a drive. Note, however, that the allocation information may be inaccurate if the drive has been marked read/only.

Function 27 returns in register pair HL, the base address of the allocation vector for the currently selected drive. If a physical error is encountered when the BDOS error mode is one of the return modes (see Function 45) , Function 27 returns the value OFFFFH in the register pair HL.

In banked switched MP/M II systems, the allocation vector may be placed in bank zero. This is an XIOS option. In this case, a transient program that has been loaded into another bank cannot access the allocation vector. However, the BDOS function, Get Disk Free Space (Function 46), can be used to directly return the number of free 128 byte records on a drive. In fact, the MP/M II utilities that display a drive's free space (STAT,SDIR, and SHOW) use Function 46 for that purpose.

FUNCTION 28: WRITE PROTECT DISK
Entry Parameters: Register C: 1CH Returned value: Register A: Return code

The Write Protect Disk function provides temporary write protection for the currently selected disk by marking the drive as read/only. No process can write to a disk that is in the read/only state. A successful drive reset operation must be performed for a read/only drive to restore it to the read/write state (see Functions 13 and 37).

The Write Protect Disk function is conditional under MP/M II. If another process has an open file on the drive, this function is denied and the value OFFH is returned to the calling process. otherwise, register A is set to zero. Note that a drive in the read/only state cannot be reset by a process if another process has an open file on the drive.

FUNCTION 29: GET READ/ONLY VECTOR
Entry Parameters: Register C: 1DH Returned Value: Registers HL: R/O Vector Value

Function 29 returns a bit vector in register pair HL that indicates which drives have the temporary read/only bit set. The read/only bit is set either by a BDOS Write Protect Disk call, or by the automatic software mechanisms within MP/M II that detect changed disk media.

The format of the bit vector is analogous to that of the login vector returned by Function 24. The least significant bit corresponds to drive A, while the most significant bit corresponds to drive P.

FUNCTION 30: SET FILE ATTRIBUTES
Entry Parameters: Register C: 1EH Registers DE: FCB Address Returned Value: Register A: Directory Code

The Set File Attributes function is the only BDOS function that allows a program to manipulate file attributes. Other BDOS functions can interrogate these file attributes but cannot change them. The file attributes that can be set or reset by Function 30 are: fl' through f4’, R/O (tl'), System (t2’), and Archive (t3'). The register pair DE addresses an FCB containing a filename with the appropriate attributes set or reset. The calling process must ensure that it does not specify an ambiguous filename. In addition, if the specified file is password protected, the correct password must be placed in the first eight bytes of the current DMA buffer, or have been previously established as the default password (see Function 106).

Function 30 searches the FCB specified directory for an entry belonging to the current user number that matches the FCB specified name and type fields. The function then updates the directory to contain the selected indicators. File attributes tl', t2', and t3' are defined by MP/M II. They are described in Section 2.2.4. Attributes fl' through f4' are not presently used, but may be useful for application programs, because they are not involved in the matching process used by the BDOS during Open File and Close File operations. Indicators f5' through f8' are reserved for use as interface attributes.

This function is not performed if the file specified by the referenced FCB is currently open for another process. It is performed, however, if the referenced file is open for the calling process in locked mode. After successfully setting the attributes of a file opened by the calling process, any subsequent file reference requiring an open FCB returns a checksum error. This function does not set the attributes of a file currently open in read/only or unlocked mode for any process.

Upon return, Function 30 returns a Directory Code in register A with the value 0 to 3 if the function was successful, or OFFH (255 Decimal) if the file specified by the referenced FCB was not found. Register H is set to zero in both of these cases. If a physical or extended error was encountered, the Set File Attributes function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the process is terminated. Otherwise, Function 30 returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

```
01 : Permanent error
02 : Read/only disk
04 : Select error
05 : File open by another process
07 : File password error
09 : ? in file name or type field
```

```
FUNCTION 31: GET ADDR(DISK PARMS

Entry Parameters:
Register    C:    1FH

Returned Value:
Registers HL: DPB Address
```

Function 31 returns in register pair HL, the address of the XIOS-resident Disk Parameter Block (DPB) for the currently selected drive. (Refer to the MP/M II System Guide for the format of the DPB) . The calling process can use this address to extract the disk parameter values for display or to compute the space on a drive.

If a physical error is encountered when the BDOS error mode is one of the return modes (see Function 45), Function 31 returns the value OFFFFH in the register pair HL.

```
FUNCTION 32: SET/GET USER CODE

Entry Parameters:
Register    C:    20H
Register    E:    OFFH (get) or
User Code (set)

Returned Value:
Register    A:    Current Code or
                  (no value)
```

A process can change or interrogate the currently active user number by calling Function 32. If register E = OFFH, then the value of the current user number is returned in register A where the value is in the range of 0 to 15. If register E is not OFFH, then the current user number is changed to the value of E (modulo 16).

```
FUNCTION 33: READ RANDOM

Entry Parameters:
Register    C: 21H
Registers DE: FCB Address

Returned    Value:
Register    A: Error Code
Register    H: Physical Error
```

The Read Random function is similar to the Read Sequential function except that the read operation takes place at a particular random record number, selected by the 24-bit value constructed from the three byte (rO, rl, r2) field beginning at position 33 of the FCB. Note that the sequence of 24 bits is stored with the least significant byte first (rO) , the middle byte next (rl) , and the high byte last (r2). The random record number can range from 0 to 242,143. This corresponds to a maximum value of 3 in byte r2.

To read a file with Function 33, the calling process must first open the base extent (extent 0). This ensures that the FCB is properly initialized for subsequent random access operations. (The base extent may or may not contain any allocated data) . Function 33 places the specified record number in the random record field, and then BDOS reads the record into the current DMA address. The function automatically sets the logical extent and current record values, but unlike the Read Sequential function, it does not advance the record number. Thus a subsequent Read Random call re-reads the same record. After a random read operation, a file can be accessed sequentially, starting from the current randomly accessed position. However, the last randomly accessed record is re-read or re-written when switching from random to sequential mode.

If the BDOS Multi-Sector count is greater than one (see Function 44), the Read Random function reads multiple consecutive records into memory beginning at the current DMA. The rO,rl, and r2 field of the FCB is automatically incremented to read each record. However, the FCBs random record number is restored to the first record's value upon return to the calling process. Upon return, the Read Random function sets register A to zero if the read operation was successful. otherwise, register A contains one of the following error codes:

```
 01 :  Reading unwritten data
 03 :  Cannot Close current extent
 04 :  Seek to unwritten extent
 06 :  Random record number out of range
 10 :  FCB checksum error
 11 :  Unlocked file verification error
255 :  Physical error : refer to register H
```

Error Code 01 is returned when the Read Random function accesses a data block that has not been previously written.

Error Code 03 is returned when the Read Random function cannot close the current extent prior to moving to a new extent.

Error Code 04 is returned when a read random operation accesses an extent that has not been created.

Error Code 06 is returned when byte 35 (r2) of the referenced FCB is greater than 3.

Error Code 10 is returned if the referenced FCB failed the FCB checksum test.

Error Code 11 is returned if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. This error is only returned for files open in unlocked mode.

Error Code 255 is returned if a physical error was encountered and the BDOS error mode is one of the return modes (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console and the calling process is terminated. When a physical error is returned to the calling process, it is identified by the four low-order bits of register H as shown below:

```
01 : Permanent Error
04 : Select Error
```

The Read Random function also sets the four high-order bits of register H on all error returns when the BDOS Multi-Sector Count is greater than one. In this case, the four bits contain an integer set to the number of records successfully read before the error was encountered. This value can range from 0 to 15. The high-order four bits of register H are always zeroed when the Multi-Sector Count is equal to one.

```
FUNCTION 34: WRITE RANDOM

Entry Parameters:
Register    C: 22H
Registers DE: FCB Address

Returned    Value:
Register    A: Error Code
Register    H: Physical error
```

The Write Random function is analogous to the Read Random Function, except that data is written to the disk from the current DMA address. If the disk extent and/or data block where the data is to be written is not already allocated, the BDOS automatically performs the allocation before the write operation continues.

To write to a file using the Write Random function, the calling process must first open the base extent (extent 0). This ensures that the FCB is properly initialized for subsequent random access operations. The base extent may or may not contain any allocated data, but opening extent 0 records the file in the directory so that it is can be displayed by the DIR utility. If a process does not open extent 0 and allocates data to some other extent, the file will be invisible to the DIR utility.

The Write Random function sets the logical extent and current record positions to correspond with the random record being written, but does not change the random record number. Thus sequential read or write operations can follow a random write, with the current record being re-read or re-written as the calling process switches from random to sequential mode.

If the BDOS Multi-Sector count is greater than one (see Function 44), the Write Random function reads multiple consecutive records into memory beginning at the current DMA. The rO,rl, and r2 field of the FCB is automatically incremented to write each record. However, the FCB's random record number is restored to the first record's value upon return to the calling process. Upon return, the Write Random function sets register A to zero if the write operation was successful.

Otherwise, register A contains one of the following error codes:

```
     02 : No available data block
     03 : Cannot Close current extent
     05 : No available directory space
     06 : Random record number out of range
     08 : Record locked by another process
     10 : FCB checksum error
     11 : Unlocked file verification error
    255 : Physical error : refer to register H
```

Error Code 02 is returned when the write command attempts to allocate a new data block to the file and no unallocated data blocks exist on the selected disk drive.

Error Code 03 is returned when the Read Random function cannot close the current extent prior to moving to a new extent.

Error Code 05 is returned when the write function attempts to create a new extent that requires a new directory entry and no available directory entries exist on the selected disk drive.

Error Code 06 is returned when byte 35 (r2) of the referenced FCB is greater than 3.

Error Code 08 is returned when the Write Random function attempts to write to a record locked by another process. This error is only returned for files open in unlocked mode.

Error Code 10 is returned if the referenced FCB failed the FCB checksum test.

Error Code 11 is returned if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. This error is only returned for files open in unlocked mode.

Error Code 255 is returned if a physical error was encountered and the BDOS error mode is one of the return modes (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console and the calling process is terminated. When a physical error is returned to the calling process, it is identified by the four low-order bits of register H as shown below:

```
     01 :  Permanent error
     02 :  Read/only disk
     03 :  Read/only file
           File open in read/only mode
           File password protected in Write mode
     04 :  Select Error
```

The Write Random function also sets the four high-order bits of register H on all error returns when the BDOS Multi-Sector Count is greater than one. In this case, the four bits contain an integer set to the number of records successfully read before the error was encountered. This value can range from 0 to 15. The high-order four bits of register H are always zeroed when the Multi-Sector Count is equal to one.

```
FUNCTION 35: COMPUTE FILE SIZE

Entry Parameters:
Register    C: 23H
Registers DE: FCB Address

Returned    Value:
Register    A: Error Flag
Register    H: Physical or
               Extended error

Random Record Field Set
```

The Compute File Size function determines the "virtual" file size, which is, in effect, the address of the record immediately following the end of the file. The "virtual" size of a file corresponds to the physical size if the file is written sequentially. If the file is written in random mode, gaps might exist in the allocation, and the file might contain fewer records than the indicated size. For example, if a single record with record number 262,143 (the MP/M II maximum) is written to a file using the Write Random function, then the "virtual" size of the file is 262,144 records even though only 1 data block is actually allocated.

To compute file size, the calling process passes in register pair DE, the address of a FCB in random mode format (bytes rO, rl and r2 present) . Note that the FCB must contain an unambiguous filename and type. Function 35 sets the random record field of the FCB to the random record number + 1 of the last record in the file. If the r2 byte is set to 04, then the file contains the maximum record count 262,144.

A process can append data to the end of an existing file by calling Function 35 to set the random record position to the end of file, then performing a sequence of random writes starting at the preset record address.

Note: the BDOS does not require that the file be open to use Function 35.

Upon return, Function 35 returns a zero in register A if the file specified by the referenced FCB was found, or a OFFH in register A if the file was not found. Register H is set to zero in both of these cases. If a physical or extended error was encountered, Function 35 performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the process is terminated. Otherwise, Function 35 returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended errors:

```
01 : Permanent error
04 : Select error
09 : ? in file name or type field
```

FUNCTION 36: SET RANDOM RECORD
Entry Parameters: Register C: 24H Registers DE: FCB Address Returned Value: Random Record Field Set

The Set Random Record function returns the random record number of the next record to be accessed from a file that has been read or written sequentially to a particular point. This value is returned in the random record field (bytes rO, rl, and r2) of the FCB addressed by the register pair DE. Function 36 can be useful in two ways.

First, it is often necessary to initially read and scan a sequential file to extract the positions of various "key" fields. As each key is encountered, Function 36 is called to compute the random record position for the data corresponding to this key. If the data unit size is 128 bytes, the resulting record number minus one is placed into a table with the key for later retrieval. After scanning the entire file and tabularizing the keys and their record numbers, you can move directly to a particular record by performing a random read using the corresponding random record number that was saved earlier. The scheme is easily generalized when variable record lengths are involved since the program need only store the buffer-relative byte position along with the key and record number to find the exact starting position of the keyed data at a later time.

A second use of Function 36 occurs when switching from a sequential read or write over to random read or write. A file is sequentially accessed to a particular point in the file, Function 36 is called which sets the record number, and subsequent random read and write operations continue from the next record in the file.

```
FUNCTION 37: RESET DRIVE

Entry Parameters:
Register    C:    25H
Register    DE:   Drive Vector

Returned Value
Register    A:    Return Code
```

The Reset Drive function is used to programmatically restore specified drives to the reset state (a reset drive is not logged-in and is in read/write status) . The passed parameter in register pair DE is a 16 bit vector of drives to be reset, where the least significant bit corresponds to the first drive A, and the high-order bit corresponds to the sixteenth drive, labeled P. Bit values of "l" indicate that the specified drive is to be reset.

This function is conditional under MP/M II. If another process has a file open on a drive to be reset, and the drive is removeable or read/only, the Drive Reset function is denied and no drives are reset.

Upon return, if the reset operation is successful, register A is set to zero. Otherwise, register A is set to OFFH (255 decimal) . If the BDOS error mode is not Return Error mode (see Function 45), then an error message is displayed at the console, identifying a process owning an open file.

```
FUNCTION 38: ACCESS DRIVE

Entry Parameters:
Register    C: 26H
Register    DE: Drive Vector

Returned Value:
Register    A: Return Code
Register    H: Extended Error
```

The Access Drive function inserts a special open file item into the system lock list for each specified drive. While the item exists in the lock list, the drive cannot be reset by another process. As in Function 37, the calling process passes the drive vector in register pair DE. The format of the drive vector is the same as that used in Function 37.

The Access Drive function inserts no items if insufficient free space exists in the lock list to support all the new items or if the number of items to be inserted puts the calling process over the lock list open file maximum. This maximum is a MP/M II Gensys option. If the BDOS error mode is the default mode (see Function 45), a message identifying the error is displayed at the console and the calling process is terminated. Otherwise, the Access Drive function returns to the calling process, register A is set to OFFH and register H is set to one of the following values.

10 : Process Open File limit exceeded
11 : No room in the system lock list

Register A is set to zero if the Access Drive function is successful.

FUNCTION 39: FREE DRIVE
Entry Parameters: Register C: 27H Register DE: Drive Vector

The Free Drive function purges the open lock list of all file and locked record items that belong to the calling process on the specified drives. As in Function 38, the calling process passes the drive vector in register pair DE.

Function 39 does not close files associated with purged open file lock list items. In addition, if a process references a "purged" file with a BDOS function requiring an open FCB, a checksum error is returned. A file that has been written to should be closed before making a Free Drive call to the file's drive. otherwise data may be lost.

```
FUNCTION 40: WRITE RANDOM WITH
      ZERO FILL

Entry Parameters:
Register    C:    28H
Register    DE:   FCB address

Returned Value:
Register    A:    Error Code
Register    H:    Physical Error
```

The Write Random With Zero Fill function is similar to the Write Random function (Function 34) with the exception that a previously unallocated data block is filled with zeroes before the record is written. If this function has been used to create a file, records accessed by a read random operation that contain all zeroes identify unwritten random record numbers. Unwritten random records in allocated data blocks of files created using the Write Random function contain uninitialized data.

```
FUNCTION 41: TEST AND WRITE RECORD

Entry Parameters:
Register    C:    29H
DE: FCB Address

Returned Value:
Register    A:    Error Code
Register    H:    Physical Error
```

The Test and Write Record provides a means of verifying the current contents of a record on disk before updating it. The calling process must set bytes rO, rl, and r2 of the FCB addressed by register pair DE to the random record number of the record to be tested. The original version of the record (i.e. the record to be tested) must reside at the current DMA address, followed immediately by the new version of the record. The record size can range from 128 bytes to sixteen times that value depending on the BDOS Multi Sector Count (see Function 44).

Function 41 verifies that the first record is identical to the record on disk before replacing it with the new version of the record. If the record on disk does not match, the record on disk is not changed and an error code is returned to the calling process.

The Test and Write function is intended for use in situations where more than one process has read/write access to a common file. This situation is supported under MP/M II, when more than one process opens the same file in unlocked mode. Function 41 is a logical replacement for the record lock/unlock sequence of operations because it prevents two processes from simultaneously updating the same record. Note that this function is also supported for files open in locked mode to provide compatibility between MP/M II and CP/M.

Upon return, the Test and Write Random function sets register A to zero if the function was successful. otherwise, register A contains one of the following error codes:

```
 01 : Reading unwritten data
 03 : Cannot Close current extent
 04 : Seek to unwritten extent
 06 : Random record number out of range
 07 : Records did not match
 08 : Record locked by another process
 10 : FCB checksum error
 11 : Unlocked file verification error
255 : Physical error : refer to register H
```

Error Code 01 is returned when the Test and Write function accesses a data block that has not been previously written.

Error Code 03 is returned when the Test and Write function cannot close the current extent prior to moving to a new extent.

Error Code 04 is returned when a read operation accesses an extent that has not been created.

Error Code 06 is returned when byte 35 (r2) of the referenced FCB is greater than 3.

Error Code 07 is returned when the Test and Write record test fails.

Error Code 08 is returned if the specified record is locked by another process. This error is only returned for files open in unlocked mode.

Error Code 10 is returned if the referenced FCB failed the FCB checksum test.

Error Code 11 is returned if the BDOS cannot locate the FCB's directory entry when attempting to verify that the referenced FCB contains current information. This error is only returned for files open in unlocked mode.

Error Code 255 is returned if a physical error was encountered and the BDOS error mode is one of the return modes (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console and the calling process is terminated. When a physical error is returned to the calling process, it is identified by the four low-order bits of register H as shown below:

```
01 :  Permanent error
02 :  Read/only disk
03 :  Read/only file or
      File open in read/only mode
      File password protected in Write mode
04 :  Select Error
```

The Test and Write function also sets the four high-order bits of register H on all error returns when the BDOS Multi-Sector Count is greater than one. In this case, the four bits contain an integer set to the number of records successfully tested or written before the error was encountered. This value can range from 0 to 15. The high-order four bits of register H are always zeroed when the Multi Sector Count is equal to one.

```
FUNCTION 42: LOCK RECORD

Entry Parameters:
Register          C:    2AH
DE: FCB Address

Returned Value:
Register    A:    Error Code
Register    H:    Physical Error
```

The Lock Record function locks one or more consecutive records so that no other program with access to the records can simultaneously lock or update them. This function is only supported for files open in unlocked mode. If it is called for a file open in locked or read/only mode, no locking action is performed and a successful result is returned. This is done to provide compatibility between MP/M II and CP/M.

The calling process passes in register pair DE, the address of an FCB in which the Random Record Field is filled with the random record number of the first record to be locked. The number of records to be locked is determined by the BDOS Multi-Sector Count (see Function 44) . The current DMA must contain the 2-byte File ID returned by the Open File function when the referenced FCB was opened. Note that the File ID is only returned by the Open function when the open mode is unlocked.

The Lock Record function requires that each record number to be locked reside in an allocated block for the file. In addition, Function 42 verifies that none of the records to be locked are currently locked by another process. Both of these tests are made before any records are locked.

A MP/M II system generation parameter specifies the maximum number of records that may be locked by a single process. Each locked record consumes an entry in the BDOS system lock table which is shared by locked record and open file entries. Another MP/M II system generation parameter sets the size of this table. If there is not sufficient space in the system lock table to lock all the specified records, or the process record lock limit is exceeded, then the Lock Record function locks no records and returns an error code to the calling process.

Upon return, the Lock Record function sets register A to zero if the lock operation was successful. Otherwise, register A contains one of the following error codes:

```
 01 : Reading unwritten data
 03 : Cannot Close current extent
 04 : Seek to unwritten extent
 06 : Random record number out of range
 08 : Record locked by another process
 10 : FCB checksum error
 11 : Unlocked file verification error
 12 : Process record lock limit exceeded
 13 : Invalid File ID
 14 : No room in the system lock list
255 : Physical error : refer to register H
```

Error Code 01 is returned when the Lock Record function accesses a data block that has not been previously written.

Error Code 03 is returned when the Lock Record function cannot close the current extent prior to moving to a new extent.

Error Code 04 is returned when the Lock Record function accesses an extent that has not been created.

Error Code 06 is returned when byte 35 (r2) of the referenced FCB is greater than 3.

Error Code 08 is returned if the specified record is locked by another process.

Error Code 10 is returned if the referenced FCB failed the FCB checksum test.

Error Code 11 is returned if the BDOS cannot locate the referenced FCB's directory entry when attempting to verify that the FCB contains current information.

Error Code 12 is returned when the sum of the number of records currently locked by the calling process and the number of records to be locked by the Lock Record call, exceeds the maximum allowed value. This value is an MP/M II Gensys parameter.

Error Code 13 is returned when an invalid File ID is placed in the current DMA.

Error Code 255 is returned if a physical error was encountered and the BDOS error mode is one of the return modes (see Function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console and the calling process is terminated. When a physical error is returned to the calling process, it is identified by the four low-order bits of register H as shown below:

01 : Permanent error
04 : Select Error

The Lock Record function also sets the four high-order bits of register H on all error returns when the BDOS Multi-Sector Count is greater than one. In this case, the four bits contain an integer set to the number of records successfully locked before the error was encountered. This value can range from 0 to 15. The high-order four bits of register H are always zeroed when the Multi-Sector Count is equal to one.

FUNCTION 43: UNLOCK RECORD
Entry Parameters: Register C: 2BH DE: FCB Address Returned Value: Register A: Error Code Register H: Physical Error

The Unlock Record function unlocks one or more consecutive records previously locked by the Lock Record function. This function is only supported for files open in unlocked mode. If it is called for a file open in locked or read/only mode, no locking action is performed and a successful result is returned. This is done to provide compatibility between MP/M II and CP/M.

The calling process passes in register pair DE, the address of an FCB in which the Random Record Field is filled with the random record number of the first record to be unlocked. The number of records to be unlocked is determined by the BDOS Multi-Sector Count (see Function 44) . The current DMA must contain the 2-byte File ID returned by the Open File function when the referenced FCB was opened. Note that the File ID is only returned by the open function when the open mode is unlocked.

The Unlock Record function will not unlock a record that is currently locked by another process. However, no error is returned if a process attempts to do that. Thus, if the Multi-Sector Count is greater than one, the Unlock Record function unlocks all records locked by the calling process, while skipping those records locked by other processes.

Upon return, the Unlock Record function sets register A to zero if the unlock operation was successful. otherwise, register A contains one of the following error codes:

```
 01 : Reading unwritten data
 03 : Cannot Close current extent
 04 : Seek to unwritten extent
 06 : Random record number out of range
 10 : FCB checksum error
 11 : Unlocked file verification error
 13 : Invalid File ID
255 : Physical error : refer to register H
```

Error Code 01 is returned when the Unlock Record function accesses a data block that has not been previously written.

Error Code 03 is returned when the Unlock Record function cannot close the current extent prior to moving to a new extent.

Error Code 04 is returned when the Unlock Record function accesses an extent that has not been created.

Error Code 06 is returned when byte 35 (r2) of the referenced FCB is greater than 3.

Error Code 10 is returned if the referenced FCB failed the FCB checksum test.

Error Code 11 is returned if the BDOS cannot locate the referenced FCB's directory entry when attempting to verify that the FCB contains current information.

Error Code 13 is returned when an invalid File ID is placed in the current DMA.

Error Code 255 is returned if a physical error was encountered and the BDOS error mode is one of the return modes (See function 45). If the error mode is the default mode, a message identifying the physical error is displayed at the console and the calling process is terminated. When a physical error is returned to the calling process, it is identified by the four low-order bits of register H as shown below:

```
01 : Permanent error
04 : Select Error
```

The Unlock Record function also sets the four high-order bits of register H on all error returns when the BDOS Multi-Sector Count is greater than one. In this case, the four bits contain an integer set to the number of records successfully locked before the error was encountered. This value can range from 0 to 15. The high-order four bits of register H are always zeroed when the Multi-Sector Count is equal to one.

FUNCTION 44: SET MULTI-SECTOR CNT		
Entry Parameters:		
Register	C: 2CH	
	E: Number of Sectors	
Returned Value:		
Register	A:	Return Code

The Set Multi-Sector Count function provides logical record blocking under MP/M II. It enables a process to read and write from 1 to 16 "physical" records of 128 bytes at a time during subsequent BDOS Read and Write functions. It also specifies the number of 128 byte records to be locked or unlocked by the BDOS Lock and Unlock functions.

Function 44 sets the Multi-Sector Count value for the calling process to the value passed in register E. Once set, the specified Multi-Sector Count remains in effect until the calling process makes another Set Multi-Sector Count function call and changes the value. Note that the Command Line Interpreter (CLI) sets the Multi-Sector Count to one when it initiates a transient program.

The Multi-Sector count affects BDOS error reporting for the BDOS read, write, lock and unlock functions. If an error interrupts these functions when the Multi-Sector is greater than one, they return the number of records successfully processed in the high order four bits of register H.

Upon return, register A is set to zero if the specified value is in the range of 1 to 16. Otherwise, register A is set to OFFH

FUNCTION 45: SET BDOS ERROR MODE	
Entry Parameters:	
Register	C: 2DH
	E: BDOS error mode
Returned Value:	
None	

The SET BDOS Error Mode function determines how physical and extended errors (see Section 2.2.13) are handled for a process. The Error Mode can exist in three modes: the default mode, Return Error mode and Return and Display Error mode. In the default mode, BDOS displays a system message at the console identifying the error and terminates the calling process. In the return modes, BDOS sets register A to OFFH (255 Decimal), places an error code identifying the physical or extended error in the four low-order bits of register H, and returns to the calling process. In Return and Display mode, BDOS displays the system message before returning to the calling process. No system messages are displayed, however, when BDOS is in Return Error mode.

Function 45 sets the BDOS error mode for the calling process to the mode specified in register E. If register E is set to OFFH (255 Decimal), the error mode is set to Return Error mode. If register E is set to OFEH (254 Decimal), the error mode is set to Return and Display mode. If register E is set to any other value, the error mode is set to the default mode.

```
FUNCTION 46: GET DISK FREE SPACE

Entry Parameters:
Register    C:    2EH
            E:    Drive

Returned Value:   First 3 bytes
                  of current DMA
                  buffer
Register          A:    Error Flag
Register          H:    Physical error
```

The Get Disk Free Space function determines the number of free sectors (128 byte records) on the specified drive. The calling process passes the drive number in register E, with 0 for drive A, 1 for B, etc. , through 15 for drive P in a full 16 drive system. Function 46 returns a binary number in the first 3 bytes of the current DMA buffer. This number is returned in the following format:

fsO	fsl	fs2

Disk Free Space Field Format

fso = low byte
fsl = middle byte
fs2 = high byte

Upon return, register A is set to zero if the BDOS Error Mode is the default mode. However, if the BDOS Error Mode is one of the return modes (see Function 45) and a physical error was encountered, register A is set to OFFH (255 Decimal), and register H is set to one of the following values:

01 - Permanent error
04 - Select error

```
FUNCTION 47: CHAIN TO PROGRAM

Entry Parameters:
Register    C:    2FH
```

The Chain To Program function provides a means of chaining from one program to the next without operator intervention. Although there is no passed parameter for this call, the calling process must place a command line terminated by a null byte in the default DMA buffer.

Function 47 does not return any values to the calling process because any errors encountered are handled by the Command Line Interpreter (CLI).

Note: Function 47 makes an XDOS Conditional Attach Console call for the calling process. If the calling process is detached from its console, the program chain is not performed and Function 47 returns to the calling process.

```
FUNCTION 48: FLUSH BUFFERS

Entry Parameters:
Register    C:    30H

Returned Value:
Register    A:    Error Flag
Register    H:    Permanent Error
```

The Flush Buffers function forces the write of any write pending records contained in internal blocking/deblocking buffers. This function only affects those systems that have implemented a write-deferring blocking/deblocking algorithm in their XIOS (see Section 2.2.12).

Upon return, register A is set to zero if the flush operation was successful. If a physical error was encountered, the Flush Buffers function performs different actions depending on the BDOS error mode (see Function 45). If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the calling process is terminated. otherwise, the Flush Buffers function returns to the calling process with register A set to OFFH and register H set to the following physical error code:

01 : Permanent error

```
FUNCTION 100: SET DIRECTORY LABEL

      Entry Parameters:
Register    C:    64H
Register    DE:   FCB Address

Returned Value:
      Register A : Directory Code
      Register H : Physical or
                   Extended Error
```

The Set Directory Label function creates a directory label or updates the existing directory label for the specified drive. The calling process passes in register pair DE, the address of an FCB containing the name, type, and extent fields to be assigned to the directory label. The name and type fields of the referenced FCB are not used to locate the directory label in the directory; they are simply copied into the updated or created directory label. The extent field of the FCB (byte 12) contains the user's specification of the directory label data byte. The definition of the directory abel data byte is:

```
bit   7 - Require passwords for password-protected files
      6 - Perform access date and time stamping
      5 - Perform update date and time stamping
      4 - Make function creates XFCBs
      0 - Assign a new password to the directory label
```

If the current directory label is password protected, the correct password must be placed in the first 8 bytes of the current DMA or have been previously established as the default password (see function 106). If bit 0 (the low-order bit) of byte 12 of the FCB is set to 1, it indicates that a password for the directory label has been placed in the second eight bytes of the current DMA.

Function 100 returns a Directory Code in register A with a value from 0 to 3 if the directory label create or update was successful, or OFFH (255 Decimal) if no space existed in the referenced directory to create a directory label. Register H is set to zero in both of these cases. If a physical error or extended was encountered, function 100 performs different actions depending on the BDOS error mode (see function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the calling process is terminated. otherwise, function 100 returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

```
01 : Permanent error
02 : Read/only disk
04 : Select Error
07 : File password error
```

```
FUNCTION 101:     RETURN DIRECTORY
      LABEL DATA

Entry Parameters:
Register    C:    65H
Register    E:    Drive

Returned Value:
Registers A :  Directory label
               Data Byte
Register  H :  Physical Error
```

The Return Directory Label Data function returns the data byte of the directory label for the specified drive. The calling process passes the drive number in register E with 0 for drive A, 1 for drive B, and so on through 15 for drive P in a full sixteen drive system. The format of the directory label data byte is shown below:

```
bit   7 - Require passwords for password protected files
      6 - Perform access date and time stamping
      5 - Perform update data and time stamping
      4 - Make function creates XFCBs
      0 - Directory label exists on drive
```

Function 101 returns the directory label data byte to the calling process in register A. Register A equal to zero indicates that no directory label exists on the specified drive. If a physical error is encountered by function 101 when the BDOS Error mode is in one of the return modes (see function 45) , this function returns with register A set to OFFH (255 Decimal) and register H set to one of the following:

```
01 : Permanent error
04 : Select error
```

```
FUNCTION 102: READ FILE XFCB

Entry Parameters:
Register    C: 66H
Register DE: FCB Address

Returned Value:
Register    A : Directory Code
Register    H : Physical Error
```

The Read File XFCB function reads the directory XFCB information for the specified file into bytes 20 through 32 of the specified FCB. The calling process passes in register pair DE, the address of an FCB in which the drive, filename, and type fields have been defined.

If function 102 is successful, it sets the following fields in the referenced FCB:

```
byte 12 :   XFCB password mode field
            bit 7 - Read mode
            bit 6 - Write mode
            bit 5 - Delete mode
```

Byte 12 equal to zero indicates the file has not been assigned a password.

byte 13 - 23 : XFCB password field (encrypted)

byte 24 - 27 : XFCB Create or Access time stamp field

byte 28 - 31 : XFCB Update time stamp field

Upon return, function 102 returns a Directory Code in register A with the value 0 to 3 if the XFCB read operation was successful, or OFFH (255 Decimal) if the XFCB was not found. Register H is set to zero in both of these cases. If a physical error or extended was encountered, function 102 performs different actions depending on the BDOS error mode (see function 45) . If the BDOS error mode is in the default mode, a message identifying the error is displayed at the console and the calling process is terminated. Otherwise, function 102 returns to the calling process with register A set to OFFH and register H set to one of the following physical error codes:

01 : Permanent error
04 : Select Error

```
FUNCTION 103: WRITE FILE XFCB

Entry Parameters:
Register    C: 67H
Register DE: FCB Address

Returned Value:
Register A :  Directory Code
Register H :  Physical or
              Extended Error
```

The Write File XFCB function creates a new XFCB or updates the existing XFCB for the specified file. The calling process passes in register pair DE, the address of an FCB in which the drive, name, type, and extent fields have been defined. The "ex" field, if set, specifies the password mode and whether a new password is to be assigned to the file. The for-mat of the extent byte is shown below:

```
FCB   byte 12     (ex) : XFCB password mode
      bit 7 -     Read mode
      bit 6 -     Write mode
      bit 5 -     Delete mode
      bit 0 -     assign new password to the file
```

If bit 0 is set to 1, the new password must reside in the second 8 bytes of the current DMA. If the FCB is currently password protected, the correct password must reside in the first 8 bytes of the current DMA, or have been previously established as the default password (see function 106).

Upon return, function 100 returns a Directory Code in register A with the value 0 to 3 if the XFCB create or update was successful, or OFFH (255 Decimal) if no directory label existed on the specified drive, or the file named in the FCB was not found, or no space existed in the directory to create an XFCB. Register H is set to zero in all of these cases. If a physical error or extended was encountered, function 103 performs different actions depending on the BDOS error mode (see function 45). If the BDOS error mode is the default mode, a message identifying the error is displayed at the console and the calling process is terminated. otherwise, function 103 returns to the calling process with register A set to OFFH and register H set to one of the following physical or extended error codes:

```
01 : Permanent error
02 : Read/only disk
04 : Select Error
07 : File password error
```

FUNCTION 104: SET DATE AND TIME
Entry Parameters: Register C: 68H Register DE: TOD Address Returned Value: none

The Set Date and Time function sets the system internal date and time. The calling process passes the address of a 4-byte structure containing the date and time specification in the register pair DE. The format of the date and time data structure is:

```
byte 0 - 1        : Date field
byte 2            : Hour field
byte 3            : Minute field
```

The date is represented as a 16-bit integer with day 1 corresponding to January 1, 1978. The time is represented as two bytes: hours and minutes stored as two BCD digits.

Under MP/M II, this function also sets the second field of the system date and time to zero.

FUNCTION 105: GET DATE AND TIME
Entry Parameters: Register C: 6911 Register DE: TOD Address Return Value TOD

The Get Date and Time function obtains the system internal date and time. The calling process passes in register pair DE, the address of a four-byte data structure which receives the date and time values. The format of the data structure is the same as the format described in function 104. This function is equivalent to MP/M II function 155 except that it does not return the seconds field of the internal time.

FUNCTION 106: SET DEFAULT PASSWORD		
Entry Parameters:		
Register	C:	6AH
Register	DE:	Password Address
Returned	Value: none	

The Set Default Password function allows a process to specify a password value before a file protected by the password is accessed. When the file system accesses a password protected file, it checks the current DMA and the default password for the correct value. A password error is not returned if either password is correct. The default password is maintained by the BDOS in an internal table indexed by the calling process's console number. once assigned, it is maintained until another Set Default Password call is made by a process having the same console number.

To make a function 106 call, the calling process sets register pair DE to the address of an eight byte field containing the password.

FUNCTION 107: RETURN SERIAL NUMBER		
Entry Parameters:		
Register	C:	6BH
Register	DE:	Serial number field

Function 107 returns the MP/M II serial number to the six-byte field addressed by register pair DE.

Section 3
XDOS Interface

3.1 Introduction

This section contains information on data structures used in the XDOS module. The XDOS uses these data structures to:

- manage the memory resource
- communicate messages between processes
- synchronize process execution

Also included are descriptions of the XDOS functions, including the entry parameters and returned values, and a discussion of error handling by the XDOS. The reader should be thoroughly familiar with the material covered in Section 1 before proceeding.

3.2 Process Descriptor Data Structure

Each process running under MP/M II is associated with a Process Descriptor that defines all the characteristics of the process. The XDOS uses the Process Descriptor to save and restore the state of a process. The Process Descriptor data structure is shown below in both PL/M and assembly language.

PL/M:

```
DECLARE CNS$HNDLR STRUCTURE (
   PL ADDRESS,
   STATUS BYTE,
   PRIORITY BYTE,
   STKPTR ADDRESS,
   NAME (8) BYTE,
   CONSOLE$LIST BYTE,
   MEMSEG BYTE,
   DPARAM ADDRESS,
   THREAD ADDRESS,
   DISK$SET$DMA ADDRESS,
   DISK$SLCT BYTE,
   DCNT ADDRESS,
   SEARCHL BYTE,
   SEARCHA ADDRESS,
   PD EXTENT,
   REGISTERS (10) ADDRESS,
   EXTENSION ADDRESS,
   INITIAL (0,0,200,.CNS$STK(19),
            'CNS     `,1,OFFH);
```

```
DECLARE CNS$STK (20) ADDRESS INITIAL (
   OC7C7H,OC7C7H,OC7C7H,OC7C7H,OC7C7H,OC7C7H,
   OC7C7H,OC7C7H,OC7C7H,OC7C7H,OC7C7H,OC7C7H,
   OC7C7H,OC7C7H,OC7C7H,OC7C7H,OC7C7H,OC7C7H,
   OC7C7H,STRT$CNS);
```

Assembly Language:

```
CNSHND:
      DW    0     ;PL
      DB    0     ;STATUS
      DB    200   ;PRIORITY
      DW    CNSTK+38 ;STKPTR
      DB    'CNS      ` ; NAME
      DB    0     ;CONSOLE/LIST
      DB    OFFH  ;MEMSEG (FF = resident)
      DS    2     ;DPARAM
      DS    2     ;THREAD
      DS    2     ;DISK SET DMA
      DS    I     ;DISK SLCT
      DS    2     ;DCNT
      DS    1     ;SEARCHL
      DS    2     ;SEARCHA
      DS    2     ;PD EXTENT
                  ;REGISTERS:
      DS    2     ;HL'
      DS    2     ;DE'
      DS    2     ;BC'
      DS    2     ;AF'
      DS    2     ;IY
      DS    2     ;IX
      DS    2     ;HL
      DS    2     ;DE
      DS    2     ;BC
      DS    2     ;AF
      DS    2     ;EXTENSION

CNSTK:
      DW    OC7C7H,OC7C7H,OC7C7H,OC7C7H
      DW    OC7C7H,OC7C7H,OC7C7H,OC7C7H
      DW    OC7C7H,OC7C7H,OC7C7H,OC7C7H
      DW    OC7C7H,OC7C7H,OC7C7H
      DW    CNSPR       ; CNSTK+38 = PROCEDURE ADR
```

The elements of the Process Descriptor data structure shown above are defined in Table 3-1.

Table 3-1. Process Descriptor Elements

Element	Definition
PL	2-byte link field, initially set by user when creating a process to the address of next Process Descriptor, or zero if no more exist.
STATUS	1 byte, process status, set by system. The Dispatcher reads the status byte to determine the operation to be performed on the process. The values of the status byte are shown below: 00 - process is ready to run 01 - process is dequeueing 02 - process is enqueueing 03- process is polling 04 - process is waiting for a flag 05 - process is on delay list 06 - not implemented under MP/M II 07 - terminate process 08 - set process priority 09 - Dispatch 10 - Attach console 11 - Detach console 12 - Set console 13 - Attach list 14 - Detach list
PRIORITY	1 byte, process priority, set by user.
STKPTR	2 bytes, stack pointer, initially set by user.
NAME	8 bytes, ASCII process name, set by user. The high-order bit of each byte of the process name is reserved for use by the system. The function of each of the high order bits , shown as NAME (n)is described below:
NAME(O)'	The high-order bit of NAME(O) on indicates that the process is performing direct console BIOS calls and that MP/M II should ignore all control characters. It also suppresses the normal console status check done when BDOS disk functions are called. The user may set this bit.

Table 3-1. (continued)

Element	Definition
NAME(1)'	The high-order bit of NAME(1) "on" indicates that the process is currently executing code in the serially re-usable BDOS. MP/M II does not allow a process to Terminate while it is in the BDOS. Any Attempt to abort the process will set NAME(6)' "on". This bit is set by the system; it must not be set by the user.
NAME(2)'	The high-order bit of NAME(2) "on" indicates that no stack swap is done for this process upon entering the BDOS. This bit takes precedence over the system boolean indicating whether user system stacks have been allocated. It is required when more than one process shares the same memory segment and makes BDOS function calls. The user may set this bit.
NAME(3)'	The high-order bit of NAME(3) "on" Indicates that live keyboard simulation is to be suppressed. Live keyboard simulation is done by performing console status calls at each BDOS disk I/O function call. This bit is set by the user.
NAME(4)'	The high-order bit of NAME(4) "on" indicates that extended errors resulting from BDOS calls are to be returned to the calling program; normally an error message is displayed on the console and the calling program is terminated. This bit is set by the user.
NAME(5)'	The high-order bit of NAME(5) "on" indicates that extended errors resulting from BDOS calls are to be returned to the calling program and an error message is to be displayed on the console. This bit is set by the user.
NAME(6)'	The high-order bit of NAME(6) "on" indicates that an attempt has been made to abort the process while either NAME (1) ' or NAME(7)' has been on. This bit is set by the system.

Table 3-1. (continued)

Element	Definition
NAME(7)'	The high-order bit of NAME(7) "on" indicates that the process is not to be aborted by any means. An attempt to abort this process results in setting NAME(6)1 "on". This bit is set by the user.
CONSOLE/LIST	1 byte, low-order four bits contain the console device number to be used by process, and the high-order four bits contain the list device number, set by user.
MEMSEG	1 byte, memory segment table index.
DPARAM	2 bytes, reserved for MP/M II.
THREAD	2 bytes, process list thread, set by system.
DISKSETDMA	2 bytes, default DMA address, set by system on BDOS set DMA calls, can be set by user.
DISK$SLCT	1 byte, default disk/user code, set by system on BDOS set user and disk select calls, can be set by user.
DCNT	2 bytes, reserved for MP/M II.
SEARCHL	1 byte, reserved for MP/M II.
SEARCHA	2 bytes, reserved for MP/M II.
PD EXTENT	2 bytes, reserved for MP/M II.
REGISTERS	20 bytes, 8080 / Z80 register save area, can be set by user prior to process creation in order to pass parameters to a created process. The following entries show the register storage allocation. Bytes are stored in the normal 8080/Z80 manner with the low-order register byte preceding the high-order byte.

Table 3-1. (continued)

Element	Definition
	Bytes 0- 1 HL', Alternate Z80 Bytes 2- 3 DE', Alternate Z80 Bytes 4- 5 BC', Alternate Z80 Bytes 6- 7 AF1, Alternate Z80 Bytes 8- 9 IY Bytes 10-11 IX Bytes 12-13 HL Bytes 14-15 DE Bytes 16-17 BC Bytes 18-19 AF
EXTENSION	2 bytes, reserved for MP/M II

The following conventions should be used in naming processes that are to run under MP/M II: processes that wait on queues that receive command tails from the TMPs should have the same name as the queue that they read. If a process is to be protected from being aborted by a user with the ABORT command, its name must have at least one lower-case character.

3.3 Queue Data Structures

A queue is a first-in first-out (FIFO) mechanism that is implemented in MP/M II to provide several essential functions in the multi-programming environment. Queues can be used for the communication of messages between processes, to synchronize processes, and to provide mutual exclusion.

MP/M II is designed to simplify queue management for both user and system processes. Queues are treated like disk files, and can be created, opened, written to, read from, and deleted.

The queue data structures used by MP/M II include the Queue Control Block (QCB) and the User Queue Control Block (UQCB). There are two types of Queue Control Blocks: circular or linked. The type of QCB used depends upon the size of the message the queue contains. Message sizes of 0 to 2 bytes use circular queues while message sizes of 3 or more bytes use linked queues.

3.3.1 Circular Queues

The following example illustrates how to initialize a QCB for a circular queue containing 80 messages, each of which has a one byte length. The example is shown in both PL/M and assembly language.

PL/M:

```
DECLARE CIRCULAR$QUEUE STRUCTURE (
  QL ADDRESS,
  NAME(8) BYTE,
  MSGLEN ADDRESS,
  NMBMSGS ADDRESS,
  DQPH ADDRESS,
  NQPH ADDRESS,
  MSG$IN ADDRESS,
  MSG$OUT ADDRESS,
  MSG$CNT ADDRESS,
  BUFFER (80) BYTE        )
  INITIAL (O,'CIRCQUE ',1,80);
```

Assembly Language:

```
CRCQUE:
      DS          2     ;QL
      DB          'CIRCQUE `  ; NAME
      DW          1     ;MSGLEN
      DW          80    ;NMBMSGS
      DS          2     ;DQPH
      DS          2     ;NQPH
      DS          2     ;MSGIN
      DS          2     ;MSGOUT
      DS          2     ;MSGCNT
BUFFER:
      DS          80    ;BUFFER
```

The elements of the circular queue shown above are defined in Table 3-2. The total queue overhead is 24 bytes.

Table 3-2. Circular Queue Elements

Element	Definition
QL	2-byte link, set by system.
NAME	8 ASCII character queue name, set by user.
MSGLEN	2 bytes, length of message, set by user.
NMBMSGS	2 bytes, number of messages, set by user.
DQPH	2 bytes, Dequeue list process head, set by system.
NQPH	2 bytes, Enqueue list process head, set by system.
MSG$IN	2 bytes, pointer to next message in, set by system.

Table 3-2. (continued)

Element	Definition
MSG$OUT	2 bytes, pointer to next message out, set by system.
MSG$CNT	2 bytes, number of messages in the queue, set by system.
BUFFER	n bytes, where n is equal to the message length times the number of messages. Space allocated by user, set by system. NOTE: Mutual exclusion queues require a 2-byte buffer for the owner Process Descriptor address.

3.3.2 Linked Queues

The following example illustrates how to initialize a QCB for a linked queue containing 4 messages, each 33 bytes in length.

PL/M:

```
DECLARE LINKED$QUEUE STRUCTURE (
QL ADDRESS,
NAME (8) BYTE,
MSGLEN ADDRESS,
NMBMSGS ADDRESS,
DQPH ADDRESS,
NQPH ADDRESS,
MH ADDRESS,
MT ADDRESS,
BH ADDRESS,
BUFFER (140) BYTE  )
INITIAL (O,'LNKQUE  ',33,4);
```

Assembly Language:

```
    LNKQUE:
          DS          2     QL
          DB          'LINKQUE    ; NAME
          DW          33    MSGLEN
          DW          4     NMBMSGS
          DS          2     DQPH
          DS          2     NQPH
          DS          2     MH
          DS          2     MT
          DS          2     BH
    BUFFER:
          DS          2     MSG         #1    LINK
          DS          33    MSG         #1    DATA
          DS          2     MSG         #2    LINK
          DS          33    MSG         #2    DATA
          DS          2     MSG         #3    LINK
          DS          33    MSG         #3    DATA
          DS          2     MSG         #4    LINK
          DS          33    MSG         #4    DATA
```

The elements of the linked queue shown above are defined in Table 3 3. The total queue overhead is 24 bytes.

Table 3-3. Linked Queue Elements

Element	Definition
QL	2-byte link, set by system.
NAME	8 ASCII character queue name, set by user.
MSGLEN	2 bytes, length of message, set by user.
NMBMSGS	2 bytes, number of messages, set by user.
DQPH	2 bytes, Dequeue list process head, set by system.
NQPH	2 bytes, Enqueue list process head, set by system.
MH	2 bytes, message head, set by system.
MT	2 bytes, message tail, set by system.
BH	2 bytes, buffer head, set by system.
BUFFER	n bytes where n is equal to the message Length plus two, times the number of messages. Space allocated by the user, set by the system.

3.3.3 User Queue Control Block

The User Queue Control Block (UQCB) data structure provides read/write access to queues in the same manner that an FCB provides access to a disk file. Like files, queues are "opened" by an operation that fills in the actual QCB address, which then can be read from or written to.

If the actual queue address is known, it can be used in the pointer field of the UQCB, in which case the 8-byte name field can be omitted, and an open operation is not required to access the queue. If the address is not known, then an open operation must be performed (see Function 135).

The following example illustrates how to initialize a UQCB in both PL/M and assembly language.

PL/M:

```
DECLARE USER$QUEUE$CONTROL$BLOCK STRUCTURE (
POINTER ADDRESS,
MSGADR ADDRESS,
NAME (8) BYTE         )
INITIAL (0,.BUFFER,'SPOOL   ');

DECLARE BUFFER (33) BYTE;
```

Assembly Language:

```
UQCB:
      DS    2           ;POINTER
      DW    BUFFER;     SGADR
      DB    'SPOOL   '  ;NAME
      BUFFER:
      DS    33          ;BUFFER
```

The elements of the UQCB shown above are defined in Table 3-4.

Table 3-4. UQCB Elements

Element	Definition
POINTER	2 bytes, set by system to address of actual queue during an open queue operation, or set by the user if the actual queue address is known.
MSGADR	2 bytes, address of user buffer, set by user.
NAME	8 bytes, ASCII queue name, set by user, may be omitted if the pointer field is set by the user.

3.3.4 Queue Naming Conventions

The following conventions should be used in naming queues under MP/M II: if the Terminal Message Processor (TMP) is to write directly to the queue, then the queue must have an upper-case ASCII name. Thus, when a user at a system console enters the queue name followed by a command tail, the CLI writes the command tail directly to the queue (see Section 1.5).

To make a queue inaccessible by a user at a system console, the queue name must contain at least one lower-case ASCII character. Mutual exclusion queues should be named upper-case 'MX' followed by 1 to 6 additional ASCII characters. These queues must have a two byte buffer in which the XDOS places the address of the Process Descriptor of the process owning the mutual exclusion message.

3.4 Memory Descriptor Data Structure

Each process running under MP/M II is associated with a Process Descriptor that contains a memory segment index. This index identifies a specific Memory Descriptor within MP/M II's Memory Segment Table. In MP/M II the memory segment index can have the values 0 to 7, corresponding the 8-entry Memory Segment Table, or FFH, indicating that the process is in common memory and does not use the Memory Segment Table. The XDOS uses the Memory Descriptor data structure to allocate and manage the memory resource. The Memory Descriptor contains four bytes: the memory segment base page address, the memory segment page size, the memory segment attributes, and bank. The Memory Descriptor data structure is shown below in both PL/M and assembly language.

PL/M:

```
Declare memory$descriptor structure (
base byte,
size byte,
attrib byte,
bank byte           );
```

Assembly Language:

```
MEMDES:
      DS    1     ;base
      DS    1     ;size
      DS    1     ;attributes
      DS    1     ;bank
```

The elements of the Memory Descriptor shown above are defined in Table 3-5.

Table 3-5. Memory Descriptor Elements

Element	Definition
BASE	1 byte, base page address of the memory segment, set by user.
SIZE	1 byte, size in pages of the memory segment, set by user.
ATTRIBUTE	1 byte, high-order bit "on" indicates that the memory segment is allocated, other bits are reserved for MP/M II, normally set by system, but a user may pr I e-allocate a memory segment by setting the high-order bit "on".
BANK	1 byte, bank number in the range 0 to 7, where bank 0 is the bank which is switched in when MP/M II is loaded and initialized, set by user.

3.5 System Data Page

The System Data Page is the top 256 bytes of the MP/M II Operating System. It contains static information about the system configuration which the user enters when executing GENSYS to perform system generation. It also contains dynamic information which is used by MP/M II at run time.

Table 3-6 describes the individual byte assignments within the System Data Page.

Table .3-6. System Data Page Byte Assignments

Byte	Contents
000-000	Mem$top, top page of memory
001-001	Nmb$cns, number of system consoles (TMPs)
002-002	Brkpt$RST, breakpoint RST #
003-003	Add system call user stacks, boolean
004-004	Bank switched, boolean
005-005	Z80 version, boolean
006-006	banked bdos, boolean
007-007	XIOS jump table page
008-008	RESBDOS base page
009-010	CP/NET master configuration table address
011-011	XDOS base page
012-012	RSP's (BNKXIOS top+l) base page
013-013	BNKXIOS base page
014-014	BNKBDOS base page
015-015	Maxmemseg, max memory segment number
016-047	Initial memory segment table
048-063	Breakpoint vector table, filled in by debuggers
064-079	Reserved for MP/M II
080-095	System call user stack pointer table
096-119	Reserved for MP/M II
120-121	Nmb records in MPM.SYS file
122-122	# ticks/sec
123-123	System Drive
124-124	Common Memory Base Page
125-125	Number of Rsp's
126-127	Listcp array Address
128-143	Subflg, submit flag array
144-186	Reserved for MP/M II
187-187	Max locked records/process
188-188	Max open files/process
189-190	# list items
191-192	Pointer to base of lock table free space
193-193	Total system locked records
194-194	Total system open files
195-195	Dayfile logging, boolean
196-196	Temporary file drive
197-197	Number of printers
197-241	Reserved for MP/M II
242-242	Banked XDOS base page
243-243	TMP process descriptor base
244-244	Console.dat base
245-246	BDOS/XDOS entry point
247-247	TMP.spr base
248-248	Nmbrsps, number of banked RSPs
249-249	Brsp base address
250-251	Brspl, non-resident rsp process link
252-253	Sysdatadr, XDOS internal data segment address
254-255	Rspl, resident system process link

3.6 XDOS Internal Data Segment

This section contains information regarding the location of critical variables contained in the XDOS Internal Data Segment. The information may be useful in some application programs. However, it must be accessed with caution. The information may also be useful in debugging a system by permitting access to the Ready List through the Ready List Root (RLR), both at run time as well as in a post mortem dump.

The following example, written in assembly language, illustrates a technique for accessing the Ready List Root.

```
;       MP/M Internal Data Segment Offsets

0           equ   0000h ;     time of day
osrlr       equ   0005h ;     ready list root
osdlr       equ   0007h ;     delay list root
osdrl       equ   0009h ;     dispatcher ready list
osplr       equ   000Bh ;     poll list root
osslr       equ   000Dh ;     swap list root (not used)
osqlr       equ   000Fh ;     queue list root
osthrdrt    equ   0011h ;     thread root
osnmbcns    equ   0013h ;     number of consoles
oscnsatt    equ   0014h ;     console attach table
oscnsque    equ   0034h ;     console queue
osnmbflags  equ   0054h ;     number of flags
ossysfla    equ   0055h ;     system flags
osnmbsegs   equ   0095h ;     number of memory segments
osmsegtbl   equ   0096h ;     memory segment table
ospdtbl     equ   00B6h ;     process descriptor table
osnmblst    equ   0256h ;     number of list devices
oslstatt    equ   0257h ;     list attach table
oslstque    equ   0277h ;     list queue

sysdatadr   equ   154   ;     get system data page addr

               . . .

            mvi   c,sysdatadr
            call  xdos        ; HL = system data page
            lxi   d,00fch     ; DE = offset to pointer
            dad   d
            mov   e,m
            inx   h
            mov   d,m         ; DE = base of XDOS intrl dseg
            lxi   h,osrlr     ; HL = offset to Ready List Root
            dad   d
                              ; HL = Addr of Ready List Root
```

3.7 XDOS Error Handling

The XDOS does not require an error handling capability similar to that of the BDOS, because XDOS functions involve "logical" or internal rather than "physical" or external operations. That is, the XDOS functions are implemented entirely within memory resident data structures, and any physical or extended "error" encountered would by definition be catastrophic for the system. Therefore, those XDOS functions that return a value in register A return a "boolean", which is a code indicating only whether or not the function is successful. If for some reason the function is not successful, the calling process must be able to handle this error condition. The return codes for XDOS functions are defined in Table 3-7.

Table 3-7. XDOS Return Codes

Register A Value	Meaning
0	Successful operation
FFH	Unsuccessful operation

3.8 XDOS Function Calls

The Extended Disk Operating System (XDOS) functions are covered in this section by describing the entry parameters and returned values for each XDOS function. The XDOS calling conventions are identical to those of the BDOS which are described in Section 2.1.3.

```
FUNCTION 128:ABSOLUTE MEMORY
             REQUEST
-----------------------------------
Entry Parameters:
Register    C:    80H
            DE:   MD Address

Returned Value:
Register    A: Return code
MD filled in
```

The Absolute Memory Request function allocates to the calling process a segment of memory specified by the Memory Descriptor parameter. This function allows the Command Line Interpreter (CLI) to load non-relocatable programs, such as CP/M *.COM files, based at the absolute TPA address of 0100H. The calling process passes the address of a Memory Descriptor in register pair DE, setting the base byte; the XDOS sets the other bytes upon return. The Memory Descriptor data structure is described in Section 3.4.

Function 128 returns a "boolean" indicating whether or not the allocation was successful. A returned value of FFH indicates failure to allocate the requested memory, and a value of 0 indicates success. If the Absolute Memory Request is a success, the memory segment index of the calling process is set to reflect that of the allocated memory. Thus, it is extremely important that this function only be invoked from a process residing in common memory. Note that base and size specify base page address and page size where a page is 256 bytes.

```
FUNCTION 129:  RELOCATABLE MEMORY
               REQUEST

Entry Parameters:
Register    C:    81H
            DE:   MD Address

Returned Value:
Register    A:    Return code
MD filled in
```

The Relocatable Memory Request function allocates the requested contiguous memory to the calling process. The calling process passes the address of a Memory Descriptor in register pair DE, setting the size byte; the XDOS sets the other bytes upon return

Function 129 returns a "boolean" in register A indicating whether or not the allocation was successful. A returned value of FFH indicates failure to satisfy the request, and a value of 0 indicates success. If the Relocatable Memory Request is a success, the memory segment index of the calling process is set to reflect that of the allocated memory. Thus, it is extremely important that this function only be invoked from a process residing in common memory.

Note that base and size specify base page address and page size where a page is 256 bytes.

```
FUNCTION 130: MEMORY FREE

Entry Parameters:
Register    C:    82H
DE: MD Address
```

The Memory Free function returns the specified memory segment owned by the calling process back to the operating system. The calling process passes the address of a Memory Descriptor in register pair DE. Function 130 does not return a value in register A.

```
FUNCTION 131: POLL

Entry Parameters:
Register    C:    83H
            E:    Device Number
```

The Poll function polls the specified device until a ready condition is received. The calling process relinquishes the CPU until the poll is satisfied, allowing other processes to execute.

Function 131 is intended for use in the custom XIOS because the XIOS associates the device number with the actual code executed for the poll operation. This does not exclude other uses of the Poll function, but it does mean that an application program making a poll call must be matched to a specific XIOS.

```
FUNCTION 132: FLAG WAIT

Entry Parameters:
Register    C:    84H
            E:    Flag Number

Returned Value:
Register    A:    Return code
```

The Flag Wait function causes a process to relinquish the CPU until the flag specified in the call is set. The flag wait operation is used in an interrupt-driven system to cause the calling process to "wait" until a specific interrupt condition occurs.

Function 132 returns a "boolean" in register A indicating whether or not a successful flag wait was performed. A returned value of FFH indicates that no flag wait occurred because another process was already waiting on the specified flag. A returned value of 0 indicates success.

Note that flags are non-queued, which means that access to flags must be carefully managed. Typically, the physical interrupt handlers set flags while a single process waits on each flag.

FUNCTION 133: FLAG SET		
Entry Parameters:		
Register	C:	85H
	E:	Flag Number
Returned Value:		
Register	A:	Return code

The Flag Set function "wakes up" a waiting process. The Flag Set function is usually called by an interrupt service routine after servicing an interrupt and determining which flag is to be set.

Function 133 returns a "boolean" in register A indicating whether or not a successful flag set was performed. A returned value of FFH indicates that a flag over-run has occurred; that is, the flag was already set when a flag set function was called. A returned value of 0 indicates success.

FUNCTION 134: MAKE QUEUE		
Entry Parameters:		
Register	C:	86H
DE:	QCB Address	

The Make Queue function sets up a Queue Control Block. A queue is configured as either circular or linked depending upon the message size. Message sizes of 0 to 2 bytes use circular queues while message sizes of 3 or more bytes use linked queues.

The calling process passes the address of the Queue Control Block (QCB) in register pair DE. The QCB must contain the queue name, message length, number of messages, sufficient space to accommodate the messages, and links if the queue is linked.

The QCB data structures for both circular and linked queues are described in Section 3.3.

Queues can only be created either in common memory or by user programs in non-banked systems because queues are all maintained on a linked list that must be accessible at all times. That is, a queue cannot reside in a memory segment that is bank-switched. However, a queue created in common memory can be accessed by all system and user programs.

```
FUNCTION 135: OPEN QUEUE

Entry Parameters:
Register    C:    87H
            DE:   UQCB Address

Returned Value:
Register    A:    Return code
```

The Open Queue function places the actual QCB address into the User Queue Control Block (UQCB) . Function 135 allows a user program to access queues by specifying only the queue name. The process obtains the actual address of the itself by calling Function 135, and then reads from or writes to the queue using the XDOS queue read and write functions.

Function 135 returns a "boolean" in register A indicating whether or not the open queue operation was successful. A returned value of OFFH indicates failure, while a 0 indicates success.

The user Queue Control Block data structure is described in Section 3.3.

```
FUNCTION 136: DELETE QUEUE

Entry Parameters:
Register    C:    88H
            DE:   QCB Address

Returned Value:
Register    A:    Return Code
```

The Delete Queue function removes the specified queue from the queue list. The calling process passes the address of QCB for the specified queue in register pair DE.

Function 136 returns a "boolean" in register A indicating whether or not the queue was deleted. A returned value of OFFH indicates failure, usually because some process is DQing from the queue. A returned value of 0 indicates success.

```
FUNCTION 137: READ QUEUE

Entry Parameters:
Register    C:    89H
            DE:   UQCB Address

Returned Value:
Message read
```

The Read Queue function reads a message from the queue specified by the UQCB. If no message is available at the queue, the calling process relinquishes the CPU until another process writes a message at the queue. The calling process passes the address of the UQCB in register pair DE, and when a message becomes available at the queue, Function 137 copies it into the buffer addressed by the MSGADR field of the UQCB.

```
FUNCTION 138:  CONDITIONAL READ
               QUEUE

Entry Parameters:
Register    C:    8AH
            DE:   UQCB Address

Returned Value:
Register    A:    Return code
Message read if available
```

The Conditional Read Queue function reads a message from a queue specified by the UQCB only when the queue contains a message. This function can be used to prevent the calling process from being suspended from execution if no messages exist. The calling process passes the address of the UQCB in register pair DE, and if a message is available at the queue, Function 138 copies it into the buffer addressed by the MSGADR field of the UQCB.

Function 138 returns a "boolean" in register A indicating whether or not a message was available at the queue. A returned value of OFFH indicates no message, while a zero indicates that a message was available and was copied into the user buffer.

FUNCTION 139: WRITE QUEUE
Entry Parameters: Register C: 8BH DE: UQCB Address Message to be sent

The Write Queue function writes a message to a queue specified by the UQCB. If no buffers are available at the queue, the calling process relinquishes the CPU until one becomes available. The calling process passes the address of the UQCB in register pair DE, and when a buffer is available at the queue, the function copies the buffer addressed by the MSGADR field of the UQCB into the queue. Function 139 does not return a value in register A.

FUNCTION 140: CONDITIONAL WRITE QUEUE
Entry Parameters: Register C: 8CH DE: UQCB Address Message to be sent Returned Value: Register A: Return code

The Conditional Write Queue function writes a message to queue specified by the UQCB only when a buffer is available. This function can prevent the calling process from being suspended from execution if the queue buffers are full. The calling process passes the address of the UQCB in register pair DE, and if a buffer is available at the queue, the function copies the buffer addressed by the MSGADR field of the UQCB into the actual queue.

Function 140 returns a "boolean" in register A indicating whether or not a buffer was available at the queue. A returned value of OFFH indicates no buffer, while a zero indicates that a buffer was available and that the user buffer was copied into it.

FUNCTION 141: DELAY		
Entry Parameters:		
Register	C:	8DH
	DE:	Number of Ticks

The Delay function delays execution of the calling process for the specified number of system time units, thus allowing other processes to use the CPU while the specified period of time elapses. Use of Function 141 avoids the typical programmed delay loop, which should be avoided under MP/M II because it consumes the CPU.

The system time unit is typically 60 Hz (16.67 milliseconds), but can vary according to application. For example, it is likely that in Europe it would be 50 Hz (20 milliseconds).

The calling process passes a 16-bit integer in register pair DE which specifies the number of ticks the process is to be delayed. Since calling the delay procedure is usually asynchronous to the actual time base itself, there is up to one tick of uncertainty in the exact amount of time delayed. Thus, a delay of 10 ticks guarantees a delay of at least 10 ticks, but it may be nearly 11 ticks.

FUNCTION 142: DISPATCH		
Entry Parameters:		
Register	C:	8EH

The Dispatch function causes MP/M II to determine the highest priority ready process, and then give that process the CPU. Function 142 is intended for non-interrupt driven systems in which it is desirable to enable a compute-bound process to periodically relinquish the CPU. Since all user processes usually run at the same priority, invoking Dispatch at various points in a program allows other processes access to the CPU in a round-robin fashion. Dispatch can also safely enable interrupts following the execution of a disable interrupt instruction (DI).

There are no parameters passed in register pair DE, and no values returned in register A. The process calls Function 142 by passing the function number 8EH in register C. Note: Calling Dispatch does not remove the calling process from the Ready List.

```
FUNCTION 143: TERMINATE PROCESS

Entry Parameters:
Register    C:    8FH
            D:    Conditional
                  Memory Free
            E:    Terminate Code
```

The Terminate Process function terminates the calling process, which passes parameters in registers D and E, indicating whether or not the process should be terminated if it is a system process, and if the memory segment owned by the calling process is to be released. A OFFH in the E register indicates that the process should be unconditionally terminated; a zero indicates that only a user process is to be deleted. If the calling process is a user process and register D contains a OFFH, the memory segment owned by the process is not released. Thus, a process that is a child of a parent process, both of which are executing in the same memory segment, can terminate without freeing the memory segment that is also occupied by the parent.

Function 143 does not return any value in register A. The calling process simply ceases to exist as far as MP/M II is concerned.

```
FUNCTION 144: CREATE PROCESS

Entry Parameters:
Register    C:    90H
            DE:   PD Address

Returned Value:
PD filled in
```

The Create Process function creates one or more processes by placing the passed Process Descriptors on the MP/M II Ready List.

The calling process passes the address of a Process Descriptor in register pair DE. The first field of the Process Descriptor is a link field that can point to another Process Descriptor.

Processes can only be created either in common memory or by user programs in non-banked systems because Process Descriptors are all maintained on linked lists that must be accessible at all times.

The Process Descriptor data structure is described in Section 3.2.

```
FUNCTION 145: SET PRIORITY
-----------------------------
Entry Parameters:
Register    C: 91H
            E: Priority
```

The Set Priority function sets or changes the priority of the calling process to that of the passed parameter. The calling process passes the priority in register E. Function 145 does not return a value in register A.

This function is useful when a process needs to have a high priority during an initialization phase, but after that is to run at a lower priority.

```
FUNCTION 146: ATTACH CONSOLE
-----------------------------
Entry Parameters:
Register    C:    92H
```

The Attach Console function attaches the console specified in the CONSOLE field of the Process Descriptor to the calling process. If the console is already attached to some other process, the calling process relinquishes the CPU until the other process detaches from the console. When the console becomes free and the calling process is the highest priority process waiting for the console, the attach operation takes place.

There are no parameters passed in registers D and E, and no values returned in register A. The process calls Function 146 by passing the function number 92H in register C.

```
FUNCTION 147: DETACH CONSOLE

Entry Parameters:
Register    C:    93H
```

The Detach Console function detaches from the calling process the console specified in the CONSOLE field of the Process Descriptor. If the console is not currently attached, no action takes place.

There are no parameters passed in registers D and E, and no values returned in register A. The process calls Function 147 by passing the function number 93H in register C.

```
FUNCTION 148: SET CONSOLE

Entry Parameters:
Register    C: 94H
            E: Console
```

The Set Console function detaches the currently attached console and then attaches the specified console. If the console to be attached is already attached to another process, the calling process relinquishes the CPU until the other process detaches from the console. When the console becomes available and the calling process is the highest priority process waiting for the console, the attach operation takes place.

The calling process passes the number of the console to be attached in register E. The function does not return a value in register A.

```
FUNCTION 149: ASSIGN CONSOLE

Entry Parameters:
Register    C:    95H
            DE:   APB Address

Returned Value:
Register    A:    Return code
```

The Assign Console function unconditionally assigns a console to a specified process. That is, the assignment is made regardless of whether or not any other process is currently waiting to attach the console. The calling process passes the address of a data structure called the Assignment Parameter Block (APB). This data structure contains the console number for the assignment, an 8 character ASCII process name, and a "boolean" indicating whether or not a match with the CONSOLE field of the Process Descriptor is required (true or OFFH indicates it is required).

It is extremely important to note that the calling process must own the console or the console must be currently unattached for this function to perform properly.

Function 149 returns a "boolean" in register A indicating whether or not the assignment was made. A returned value of OFFH indicates failure to assign the console, either because a Process Descriptor with the specified name could not be found, or because a match was required, and the CONSOLE field of the Process Descriptor did not match the specified console. A returned value of zero indicates a successful assignment.

```
FUNCTION 150: SEND CLI COMMAND

Entry Parameters:
Register    C:    96H
            DE:   CLICMD Address
```

The Send CLI Command function permits running processes to send command lines to the Command Line Interpreter (see Section 1.5). The calling process passes the address of a data structure called CLI Command (CLICMD) in register pair DE. This data structure contains: the default disk/user code, the console and the command line. Initialization of the CLICMD data structure is shown below in both PL/M and assembly language.

```
PL/M:
Declare CLI$command structure  (
disk$user byte,
console byte,
command$line (129) byte);
```

Assembly Language:

```
CLICMD:
      DS    1     ;default disk / user code
      DS    1     ;console number
      DS    129   ;command line
```

The default disk/user code is the first byte of the data structure. The high-order four bits contain the default disk drive and the low order four bits contain the user code. The second byte of the data structure contains the console number for the process being executed. The ASCII command line begins with the third byte and is terminated with a null byte.

It is extremely important to note that the CLI must own the console specified in the parameter of the Send CLI Command function. This assignment of the console to the CLI can be done with the Function 149, Assign Console.

Function 150 does not return a value in register A.

FUNCTION 151: CALL RESIDENT SYSTEM PROCEDURE		
Entry Parameters:		
Register	C:	97H
	DE:	CPB Address
Returned Value:		
Registers HL: Return code		

The Call Resident System Procedure function permits a process to call the optional Resident System Procedures (RSPs) . The calling process passes the address of a data structure called the Call Parameter Block, (CPB) in register pair DE. The CPB data contains the address of an 8-character ASCII RSP name followed by a two-byte parameter that the calling process passes to the RSP. Initialization of the CPB data structure is shown below in both PL/M and assembly language.

```
PL/M:
Declare CALL$PB structure  (
Name$adr address,
Param address ) initial (
.name,O);

Declare name (8) byte initial (
'Procl   ‘);


Assembly Language:

CALLPB:
      DW          NAME
      DW          0     ;parameter
NAME:
      DB          'Procl   ‘
```

Function 151 returns a 0001H in register pair HL if the RSP called is not present. Otherwise, it returns the code passed back from the RSP. Typically, a returned value of FFH indicates failure while a zero indicates success.

```
FUNCTION 152: PARSE FILENAME

Entry Parameters:
Register    C:    98H
            DE:   PFCB Address

Returned Value:
Registers HL: Return code
Parsed file control block
```

The Parse Filename function parses an ASCII file specification (FILENAME) and prepares a File Control Block (FCB). The calling process passes the address of a data structure called the Parse Filename Control Block, (PFCB) in register pair DE. The PFCB contains the address of the ASCII filename string followed by the address of the target FCB. Initialization of the PFCB data structure is shown below in both PL/M and assembly language.

```
PL/M:
Declare Parse$FN$CB structurem (
File$name$adr address,
FCB$adr address ) initial (
.file$name,.fcb );

Declare file$name (128) byte;
Declare fcb (36) byte;

Assembly Language:

PFNCB:
      DW    FLNAME
      DW    FCB

FLNAME:
      DS    128
FCB:
      DS    36
```

Function 152 assumes the file specification to be in the following form:

```
{D:}{FILENAMEH}{.TYP}{;PASSWORD}
```

where those items enclosed in curly brackets are optional.

The Parse Filename function parses the first file specification it finds in the input string. The function first eliminates leading blanks and tabs. The function then assumes that the file specification ends on the first delimiter it hits that is out of context with the specific field it is parsing. For instance, if it finds a colon (:) and it is not the second character of the file specification, the colon delimits the whole file specification. The function recognizes the following characters as delimiters:

```
space
tab
return
null
; (semicolon) - except before password field
= (equal)
< (less than)
> (greater than)
. (dot) - except after filename and before type
: (colon) - except before filename and after drive
, (comma)
[ (left square bracket)
] (right square bracket)
/ (slant)
$ (dollar)
```

If the function reaches a non-graphic character (in the range 1 through 31), not listed above, it treats it as an error.

The Parse Filename function initializes the specified FCB as follows:

byte 0	The drive field is set to the specified drive. If the drive is not specified, the default value is used. O=default, I=A, 2=B, etc.
byte 1-8	The name is set to the specified filename. All letters are converted to upper-case. If the name is not eight characters long, the remaining bytes in the filename field are padded with blanks. If the filename has an asterisk (*), all remaining bytes in the filename field are filled in with question marks (?). An error occurs if the filename is more than eight bytes long.
byte 9-11	The type is set to the specified filetype. If no type is specified, the type field is initialized to blanks. All letters are converted to upper-case. If the type is not three characters long, the remaining bytes in the file type field are padded with blanks. If an asterisk (*) occurs, all remaining bytes are filled in with question marks (?). An error occurs if the type field is more than 3 bytes long.

byte 12-15	Filled in with zeroes.
byte 16-23	The password field is set to the specified password. If no password is specified, it is initialized to blanks. If the password is not eight characters long, remaining bytes are padded with blanks. All letters are converted to upper-case. If the password field is more than eight bytes long, an error occurs.
byte 24-25	The offset of the beginning of the password in the FILENAME string is placed here. If no password is specified, this field is set to zero. It should be noted that the password indicated by this field is in the FILENAME string, which is not modified by the Parse Filename function. If there are any lower-case characters in the password, they will have to be converted to upper-case to make it the same as the password field of the FCB.
byte 26	The number of characters in the specified password is placed here. If no password is specified, this field is set to zero.

If an error occurs, all fields that have not been parsed are set to their default values, and the function returns a OFFFFh in register pair HL indicating the error.

On a successful parse, the Parse Filename function checks the next item in the FILENAME string. It skips over trailing blanks and tabs and looks at the next character. If the character is a null or carriage return, it returns a 0 indicating the end of the FILENAME string. If the next character is a delimiter, it returns the address of the delimiter. If the next character is not a delimiter, it returns the address of the delimiting blank or tab.

If the first non-blank or non-tab character in the FILENAME string is a null (0) or carriage return, the Parse Filename function returns a zero indicating the end of string, and the FCB is initialized to its default values.

If the Parse Filename function is to be used to parse a subsequent filename in the FILENAME string, the returned address should be advanced over the delimiter before placing it in the PFCB.

```
FUNCTION 153: GET CONSOLE NUMBER

Entry Parameters:
Register    C:    99H

Returned Value:
Register    A:    Console Number
```

The Get Console Number function obtains the value of the CONSOLE field from the Process Descriptor of the calling process. The calling process passes the function number 99H in register C, and the function returns the console number in register A.

```
FUNCTION 154: SYSTEM DATA ADDRESS

Entry Parameters:
Register    C:    9AH

Returned Value:
Registers HL: System Data
              Page Address
```

The System Data Address function returns the base address of the system data page. The system data page resides in the top 256 bytes of the MP/M II Operating System. It contains configuration information entered by the MP/M II GENSYS program as well as run time data including the submit flags. The contents of the system data page are described in Section 3.5.

The calling process passes the function number 9AH in register C, and the function returns the base address of the system data page in register pair HL.

FUNCTION 155: GET DATE AND TIME
Entry Parameters: Register C: 9BH DE: TOD Address Returned Value: Time and date

The Get Date and Time function returns the current encoded date and time. The calling process passes the address of a data structure called the TOD in register pair DE. The TOD data structure represents the date as a 16-bit integer, with day 1 corresponding to January 1, 1978. It represents the time as three bytes: hours, minutes, and seconds, stored as two BCD digits.

Initialization of the TOD data structure is shown below in both

PL/M and assembly language.

```
PL/M:
Declare TOD structure (
date address,
hour byte,
min byte,
sec byte );
```

Assembly Language:

```
TOD:
      DS    2     ;Date
      DS    1     ;Hour
      DS    1     ;Minute
      DS    1     ;Second
```

FUNCTION 156: RETURN PROCESS DESCRIPTOR ADDRESS		
Entry Parameters:		
Register	C:	9CH
Returned Value:		
Register	HL:	PD Address

The Return Process Descriptor Address function obtains the address of the calling processes process descriptor. By definition, this is the head of the ready list.

FUNCTION 157: ABORT SPECIFIED PROCESS		
Entry Parameters:		
Register	C:	9DH
Register	DE:	APB Address
Returned Value:		
Register	A:	Return Code

The Abort Specified Process function permits a process to terminate another specified process. The calling process passes the address of a data structure called an Abort Parameter Block (ABTPB) in register pair DE. Initialization of the ABTPB is shown below in both PL/M and assembly language.

```
PL/M:
Declare Abort$paramter$block structure (
pdadr address,
termination$code address,
name (8) byte,
console byte );
```

```
Assembly Language:
      APB:
            DS          2     ;process descriptor address
            DS          2     ;termination code
            DS          8     ;process name
            DS          1     ;console used by process
```

If the Process Descriptor address is known, it can be filled in and the process name and console can be omitted. Otherwise, the Process Descriptor address field should be a 0, and the process name and console must be specified. In either case, the termination code, which is the parameter passed to Function 143, Terminate Process, must be supplied.

FUNCTION 158: ATTACH LIST
Entry Parameters: Register C: 9EH

The Attach List function attaches the list device specified in the CONSOLE/LIST field of the Process Descriptor to the calling process. If the list device is already attached to some other process, the calling process relinquishes the CPU until the other process detaches from the list device. When the list device becomes free and the calling process is the highest priority process waiting for the list device, the attach operation takes place.

The process calls Function 158 by passing the function number 9EH in register C. The function does not return a value in register A.

FUNCTION 159: DETACH LIST
Entry Parameters: Register C: 9FH

The Detach List function detaches the list device specified in the CONSOLE/LIST field of the Process Descriptor from the calling process. If the list device is not currently attached, no action takes place.

The process calls Function 159 by passing the function number 9FH in register C. The function does not return a value in register A.

FUNCTION 160: SET LIST
Entry Parameters: Register C: AOH E: List Device

The Set List function detaches the list device currently attached to the calling process and then attaches the specified list device. If the list device to be attached is already attached to another process, the calling process relinquishes the CPU until the other process detaches from the list device. When the list device becomes free and the calling process is the highest priority process waiting for the device, the attach operation takes place.

The calling process passes the number of the list device to be attached in register E. The function does not return a value in register A.

FUNCTION 161: CONDITIONAL ATTACH LIST
Entry Parameters: Register C: AlH Returned Value: Register A: Return Code

The Conditional Attach List function attaches the list device specified in the CONSOLE/LIST field of the Process Descriptor to the calling process only if the list device is currently unattached.

If the list device is currently attached to another process, the function returns a value of OFFH in register A, indicating that the list device could not be attached. The function returns a value of zero to indicate that either the list device is already attached to the process, or that it was unattached and a successful attach operation was made.

FUNCTION 162: CONDITIONAL ATTACH CONSOLE		
Entry Parameters:		
Register	C:	A2H
Returned Value:		
Register	A:	Return Code

The Conditional Attach Console function attaches the console specified in the CONSOLE/LIST field of the Process Descriptor to the calling process only if the console is currently unattached.

If the console is currently attached to another process, the function returns a value of OFFH in register A, indicating that the console could not be attached. The function returns a value of zero to indicate that either the console is already attached to the process or that it was unattached and a successful attach operation was made.

FUNCTION 163: RETURN MP/M VERSION NUMBER		
Entry Parameters:		
Register	C:	A3H
Returned Value:		
Register	HL:	Version Number

The Return MP/M Version Number function provides information that allows version independent programming. The function returns a two-byte value, with H = 01 for MP/M II and L the MP/M II revision level.

FUNCTION 164: GET LIST NUMBER		
Entry Parameters:		
Register	C:	A4H
Returned Value:		
Register	A:	List Number

The Get List Number function returns the value of the list device from the Process Descriptor of the calling process. The process calls Function 164 by passing the function number A4H in register C. The function returns the list device number in register A.

Section 4
ASM, An 8080 Assembler

4.1 Overview

ASM reads an assembly language source file from the disk and produces 8080 machine language in Intel hex format. Invoke ASM by entering an ASM command in either of the following forms:

```
ASM filename

ASM filename.parms
```

In both cases, the assembler assumes there is a file on the disk with the name:

```
filename.ASM
```

that contains an 8080 assembly language source file. The first and second forms shown above differ only in that the second form passes parameters to the assembler to control source file access and hex and print file destinations.

In either case, MP/M II loads ASM, which prints the message:

```
MP/M ASSEMBLER VER 2.0
```

where n.n is the current version number. In response to the command, the assembler reads the source file with assumed filetype "ASM" and creates two output files:

```
filename.HEX

filename.PRN
```

The HEX file contains the machine code corresponding to the original program in Intel hex format, and the PRN file contains an annotated listing showing generated machine code, error flags, and source lines. If errors occur during translation, they are listed in the PRN file as well as at the console.

The second command form can redirect input and output files from their defaults. The "parms" portion of the command is a three letter group that specifies the origin of the source file, the destination of the hex file, and the destination of the print file. The form is:

```
filename.plp2p3
```

where pl, p2, and p3 are replaced by single letters whose meanings are defined in Table 4-1.

Table 4-1. ASM Parameters

Symbol	Valid Letters	Meaning
P1	A,B,. . ,P	designates the drive that contains the source file
p2	A,B,. . ,P	designates the drive that receives the hex file.
	z	skips the generation of the hex file
p3	A,B,. . ,P	designates the drive that receives the print file
	X	places the listing at the console
	z	skips generation of the print file

Thus, the command:

```
ASM PROG.AAA
```

indicates that the assembler takes the source file (PROG.ASM) from drive A, and also creates the hex (PROG.HEX) and print (PROG.PRN) files on drive A. This command is the default if the assembler is run from drive A without the optional parameters, as shown below:

```
OA>ASM PROG
```

The command:

```
OA>ASM PROG.ABX
```

indicates that the assembler takes the source file from drive A, place the hex file on drive B and sends the listing file to the console. The command:

```
OA>ASM PROG.BZZ
```

takes the source file from drive B and skips the generation of the hex and print files. Use this command for a fast execution of the assembler to check program syntax.

The source program format is compatible with the Intel 8080 assembler, although macros are not supported. However, certain extensions in the MP/M II assembler make it easier to use. These extensions are described below.

4.2 Program Format

An assembly language program acceptable as input to the ASM assembler consists of a sequence of statements of the form:

```
line#       label operation   operand     ;comment
```

where any or all the fields can be present in a particular instance. Each assembly language statement must be terminated with a carriage return and line feed (ED automatically inserts a line feed) or with an exclamation mark, !, which is treated as an end-of-line by the assembler. Thus, multiple assembly language statements can be written on the same physical line if separated by exclamation marks.

The line# is an optional decimal integer value representing the source program line number, which is allowed on any source line. Because these line numbers are inserted automatically by line oriented editors, ASM ignores this field if present. The label field takes either of the forms below:

```
identifier
identifier:
```

Labels are optional, except where noted in particular statement types. An identifier is a sequence of any alphanumeric characters, but the first character must be alphabetic. You can use identifiers freely to label elements such as program steps and assembler directives. only the first 16 characters are significant in an identifier, except for an embedded dollar symbol, $, which can be used to improve readability of the name. All lower-case alphabetics are treated as if they were upper-case. optionally, a colon can follow the identifier. Thus, the following are all valid labels:

```
X           xy    long$name
X:          yxl:  longer$named$data:
XlY2        Xlx2  x234$5678$9012$3456:
```

The operation field contains either an assembler directive, a pseudo operation, or an 8080 machine operation code. The pseudo operations and machine operation codes are described in Section 4.5. Section 4.4 describes the assembler directives.

The operand field of the statement generally contains an expression formed out of constants and labels, along with arithmetic and logical operations on these elements. The complete details of properly formed expressions are given in Section 4.3.

The comment field can contain any characters following the semicolon, ;, until the next real or logical end-of-line. These characters are read and listed, but are otherwise ignored by the assembler. The MP/M assembler also treats statements that begin with an asterisk, *, in column one as comment statements. These are listed and ignored in the assembly process.

The assembly language source program is a sequence of statements as defined above, optionally terminated by an END statement. ASM ignores statements following END.

4.3 Forming the Operand

To completely describe the operation codes and pseudo operations, it is necessary to first present the form of the operand field, because it appears in nearly all statements. Expressions in the operand field consist of simple operands (labels, constants, and reserved words), combined in properly formed subexpressions by arithmetic and logical operators. ASM evaluates each expression as the assembly proceeds. Each expression must evaluate to a 16-bit value. Further, the number of significant digits in the result must not exceed the intended use. That is, if an expression is to be used in a byte move immediate instruction, then the most significant 8 bits of the expression must be zero. The restrictions on the expression significance are given with the individual instructions.

4.3.1 Labels

As discussed above, a label is an identifier that appears as part of a particular statement. In general, the label is given a value determined by the type of statement that it precedes. If the label appears in a statement that generates machine code or reserves memory space (for example, a MOV instruction or a DS pseudo operation) , then the label is given the value of the program address that it labels. If the label precedes an EQU or SET, then ASM gives the label the value that results from evaluating the operand field. Except for the SET statement, an identifier can label only one statement.

When a label appears in the operand field, ASM substitutes its value during assembly. This value can then be combined with other operands and operators to form the operand field for a particular instruction.

4.3.2 Numeric Constants

A numeric constant is a 16-bit value in one of several bases. The base, called the radix of the constant, is denoted by a trailing radix indicator. The radix indicators recognized by ASM are defined in Table 4-2, below.

Table 4-2. ASM Radix Indicators

Indicator	Base
B	binary constant (base 2)
0	octal constant (base 8)
Q	octal constant (base 8)
D	decimal constant (base 10)
H	hexadecimal constant (base 16)

Q is accepted as an alternate radix indicator for octal numbers to minimize confusion between the letter 0 and the digit 0. Any numeric constant that does not terminate with a radix indicator is assumed to be a decimal constant.

A constant is thus composed as a sequence of digits, followed by an optional radix indicator, where the digits are in the appropriate range for the radix. That is, binary constants must be composed of the digits 0 and 1 octal constants can contain digits in the range 0-7, while decimal constants contain decimal digits. Hexadecimal constants contain decimal digits as well as hexadecimal digits A (10D), B (11D), C (12D), D(13D), E (14D), and F(15D).

Note that the leading digit of a hexadecimal constant must be a decimal digit so that ASM cannot confuse a hexadecimal constant with an identifier (a leading 0 will always suffice) . A constant composed in this manner must evaluate to a binary number that can be contained within a 16-bit counter; otherwise, it is truncated to the least significant 16-bits. Similar to identifiers, imbedded $'s are allowed within constants to improve their readability. Finally, ASM translates the radix indicator to upper-case if a lower-case letter is encountered. The following are all valid numeric constants:

```
1234      1234D     1100B     1111$0000$1111$OOOOB
1234H     OFFEH     33770     33$77$22Q
3377o     Ofe3h     1234d     Offffh
```

4.3.3 Reserved Words

Several reserved character sequences have predefined meanings in the operand field of a statement. The names of 8080 registers, when encountered by the assembler, are translated to the values shown in Table 4-3.

Table 4-3. 8080 Registers

Register Letter	Value
A	7
B	0
C	1
D	2
E	3
H	4
L	5
M	6
SP	6
PSW	6

Again, lower-case names have the same values as their upper-case equivalents. Machine instructions can also appear in the operand field; if so, they evaluate to their internal codes. For instructions that require operands in which the specific operand becomes a part of the binary bit pattern of the instruction (for example, MOV A,B), the value of the instruction (in this case MOV) is the bit pattern of the instruction with zeroes in the optional fields (e.g., MOV produces 40H).

When the $ symbol occurs in the operand field but not imbedded within an identifier or numeric constant, its value becomes the address of the next instruction to generate, not including the instruction contained within the current logical line.

4.3.4 String Constants

String constants represent sequences of ASCII characters, and are represented by enclosing the characters within apostrophe symbols, '. All strings must be fully contained within the current physical line, thus allowing ! symbols within strings, and must not exceed 64 characters in length. The apostrophe character can be included within a string by entering it as a double apostrophe, '', which becomes a single apostrophe when read by the assembler. Except for the DB pseudo operation, the string length is restricted to either one or two characters, which become an 8-bit or 16-bit value, respectively. Two-character strings become a 16-bit constant, with the second character as the low-order byte, and the first character as the high-order byte.

The value of a character is its corresponding ASCII code (see Appendix I). There is no case translation within strings, and so both upper- and lower-case characters can be represented. Note, however, that only graphic (printing) ASCII characters are allowed within strings. Some examples of valid strings are

```
'A'         ‘AB’        'ab’        ‘c’
‘’’’        ‘a’’’       ‘’’’’’      ’’’’″’
'Walla Walla Wash.'
'She said ''Hello'' to me.'
'I said ''Hello'' to her.'
```

4.3.5 Arithmetic and Logical Operators

The operands described above can be combined in normal algebraic notation using any combination of properly formed operands, operators, and parenthesized expressions. The operators recognized in the operand field are summarized in Table 4-4.

Table 4-4. Arithmetic and Logical Operators

Operation	Result
A + b	unsigned arithmetic sum of a and b
A - b	unsigned arithmetic difference between a and b
+ b	unary plus (produces b)
- b	unary minus (identical to 0 - b)
a * b	unsigned multiplication of a and b
a / b	unsigned division of a by b
a MOD b	remainder after a / b
a NOT b	logical inverse of b: all O's become l's, l's become O's - b is a 16-bit value
a AND b	bit-by-bit logical and of a and b
a OR b	bit-by-bit logical or of a and b
a XOR b	bit-by-bit logical exclusive or of a and b
a SHL b	the value that results from shifting a to the left by an amount b, with zero fill
a SHR b	the value that results from shifting a to the right by an amount b, with zero fill

In Table 4-4, a and b represent simple operands such as labels, numeric constants, reserved words, one- or-two character strings, or fully enclosed parenthesized subexpressions such as the examples below.

```
10+20       10h+37Q     Ll /3 (L2+4)  SHR 3
('a' and 5fh) + ‘0’     ('B'+B) OR (PSW+M)
(1+(2+c)) shr (A-(B+1))
```

Note that all computations performed at assembly time are 16 bit unsigned operations. Thus, -1 is computed as 0-1, which results in the value Offffh (i.e. , all 1 's) . The resulting expression must fit the operation code in which it is used. If, for example, the expression is used in a ADI (add immediate) instruction, then the high-order eight bits of the expression must be zero. For example, the operation "ADI -l" produces an error message because -1 becomes Offffh, which cannot be represented as an 8-bit value. "ADI (-l) AND OFFH" is acceptable because the "AND" operation zeroes the high order bits of the expression.

4.3.6 Precedence of Operators

As a convenience to the programmer, ASM assumes that operators have a relative precedence of application. This allows you to write expressions without nested levels of parentheses. Expressions have assumed parentheses defined by relative precedence. The order of application of operators in unparenthesized expressions is listed below. operators listed first have highest precedence; they are applied first in an unparenthesized expression. operators listed last have lowest precedence. Operators listed on the same line have equal precedence, and are applied from left to right as they are encountered in an expression.

```
1) / MOD SHL SHR
2) - +
3) NOT
4) AND
5) OR XOR
```

Due to this hierarchy, the expressions shown to the left below are interpreted by the assembler as the fully parenthesized expressions shown to the right:

```
a * b + c                     (a * b) + c
a + b * c                     a + (b * c)
a MOD b * c SHL d             ((a MOD b) * c) SHL d
a OR b AND NOT c + d SHL e    a OR (b AND (NOT (c + (d SHL e))))
```

Balanced parenthesized subexpressions can always override the assumed parentheses, and so the last expression above could be rewritten to force application of operators in a different order, such as:

```
(a OR b) AND (NOT c) + d SHL e
```

This expression has the assumed parentheses:

```
(a OR b) AND HNOT c) + (d SHL e) )
```

Note that an unparenthesized expression is well-formed only if the expression which results from inserting the assumed parentheses is well-formed.

4.4 Assembler Directives

Assembler directives set labels to specific values during the assembly, perform conditional assembly, define storage areas, and specify starting addresses in the program. Each assembler directive is denoted by a "pseudo operation" that appears in the operation field of the line. The acceptable pseudo operations are summarized in Table 4-5, and described individually in the following sections.

Table 4-5. ASM Directives

Symbol	Function
ORG	set the program or data origin
END	end program, optional start address
EQU	numeric "equate"
SET	numeric "set"
IF	begin conditional assembly
ENDIF	end of conditional assembly
DB	define data bytes
DW	define data words
DS	define data storage area

4.4.1 The ORG Directive

The ORG statement takes the form:

```
label ORG   expression
```

where "label" is an optional program label, and "expression" is a 16-bit expression, consisting of operands that are defined ahead of the ORG statement. The assembler begins machine code generation at the location specified in the expression. There can be any number of ORG statements within a particular program; however, there are no checks to ensure that the programmer is not defining overlapping memory areas. Note that most programs written for the MP/M II system begin with the following ORG statement:

```
ORG 100H
```

This starts machine code generation at the base of an MP/M II transient program area. To prepare a page-relocatable program for execution under MP/M II, assemble the source program twice, adding 100H to each ORG statement during the second assembly. Concatenate the two hex files generated by the assemblies using PIP, then submit the concatenated file to the GENMOD utility which produces a file of type PRL.

If a label is specified in the ORG statement, then the label is given the value of the expression. This label can then be used in the operand field of other statements to represent this expression.

4.4.2 The END Directive

The END statement is optional in an assembly language program, but if it is present it should be the last statement because all subsequent statements are ignored. The two forms of the END directive are:

```
label       END
label       END   expression
```

where the label is optional. If the first form is used, the assembly process stops and the default starting address of the program is taken as 0000. Otherwise, the expression is evaluated and becomes the program starting address. This starting address is included in the last record of the Intel-formatted machine code "hex" file which results from the assembly. Thus, most CP/M assembly language programs end with the statement:

```
END         100H
```

which results in the default starting address of 100H.

4.4.3 The EQU Directive

The EQU (equate) statement sets up synonyms for particular numeric values. The form is:

```
label       EQU   expression
```

where the label must be present, and must not label any other statement. The assembler evaluates the expression and assigns this value to the identifier given in the label field. The identifier is usually a name that describes the value in a more human-oriented manner. Then this name can be used throughout the program to 11parameterize" certain functions. Suppose for example, that data received from a teletype appears on a particular input port, and data is sent to the teletype through the next output port in sequence. The series of equate statements could define these ports for a particular hardware environment, as shown below:

```
TTYBASE           EQU   10H         ;BASE PORT NUMBER FOR TTY
TTYIN             EQU   TTYBASE     ;TTY DATA IN
TTYOUT            EQU   TTYBASE+1   ;TTY DATA OUT
```

At a later point in the program, the statements that access the teletype could appear as shown below:

```
IN          TTYIN       ;READ TTY DATA TO REG-A
. . .
OUT         TTYOUT      ;WRITE DATA TO TTY FROM REG-A
```

This makes the program more readable than if the absolute I/O ports had been used. Further, if you redefine the hardware environment to start the teletype communications ports at 7FH instead of 10H, you need only change the first statement to:

```
TTYBASE           EQU   7FH   ;BASE PORT NUMBER FOR TTY
```

and the program can be reassembled without changing any other statements.

4.4.4 The SET Directive

The SET statement is   similar to the EQU, taking the form:

```
label       SET   expression
```

It differs from SET in that the label can occur on other SET statements within the program. The expression is evaluated and becomes the current value associated with the label. Thus, the EQU statement defines a label with a single value, while the SET statement defines a value that is valid from the current SET statement to the next SET statement where the label occurs. SET is most often used to control conditional assembly.

4.4.5 The IF and ENDIF Directives

The IF and ENDIF statements define a range of assembly language statements to be included or excluded during the assembly process. The form is:

```
IF    expression
statement#1
statement#2
  . . .
statement#n
ENDIF
```

Upon encountering the IF statement, the assembler evaluates the expression following the IF. All operands in the expression must be defined ahead of the IF statement. If the least significant bit of the evaluated expression is a 1, then statement#1 through statement#n are assembled; if the least significant bit of the evaluated expression is zero, then the statements are listed but not assembled. Conditional assembly is often used to write a single 11generic" program which includes a number of possible run-time environments, with only a few specific portions of the program selected for any particular assembly. The following program segments for example, might be part of a program that communicates with either a teletype or a CRT console (but not both) by selecting a particular value for TTY before the assembly begins.

```
TRUE        EQU   OFFFFH      ;DEFINE VALUE OF TRUE
FALSE       EQU   NOT TRUE    ;DEFINE VALUE OF FALSE

TTY         EQU   TRUE        ;TRUE IF TTY, FALSE IF CRT

TTYBASE     EQU   10H         ;BASE OF TTY I/O PORTS
CRTBASE     EQU   20H         ;BASE OF CRT I/O PORTS
            IF    TTY         ;ASSEMBLE RELATIVE TO TTYBASE
CONIN       EQU   TTYBASE     ;CONSOLE INPUT
CONOUT      EQU   TTYBASE+l   ;CONSOLE OUTPUT
            ENDIF

            IF    NOT TTY     ;ASSEMBLE RELTIVE TO CRTBASE
CONIN       EQU   CRTBASE     ;CONSOLE INPUT
CONOUT      EQU   CRTBASE+l   ;CONSOLE OUTPUT
            ENDIF
            . . .
            IN    CONIN       ;READ CONSOLE DATA
            . . .
            OUT   CONOUT      ;WRITE CONSOLE DATA
```

In this case, the program assembles for an environment where a Teletype is connected, based at port 10H. The statement defining TTY could be changed to:

```
TTY   EQU   FALSE
```

and, in this case, the program assembles for a CRT based at port 20H.

4.4.6 The DB Directive

The DB directive allows the programmer to define initialized storage areas in single precision (byte) format. The statement form is:

```
      label       DB    e#1, e#2, ..., e#n
```

where e#l through e#n are either expressions that evaluate to 8-bit values (the high-order eight bits must be zero) , or are ASCII strings of length no greater than 64 characters. There is no practical restriction on the number of expressions included on a single source line. The expressions are evaluated and placed sequentially into the machine code file starting at the current program address generated by the assembler. String characters are similarly placed into memory starting with the first character and ending with the last character. Strings of length greater than two characters cannot be used as operands in more complicated expressions; that is, they must stand alone between the commas. Note that ASCII characters are always placed in memory with the parity bit reset (0), and that there is no translation from lower to upper-case within strings. The optional label can reference the data area throughout the remainder of the program. Examples of valid DB statements are:

```
      data:       DB    0,1,2,3,4,5
                  DB    data and Offh,5,377Q,1+2+3+4
      signon:     DB    ‘please type your name',cr,lf,O
                  DB    'AB' SHR 8, 'C', 'DE' AND 7FH
```

4.4.7 The DW Directive

The DW statement is similar to the DB statement except that it initializes two byte words of storage instead of single bytes. The form is:

```
      label       DW    e#1, e#2, . . ., e#n
```

e#l through e#n are expressions that evaluate to 16-bit results. Note that ASCII strings of one or two characters are allowed, but strings longer than two characters are not acceptable. In all cases, the data storage is consistent with the 8080 processor: the least significant byte of the expression is stored first in memory, followed by the most significant byte. Here are some examples:

```
      doub:       DW    Offefh,doub+4,signon-$,255+255
                  DW    ‘a’,5,’ab','CD',6 shl 8 or llb
```

4.4.8 The DS Directive

The DS statement reserves an area of uninitialized memory, and takes the form:

```
label DS    expression
```

where the label is optional. The assembler begins subsequent code generation after the area reserved by the DS. Thus, the DS statement given above has exactly the same effect as the following statements:

```
label:      EQU   $      ;LABEL VALUE IS CURRENT CODE LOCATION
            ORG          $+expression   ;MOVE PAST RESERVED AREA
```

4.5 Operation Codes

Assembly language operation codes are the principal part of assembly language programs, and form the operation field of the instruction. In general, ASM accepts all the standard mnemonics for the Intel 8080 microcomputer, which are given in detail in the Intel manual "8080 Assembly Language Programming Manual." Labels are optional on each input line and, if included, take the value of the instruction address immediately before the instruction is issued. Table 4-7 lists the individual operators briefly, but you should reference the Intel manual for detailed descriptions.

Table 4-7 lists each operation code in its most general form with a specific example, then gives a short explanation with any special restrictions. In the Form column, "en" symbolizes an expression. Table 4-6, below, defines the "en" symbols.

Table 4-6. Expression Symbols

Symbol	Represents
e3	a 3-bit value in the range 0-7 that can be one of the predefined registers A, B, C, D, E, H, L, M, SP, or PSW.
e8	an 8-bit value in the range 0-255.
e16	a 16-bit value in the range 0-65535.

The expressions can be formed from an arbitrary combination of operands and operators. In some cases, the operands are restricted to particular values within the allowable range, such as the PUSH instruction. Table 4-7 notes such cases as they are encountered.

The operation codes summarized in Table 4-7 fall into six categories. Jump, Call and Return instructions can test the condition flags set in the CPU and transfer control to another location. Immediate Operand instructions load single- or double precision registers or single-precision memory cells with constant values. These also include instructions that perform immediate arithmetic or logical operations on the accumulator (register A).

Increment and Decrement instructions are provided for both single- and double-precision registers. Data Movement instructions transfer data from memory to the CPU and from the CPU to memory. Arithmetic Logic Unit instructions perform arithmetic and logical operations on the single-precision accumulator. Control instructions enable and disable interrupts, halt program execution, and perform a no-operation function.

Table 4-7. ASM Operation Codes

Form	Example	Explanation
	Jumps, Calls and Returns	
JMP e16	JMP Ll	Jump unconditionally to label
JNZ e16	JNZ L2	Jump on non zero condition to Label
JZ e16	JZ 100H	Jump on zero condition to label
JNC e16	JNC L1+4	Jump on no carry to label
ic e16	JC L3	Jump on carry to label
JPO e16	JPO $+8	Jump on odd parity to label
JPE e16	JPE L4	Jump on even parity to label
JP e16	JP GAMMA	Jump on positive result to label
jmp e16	JM al	Jump on minus to label
CALL e16	CALL Sl	Call subroutine unconditionally
CNZ e16	CNZ S2	Call subroutine if non zero flag
CZ e16	CZ 100H	Call subroutine on zero flag
CNC e16	CNC Sl+4	Call subroutine if no carry set
cc e16	CC S3	Call subroutine if carry set
CPO e16	CPO S+8	Call subroutine if parity odd
CPE e16	CPE S4	Call subroutine if parity even

Table 4-7. (continued)

Form	Example	Explanation
		Jumps, Calls and Returns
CP e16	CP	GAMMA Call subroutine if positive Result
CM e16	CM	bl$c2 Call subroutine if minus flag
RST e3	RST 0	Programmed "restart", equivalent to CALL 8*e3, except one byte instruction
RET		Return from subroutine
RNZ		Return if non zero flag set
RZ		Return if zero flag set
RNC		Return if no carry
RC		Return if carry flag set
RPO		Return if parity is odd
RPE		Return if parity is even
RP		Return if positive result
RM		Return if minus flag is set
		Immediate Operand Instructions
MVI e3,e8		MVI B,255 Move immediate data to register A, B, C, D, E, H, L, or M (memory)
ADI e8		ADI 1 Add immediate operand to A without carry
ACI e8		ACI OFFH Add immediate operand to A with Carry
SUI e8		SUIL + 3 Subtract from A without borrow (carry)
SBI e8		SBI L AND 11B Subtract from A with borrow (carry)
ANI e8		ANI $ AND 7FH Logical "and" A with immediate data

Table 4-7. (continued)

Form	Example	Explanation
	Immediate Operand Instructions	
XRI e8	XRI 1111$0000B	"Exclusive or" A with immediate Data
ORI e8	ORI L AND 1+1 data	Logical "or" A with immediate
CPI e8	CPI 'a'	Compare A with immediate data (same as SUI except register A not changed)
LXI e3,e16	LXI B,100H	Load extended immediate to register pair (e3 must be equivalent to B, D, H, or SP)
	Increment and Decrement Instructions	
INR e3	INR E	Single precision increment register(e3 produces one of A, B, C, D, E, H, L, M)
DCR e3	DCR A	Single precision decrement Register (e3 produces one of A, B, C, D, E, H, L, M)
INX e3	INX SP	Double precision increment register pair (e3 must be equivalent to B, D, H, or SP)
DCX e3	DCX B	Double precision decrement register pair (e3 must be equivalent to B, D, H, or SP)
	Data Movement Instructions	
MOV e3,e3	MOV A,B	Move data to leftmost element from rightmost element (e3 produces one of A, B, C, D, E, H, L, or M). MOV M,M is Disallowed
LDAX e3	LDAX B	Load register A from computed address (e3 must produce either B or D)
STAX e3	STAX D	Store register A to computed address (e3 must produce either B or D)

Table 4-7. (continued)

Form	Example	Explanation
		Data Movement Instructions
LHLD e16	LHLD L1	Load HL direct from location e16 (double precision load to H and L)
SHLD e16	SHLD L5+x	Store HL direct to location e16 (double precision store from H and L to memory)
LDA e16	LDA Gamma	Load register A from address e16
STA e16	STA X3-5	Store register A into memory at e16
POP e3	POP PSW	Load register pair from stack, set SP (e3 must produce one of B, D, 11, or PSW)
PUSH e3	PUSH B	Store register pair into stack, set SP (e3 must produce one of B, D, H, or PSW)
IN e8	IN 0	Load register A with data from port e8
OUT e8	OUT 255	Send data from register A to port e8
XTHL		Exchange data from top of stack with HL
PCHL		Fill program counter with data from HL
SPHL		Fill stack pointer with data from HL
XCHG		Exchange DE pair with HL pair
		Arithmetic Logic Unit Operations
ADD e3	ADD B	Add register given by e3 to accumulator without carry (e3 must produce one of A, B, C, D, E, H, or L)
ADC e3	ADC L	Add register given by e3 to A with carry, (e3 must produce one of A, B, C, D, E, H, or L)

Table 4-7. (continued)

Form	Example	Explanation
		Arithmetic Logic Unit Operations
SUB e3	SUB H	Subtract register given by e3 from A without carry, (e3 must produce one of A, B, C, D, E, H, or L)
SBB e3	SBB 2	Subtract register given by e3 from A with carry, (e3 must produce one of A, B, C, D, E, H, or L)
ANA e3	ANA 1+1	Logical "and" of Register given by e3 with A, (e3 must produce one of A, B, C, D, E, H, or L)
XRA e3		XRA A "Exclusive or" of register given by e3 with A, (e3 must produce one of A, B, C, D, E, H, or L)
ORA e3	ORA B	Logical "or" of register given by e3 with A, (e3 must produce one of A, B, C, D, E, H, or L)
CMP e3	CMP	Compare register given by e3 with A, (e3 must produce one of A, B, C, D, E, H, or L)
DAA		Decimal adjust register A based upon last arithmetic logic unit operation
CMA		Complement bits in register A
STC		Set carry flag to 1
CMC		Complement carry flag
RLC		Rotate bits left, (re)set carry as a side effect (high-order A bit becomes carry)
RRC		Rotate bits right, (re)set carry as side effect (low order A bit becomes carry)
RAL		Rotate carry/A register to left (carry is involved in the rotate)

Table 4-7. (continued)

Form	Example	Explanation
		Arithmetic Logic Unit Operations
RAR		Rotate carry/A register to right (carry is involved in the rotate)
DAD e3	DAD B	Double precision add register pair e3 to HL (e3 must produce B, D, H, or SP)
		Control Instructions
HLT		Halt the 8080 processor
DI		Disable the interrupt system
EI		Enable the interrupt system
NOP		No operation

4.6 Error Messages

When ASM finds errors within the assembly language program, it lists them as single-character codes in the leftmost position of the source listing. The line in error is also echoed at the console so that the source listing need not be examined to determine if errors are present. Table 4-8 defines the error codes.

Table 4-8. Assembly Error Codes

Code	Meaning
D	Data error: element in data statement cannot be placed in the specified data area
E	Expression error: expression is ill-formed and cannot be evaluated
L	Label error: label cannot appear in this context may be duplicate label
N	Not implemented: features that will appear in future ASM versions (e.g., Macros) are recognized and flagged, but are unsupported in this version
O	Overflow: expression is too complicated, has too many pending operators to be computed; simplify it
P	Phase error: label does not have the same value on two subsequent passes through the program
R	Register error: the value specified as a register is not compatible with the operation code
S	Syntax error: the fields of this statement are ill Formed and cannot be processed properly; may be due to invalid characters or misplaced delimiters
U	Undefined symbol: label operand in this statement has not appeared elsewhere on the left side of a statement that generates machine code or reserves memory space, as in a MOV instruction, a DS pseudo operation, or an EQU or SET directive
V	Value error: operand encountered in expression is improperly formed

Several error message can be printed at the terminal if a disk error condition occurs. Table 4-9 summarizes these error messages.

Table 4-9. ASM Terminal Messages

Message	Meaning
NO SOURCE FILE PRESENT	The file specified in the ASM command does not exist on disk.
NO DIRECTORY SPACE	The disk directory is full; erase files that are not needed, then retry.
SOURCE FILE NAME ERROR	Improperly formed ASM filename wildcard ? and * characters are not allowed.
SOURCE FILE READ ERROR	Source file cannot be read properly by the assembler; type file at console to determine the point of error.
OUTPUT FILE WRITE ERROR	output files cannot be written properly; most likely cause is a full disk; erase and retry.
CANNOT CLOSE FILE	Output file cannot be closed; check to see if disk is write protected.

Section 5
RDT

5.1 RDT Overview

The Relocatable Debugging Tool (RDT) allows the user to test and debug programs in the MP/M II environment. Multiple RDTs can be relocated for execution in a non-banked system, or assigned absolute memory locations for execution in a bank-switched system. RDT commands are a superset of the CP/M debugger, DDT (see the command summary in Table 5-1.) The additional commands allow RDT to debug relocatable code and save patched programs. However, there is one important difference between RDT and DDT. RDT is a PRL file and DDT is a COM file. Thus RDT can debug both COM and PRL files, while DDT can only debug COM files. Note: RDT cannot read a file that is password protected.

5.2 Invoking RDT

Invoke RDT by entering one of the following commands:

```
RDT
RDT filespec
```

The first command simply loads and executes RDT. After displaying its sign-on message and prompt character, RDT is ready to accept operator commands. The second command is similar to the first, except that after RDT is loaded, it loads the file specified by filespec.

The second command is equivalent to first invoking RDT and then using the I (Input) command to insert a filename into the default FCB at Base+005CH, as shown in the following sequence.

```
OA>RDT
00:22:55 A:RDT          .PRL  (USER n)
[MP/M] DDT VERS 2.0
NEXT PC
0100 0100
-1 filespec
-R
-
```

At this point, the program named by filespec is loaded and ready for debugging.

5.3 RDT Command Conventions

When RDT is ready to accept a command, it prompts the user with a hyphen, -. In response, the user can type a command line or a CONTROL-C (represented as ^C) to end the debugging session (see Section 5.4) . A command line can have up to 32 characters, and must be terminated with a carriage return. While entering commands, use the standard MP/M II line-editing functions (^X, ^H, etc.) to correct typing errors. RDT does not process the command line until a carriage return is entered.

The first character of each command line determines the command action. Table 5-1 summarizes RDT commands. RDT commands are defined individually in Section 5.5.

Table 5-1. RDT Command Summary

Command	Action
A	enter assembly language statements
*B	set or reset bitmap bits
D	display memory in hexadecimal and ASCII
F	fill memory block with a constant
G	begin execution with optional breakpoints
I	set up file control block and command tail
L	list memory using assembler mnemonics
M	move memory block
*N	normalize and relocate program to RDT's memory segment
R	read disk file into memory
S	set memory to new values
T	trace program execution
U	untraced program monitoring
*V	compute parameter value for W command
*W	write contents of memory block to disk
X	examine and modify CPU state

*RDT only

The command character can followed by one or more arguments, which may be hexadecimal values, filenames or other information, depending on the command. RDT assumes all values the user enters are hexadecimal. If the user enters more than four digits, RDT truncates them on the left; that is, RDT only uses the last four. Arguments should be separated from each other by commas or spaces. Note: no spaces are allowed between the command character and the first argument.

5.4 Terminating RDT

The user terminates RDT by typing a TC in response to the hyphen prompt. RDT responds with the query:

```
Abort (Y/N) ?
```

Note: MP/M II does not have the SAVE facility found in CP/M so if RDT is used to patch a file, the user must write the file to disk using the W command before terminating RDT.

5.5 RDT Commands

This section defines RDT commands and their arguments. RDT commands give the user control of program execution and allow the user to display and modify system memory and the CPU state.

5.5.1 The A (Assemble) Command

The A command assembles 8080 mnemonics directly into memory. The form is:

```
Aa
```

where a is the hexadecimal address where assembly is to start. RDT responds to the A command by displaying the address of the memory location where assembly is to begin. At this point, the user enters assembly language statements. When a statement is entered, RDT converts it to machine code, places the value(s) in memory, and displays the address of the next available memory location. This process continues until the user enters a blank line or a line containing only a period.

RDT responds to invalid statements by displaying a question mark,?, and redisplaying the current assembly address.

5.5.2 The B (Bitmap Bit Set/Reset) Command

The B command enables the user to update the bitmap of a page relocatable file. The user reads the file in, makes changes to the code, and then determines the bytes that need relocation (i.e. the high-order address bytes of jump instructions). The user then updates the bitmap with the B command. There are two parameters specified: the address to be modified, followed by a 0 to reset a byte previously marked for relocation or a 1 to mark a byte for relocation. The form is:

```
Ba,n
```

where a is the hexadecimal address, and n is either a 0 or 1.

5.5.3 The D (Display) Command

The D command displays the contents of memory as 8-bit hexadecimal values and in ASCII. The forms are:

```
D
Ds
Ds,f
```

where s is the hexadecimal address where the display is to start, and f is the address where the display is to finish. In response to the first form shown above, RDT displays memory from the current display address for 16 display lines. The response to the second form is similar to the first, except that the display address is f first set to the address s. The third form displays the memory block between locations s and f.

Memory is displayed on one or more display lines. Each display line shows the values of up to 16 memory locations. For the first three forms, the display line appears as follows:

```
aaaa bb bb . . . bb cc . . . c
```

where aaaa is the display address in hexadecimal: bbs represents the contents of the memory locations in hexadecimal, and cs represents the contents of memory in ASCII. Any non-displayable ASCII characters are represented by periods.

During a long display, type any character at the console to abort the D command.

5.5.4 The F (Fill) Command

The F command fills an area of memory with a byte constant. The form is:

```
Fs,f,b
```

where s is a hexadecimal starting address of the block to be filled, f is the ending address, and b is the hexadecimal byte constant. RDT stores the 8-bit value b in locations s through f by first storing b at address s, then incrementing the value of s and testing it against f. The process repeats until s exceeds f.

5.5.5 The G (Go) Command

The G command transfers control to the program being tested, and optionally sets one or two breakpoints. The forms are:

```
G
G,bl
G,bl,b2
Gs
Gs,bl
Gs,bl,b2
```

where s is a hexadecimal address where program execution is to start, and bl and b2 are hexadecimal addresses of breakpoints.

In the first three forms, no starting address is specified, so RDT starts execution of the program under test at the current value of the program counter. (Use an X command to determine the current value of the counter). The first form transfers control to the user's program without setting any breakpoints. The next two forms respectively set one and two breakpoints before passing control to the user's program. The next three forms are analogous to the first three, except that the user's program counter is first set to s.

Once control is transferred to the program under test, it executes in real time until a breakpoint is encountered. At this point, RDT regains control, clears all breakpoints, and indicates the address at which execution of the program under test was interrupted as follows:

```
*aaaa
```

where aaaa is the hexadecimal address where the break occurred. When a breakpoint returns control to RDT, the instruction at the breakpoint address has not yet been executed.

5.5.6 The I (Input Command Tail) Command

The I command inserts a filename into the default FCB at Base+005CH, relative to the base of the segment in which RDT is loaded. The form is:

```
I<command tail>
```

where <command tail> is a character string that usually contains one or more filenames. The first filename is parsed into the default FCB at Base+005CH. The optional second filename (if specified) is parsed into the second part of the default FCB beginning at Base+006CH.

5.5.7 The L (List) Command

The L command lists the contents of memory in assembly language. The forms are:

```
L
Ls
Ls,f
```

where s is the hexadecimal address where the list is to start, and f is the hexadecimal address where the list is to finish.

The first form lists 11 lines of disassembled machine code from the current list address. The second form sets the list address to s and then lists 11 lines of code. The last form lists disassembled code from s through f. In all three cases, the list address is set to the next unlisted location in preparation for a subsequent L command. When RDT regains control from a program being tested (see G, T and U commands) , the list address is set to the current value of the program counter.

Abort long displays by typing any key during the list process. Or, enter ^S to halt the display temporarily.

5.5.8 The M (Move) Command

The M command moves a block of data values from one area of memory to another. The form is:

```
Ms,f,d
```

where s is the hexadecimal starting address of the block to be moved; if is the address of the final byte to be moved, and d is the destination address of the f first byte to receive the data. Note: if d is between s and f, part of the block being moved is overwritten before it is moved, because data is transferred starting from location s.

5.5.9 The N (Normalize) Command

The N command adjusts the relocatable addresses of the page relocatable file that RDT reads into memory. The user reads the file into memory with the R command, and then uses the N command to prepare the file for execution within the memory segment where RDT is executing. The form is:

```
N
```

5.5.10 The R (Read) Command

The R command is used in conjunction with the I command to read files from disk into the TPA in preparation for debugging. The forms are:

```
R
Rb
```

where b is an optional bias address that is added to each program or data address as it is loaded. The load operation must not overwrite any of the system parameters from OOOH through OFFH (i.e., the base page of the TPA where RDT is loaded) . If b is omitted, RDT assumes b=OOOOH. The R command requires a previous I command, specifying a valid filename. The load address for each record from a HEX file is obtained from each individual HEX record. RDT assumes any file specified as type COM contains machine code in the Intel hex format. Other files are assumed to be in pure binary format.

The user can issue any number of R commands following an I command to re-read the program being debugged, if the program does not destroy the default FCB at Base+005CH.

Recall that the sequence:

```
OA>RDT
-Ifilespec
-R
```

is equivalent to:

```
OA>RDT filespec
```

When the user issues the R command, RDT responds with a load message in the form:

```
NEXT   PC
nnnn pppp
```

where nnnn is the next address following the loaded program, and pppp is the assumed program counter taken from the last record if a HEX file is specified; for other files, it is assumed to be the base of the TPA.

5.5.11 The S (Set) Command

The S command can examine or alter the contents of an individual byte in memory. The form is:

```
Sa
```

where a is the hexadecimal address to be examined or altered.

RDT displays the memory address and its current contents on the following line in the following form:

```
aaaa bb
```

where aaaa is the hexadecimal address, and bb is the byte contents of memory in hexadecimal. The user can then choose to alter the memory location or to leave it unchanged. If the user enters a valid hexadecimal value, RDT replaces the contents of the byte in memory with the new value. If no value is entered, the contents of memory are unaffected and the contents of the next address are displayed. In either case, RDT continues to display successive memory addresses and values until either a period or an invalid value is entered.

5.5.12 The T (Trace) Command

The T command traces program execution for 1 to OFFFFH program steps. The forms are:

```
T
Tn
```

where n is the number of instructions to execute before returning control to the console.

In response to the first form, RDT displays the CPU state and executes the next program step. The program terminates immediately, with the termination address displayed in the form:

```
*hhhh
```

where hhhh is the next address to execute. The user sets the display address (used in the D command) to the value of registers H and L, and sets the list address (used in the L command) to the value hhhh. The user can then examine the CPU state at program termination by using the X command.

The second form is similar to the first, except that RDT traces program execution for n steps before a breakpoint occurs. The user can force a breakpoint in the trace mode by typing a rubout character. RDT again displays the CPU state before each program step in the same format as described in the X command.

In either case, RDT transfers control to the program under test at the address indicated by the program counter. If the user does not specify n, RDT executes only one instruction. The user can abort a long trace before n steps are executed by typing any character at the console.

Note: Program tracing stops at the interface to MP/M II, and resumes after return from MP/M II to the program under test. Thus, MP/M II functions that access I/O devices such as disk drives, run in real time and avoid I/O timing problems. Programs running in trace mode execute approximately 500 times slower than real time since RDT gets control after each instruction is executed.

5.5.13 The U (Untrace) Command

The U command is identical to the T command except that the CPU state is displayed only before the first instruction is executed, rather than before every step. The forms are:

```
U
Un
```

where n is the number of instructions to execute before returning control to the console. Abort U command by striking any key at the console.

5.5.14 The V (Value) Command

The V command facilitates use of the W command by computing the parameter to follow the "W". A single parameter immediately follows the "V" which is the NEXT location following the last byte to be written to disk.

Normally, the user reads in the file, edits it, and then writes it back to disk. The read command, R, produces -a value for NEXT. This value can be entered as a parameter following the V command, and RDT computes and displays the number of sectors to be written out using the W command. The form is:

```
V
```

5.5.15 The W (Write) Command

The W command writes the contents of a contiguous block of memory to disk. The form is:

```
Wn
```

where n is the value of the parameter obtained from the V command, and is the number of sectors to be written to disk. The user enters the value for n in hexadecimal.

5.5.16 The X (Examine CPU State) Command

The X command allows the user to examine and alter the CPU state of the program under test. The forms are:

```
X
Xr
Xf
```

where r is the name of one of the 8080 CPU registers and f is the abbreviation of one of the CPU flags. The first form displays the CPU state in the format:

```
CfZfMfEfIf A=bb B=dddd D=dddd H=dddd S=dddd P=dddd inst
```

where f is the flag value, 0 or 1; bb is the byte value, and dddd is the double byte quantity corresponding to the register pair. The "inst" field contains the disassembled instruction that occurs at the location addressed by the CPU state's program counter.

The second form displays and allows the user to alter the register values, where r is one of the registers listed in Table 5 2.

The third form displays and allows the user to alter the values of the flags listed in Table 5-2.

Table 5-2. 8080 CPU Flags and Registers

Flag	Definition	Values
C	Carry Flag	(0/1)
Z	Zero Flag	(0/1)
M	Minus Flag	(0/1)
E	Even Parity Flag	(0/1)
I	Interdigit Carry	(0/1)
Register	Definition	Values
A	Accumulator	(0-FF)
B	BC register pair	(0-FFFF)
D	DE register pair	(0-FFFF)
H	HL register pair	(0-FFFF)
S	Stack Pointer	(0-FFFF)
P	Program Counter	(0-FFFF)

In each case, RDT first displays the flag or register value, and then accepts input at the console. If the user enters a value in the proper range, RDT alters the flag or register. Entering a carriage return does not alter the value. Note: RDT displays the BC, DE, and HL registers as register pairs. Thus, typing B alters the BC register pair; D alters the DE register pair, etc.

Section 6
Other Programming Utilities

6.1 GENHEX

Syntax:

```
genhex{d:}filename{.typ}xxx
```

GENHEX accepts a COM file as input and changes it to a HEX file. This utility is useful for generating HEX files as input for the GENMOD utility.

If no filetype is specified, GENHEX assumes a type of COM. In the syntax line above, xxx is the offset specified for the HEX file. GENHEX is non-destructive. That is, it does not alter the original COM file. The following is an example of a GENHEX command:

```
OA>GENHEX B:PROGRAM.COM 100
```

6.2 GENMOD

Syntax:

```
genmod {d:}filename.hex{d:}filename.prl$nnnn
```

GENMOD produces a PRL file from a HEX file. The user first concatenates two HEX files generated from the same source file. The HEX files are offset from each other by 100H bytes. GENMOD accepts the concatenated file as input, and then prepares a PRL file by generating a header record, a code and data segment, and a bit map.

In the syntax line above, nnnn is an optional parameter that can be used to specify an additional amount of memory required by the program beyond the code space. The form of parameter is 11$11 followed by four HEX digits. For example, if a program is written to use all of available memory for buffers, specifying the optional parameter ensures a minimum buffer allocation. GENMOD is non destructive. That is, it does not alter the original HEX file. The following is an example of a GENMOD command:

```
OA>GENMOD B:PROGRAM.HEX A:PROGRAM.PRL.$1000
```

6.3 PRLCOM

Syntax:

```
prlcom{d:}filename.prl{d:}filename.com
```

PRLCOM accepts a source PRL file, and produces a COM file by removing the header record and the bit map. PRLCOM is non destructive. That is, it does not alter the original PRL file. The destination file can be on the same or a different drive, but if it already exists, PRLCOM queries the user:

```
Destination File Exists, delete (Y/N)?
```

Responding with N aborts PRLCOM. The following is an example of a PRLCOM command:

```
OA>PRLCOM B:PROGRAM.PRL A:PROGRAM.C
```

6.4 DUMP

Syntax:

```
dump {d:}filename.typ
```

DUMP displays the contents of a disk file in hexadecimal format. The following is an example of a DUMP command:

```
OA>DUMP PROGRAM.COM
```

The filename must be unambiguous (i.e. no wildcard characters) Note: DUMP does not display the contents of a password protected file. DUMP displays the file's contents at the console, 16 bytes at a time, with the absolute byte address listed to the left of each line as shown in the example below:

```
0000 CD 8A 02 1F D2 10 02 CD 58 02 32 64 03 CD D3 02 0010
CD 71 02 43 66 D9 01 57 OE 01 2D F5 05 3A 2E 04 0020 FE CA
A2 E5 B3 32 02 E6 45 00 00 00 00 00 00 00 00 0030 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00
 .
 .
 .
```

The user can send the output to the printer by typing CONTROL-P before entering the DUMP command, and start and stop the output at the console with CONTROL-S/CONTROL-Q. Type any key to abort the DUMP program.

6.5 LOAD

Syntax:

```
load {d:}filename{.typ}
```

LOAD accepts as input a HEX file and produces as output a COM file, which can then be executed. LOAD is non-destructive. That is, it does not alter the original HEX file. If no filetype is specified in the command line, LOAD looks for a file of type HEX on the disk. The following is an example of a LOAD command:

```
OA>LOAD B:PROGRAM.HEX
```

Section 7
PRL File Generation

7.1 PRL Format

A Page Relocatable Program is stored on disk as a file of type .PRL. The format is shown in Table 7-1.

Table 7-1. PRL File Format

Address	Contents
0001-0002H	Program size
0004-0005H	Minimum buffer requirements (additional memory)
0006-OOFFH	Currently unused, reserved for future allocation

0100H + Program size = Start of bit map

The bit map is a string of bits identifying those bytes in the source code that require relocation. There is one byte in the bit map for every 8 bytes of source code. The most significant bit (bit 7) of the f first byte of the bit map indicates whether or not the first byte of the source code requires relocation. If the bit is lion", it indicates that relocation is required. The next bit, (bit 6), of the first byte corresponds to the second byte of the source code, and so forth.

7.2 Generating a PRL

The preferred technique for the generation of a . PRL file is to use the Digital Research Link-80 which is capable of generating a .PRL file from a REL relocatable object file. This technique is described in detail in the Link-80 Manual. A sample link command is shown below:

```
OA>Iink dump[op]
```

An alternate method of generating a PRL file is to use the Digital Research assembler, ASM. This technique is described below.

A Page Relocatable Program can be generated by assembling the source code twice; during the second assembly, 100H is added to each ORG statement. The two HEX files generated by assembling the source code twice are concatenated with PIP, and the resulting file is then provided as input to GENMOD, which produces an output file of type PRL. The GENMOD utility is described in Section 6.2.

Appendix G contains a sample Page Relocatable Program. The code in the example program illustrates the required use of ORG statements to access the BDOS and the default FCB. The following points should be noted:

- O The initial ORG is at OOOOH. This establishes the equate for the symbol BASE, the base of the relocatable segment.

- O The ORG 100H statement establishes the actual beginning of code for the program. During the second assembly, these two ORG statements are changed to 100H and 200H, respectively.

- O The first assembly generates a file that can be changed into an executable COM file with the LOAD utility. In fact, it is desirable to first debug the program as a COM file and then make it a PRL file.

- O It is vital to use BASE to offset all memory segment base page references. The program cannot make BDOS calls to absolute 0005H. In this example, BASE is used to offset the BDOS, FCB, and BUFF equates. If a program needs to determine the top of its memory segment, the following equate and code sequence can be used:

```
MEMSIZE     EQU   BASE+6

            LHLD  MEMSIZE ;HL = TOP OF MEMORY SEGMENT
```

The following steps show how to generate a Page Relocatable File for this example using the Digital Research Assembler, ASM:

1. Prepare the program, DUMP.ASM in the example, with proper ORG statements as described above.

2. Assuming a system disk is on drive A and the DUMP.ASM file is on drive B, enter the commands:

```
1A>ASM B:DUMP
      ;assemble the DUMP.ASM file
1A>ERA B:DUMP.HXO
1A>REN B:DUMP.HX0=B:DUMP.HEX
1A>PIP LST:=B:DUMP.PRN[T8]
1A>ERA B:DUMP.PRN
```

3. Edit the DUMP.ASM file, adding 100H to each ORG statement. This can be done by concatenating a preamble to the program that contains the two initial ORG statements. A submit file to perform this function, named ASMPRL.SUB is provided on the distribution diskette.

```
1A>ASM B:DUf4P.BBZ
      ;assemble the DUMP.ASM file a second time
1A>PIP B:DUMP.HEX=B:DUMP.HXO,B:DUMP.HEX
      ;concatenate the HEX files
1A>GENMOD B:DUMP.HEX B:DUMP.PRL
      ;generate the relocatable DUMP.PRL file
```

Section 8
RSP Generation

8.1 RSPs and Resident System Procedures

Resident System Processes are included with MP/M II during system generation. GENSYS searches the directory for all files with the type RSP, displays the filenames, and then prompts the user as to whether it should be included in the generated system file, MPM.SYS.

MP/M II also supports a special type of RSP called the Resident System Procedure. A Resident System Procedure provides a method of serially utilizing a block of code as a system resource. A Resident System Procedure is created by an RSP. The RSP creates a queue with the name of the Resident System Procedure and sends it a single two byte message containing the address of the procedure to be accessed serially. The RSP then terminates itself.

8.2 Generating an RSP

The method of generating an' RSP is analogous to that of generating a Page Relocatable Program (as described in Section 7) with the following exceptions:

- If LINK-80 is used, the output file type of RSP is specified with the [or] option.
- The GENMOD output file is designated RSP rather than PRL
- The code in the RSP is ORGed at OOOH rather than 100H.

8.3 RSP Code

Appendix F contains a sample Resident System Process. The code in the example program illustrates the required structure of an RSP as well as the BDOS/XDOS access mechanism. This example should be studied carefully and the following points noted:

- The first two bytes of an RSP are set to the address of the BDOS/XDOS entry point. The address is filled in by the loader; an RSP can simply access the BDOS/XDOS by loading HL from the base of the program area and then executing a PCHL instruction.
- The Process Descriptor for the RSP must immediately follow the first two bytes that contain the address of the BDOS/XDOS entry point. It is important to note the manner in which the Process Descriptor is initialized. DS

instructions are used where storage is simply allocated, while DB and DW instructions are used where data in the Process Descriptor must be initialized. Note: the stack pointer field of the Process Descriptor points to the address immediately following the stack allocation. This is the return address and is the actual process entry point.

- O The HEX file generated by assembling the RSP must span the entire program and data area. To ensure this, use a DW instruction to define the first two bytes of the RSP that contain the address of the BDOS/XDOS entry point. Using a DS instruction does not generate any HEX file code for those two bytes. The end of the program and data area must be defined in a similar manner. If the RSP has DS instructions preceding the END statement, it is necessary to place a DB statement after the DS statements and before the END statement.

8.4 Banked RSPs

MP/M II supports a form of an RSP called a banked RSP which consists of two parts: a resident portion and a banked portion. The resident portion has the Process Descriptor for the RSP, and any other data structures such as queues, which must be in common memory. The banked portion of filetype BRS contains the remainder of the RSP, usually including the actual code, stack and other data structures. The resident portion of a banked RSP must follow the rules given in the previous section for RSPs. The presence of a banked portion of an RSP is specified by setting the Process Descriptor memory segment index to zero rather than FFH. The name provided in the Process Descriptor is used to obt6in the banked portion of the RSP that has a file type of BRS. The following points should be noted about a BRS:

- o Bytes 0000-0001H of the banked RSP are reserved for the address of the resident portion of the RSP. Thus, a banked RSP must access the BDOS/XDOS functions by indirectly loading from the two bytes at relative 0000-0001H which point to the base of the resident portion of the RSP which in turn contain the BDOS/XDOS entry point address.

- O Bytes 0002-0003H of the banked RSP must contain the initial stack pointer value for the process. Thus the stack for the banked RSP is in the banked portion of the RSP and should be initialized such that the return address on top of the stack is the banked RSP entry point address.

- O Bytes 0004-OOOBH of the banked RSP must contain an ASCII name for the process. This is used for display purposes during GENSYS and MPMLDR execution.

Section 9
SPR Generation

9.1 System Page Relocatable Files

System Page Relocatable Files are placed in the MPM.SYS file during system generation. A number of SPR files are provided as part of the standard MP/M II: the resident and banked portions of the BDOS, named RESBDOS.SPR and BNKBDOS.SPR; the resident and banked portions of the XDOS, named XDOS.SPR and BNKXDOS.SPR; and the banked TMP, named TMP.SPR.

Another SPR file named the RESXIOS.SPR or BNKXIOS.SPR contains a user customized XIOS that is unique to the hardware on which MP/M II is executing. This section gives an overview of the generation technique for this custom SPR. A detailed discussion of the generation of RESXIOS.SPR or BNKXIOS.SPR is provided in Section 1.3 of the MP/M II System Guide.

9.2 Generating an SPR

The method of generating an SPR is analogous to that of generating a Page Relocatable Program (as described in Section 7) with the following exceptions:

- o if LINK-80 is used, the output file of type SPR is specified with the [os] option.
- O the GENMOD output file is designated SPR rather than PRL.
- O the code in the SPR is ORGed at OOOH rather than 100H.

Appendix A
Flag Assignments

```
+-------+
:  0    :   Reserved
+-------+
:  1    :   System time unit tick
+-------+
:  2    :   One second interval
+-------+
:  3    :   One minute interval
+-------+
:  4    :   Undefined
:       :
:   .   :   .
:       :
:   .   :   .
:       :
:   .   :   .
+-------+
:   31  :
+-------+
```

Figure A-1. Flag Assignments

Appendix B
Process Priority Assignments

0 - 31	Interrupt handlers
32 - 63	System processes
64 - 197	Undefined
198	Terminal Message Processes
199	Command Line Interpreter
200	Default user priority
201 - 254	User processes
255	Idle process

Appendix C
BDOS Function Summary

Table C-1. BDOS Function Summary

Number	Function Name	Input Parameters	Returned values
0	System Reset	none	none
1	Console Input	none	A = char
2	Console Output	E = char	none
3	Raw Console Input	none	A = char
4	Raw Console Output	E = char	none
5	List Output	E = ch6r	none
6	Direct Console I/O	see def	see def
7	Get I/O Byte Not supported under MP/M II		
8	Set I/O Byte Not supported under MP/M II		
9	Print String	DE = .Buffer	none
10	Read Console Buffer	DE = .Buffer	see def
11	Get Console Status	none	A = 00/01
12	Return Version Number	none	HL= Version#
13	Reset Disk System	none	see def
14	Select Disk	E = Disk Number	see def
15	open File	DE = .FCB	A = Dir Code
11	Close File	DE = .FCB	A = Dir Code
17	Search for First	DE = .FCB	A = Dir Code
18	Search for Next	none	A = Dir Code
19	Delete File	DE = .FCB	A = Dir Code
20	Read Sequential	DE = .FCB	A = Err Code
21	Write Sequential	DE = .FCB	A = Err Code
22	Make File	DE = .FCB	A = Dir Code
213	Rename File	DE = .FCB	A = Dir Code
24	Return Login Vector	none	HL= Login Vect*
25	Return Current Disk	none	A = Cur Disk#
26	Set DMA Address	DE = .DMA	none
27	Get Addr(Alloc)	none	HL= Alloc
28	Wtite Protect Disk	none	see def
29	Get R/O Vector	none	HL= R/O Vect*
30	Set File Attributes	DE = .FCB	see def
31	Get Addr(disk parms)	none	HL= DPB
32	Set/Get User Code	see def	see def
33	Read Random	DE = .FCB	A = Err Code
34	Write Random	DE = .FCB	A = Err Code
35	Compute File Size	DE = .FCB	ro, rl, r2
36	Set Random Record	DE = .FCB	rO, 11, L2
37	Reset Drive	DE = drive Vect	A = Err Code
38	Access Drive	DE = drive Vect	none
19	Free Drive	DE = drive Vect	none
40	Random w 0-fiil	DE = .FCB	A = Err Code
41	Test and Write Record	DE = .FCB	HL = Err Code
42	Lock Rtcord	DE = .FCB	HL = Err Code
		(Current DMA Addr	-> File ID)

Table C-1. (continued)

Number	Function Name	Input Parameters	Returned values
43	Unlock Record	DE = .FCB (Current DMA ADDR	HL = Err Code -> File ID)
44	Set Multi-Sector Count	E = #of Sectors	A = Rtn Code
45	Set BDOS Error Mode	see def	none
46	Get Disk FrLe Spece	E = Disk #	see def
47	Chain To Program	see def	none
48	Flush Buffers	none	see def
100	Set Directory Label	DE = .FCB	HL = Dir Code
101	Return Directory Label	E = Disk #	A = Label Data
102	Read File XFCB	DE = .XFCB	HL = Dir Code
103	Write File XFCB	DE = .XFCB	HL = Dir Code
104	Set Date and Time	DE = .TOD	none
105	Get Date and Time	DE = .TOD	none
106	Set Default Password	DE = .Password	none
107	Return Serial Number	DE = .serialnmb	serialnmb set

The following abbreviations are used in the table.

char = ASCII character
Dir = Directory
Err = Error
Vect = Vector

Note: DE refers to register pair DE; HL refers to register pair HL, and that A = L, and B = H upon return.

Appendix D
XDOS Function Summary

Table D-1. XDOS Function Summary

Number	Function Name	Input Parameters	Returned values
128	Absolute Memory Rqst	DE = .MD	A = Err Code
129	Relocatable Mem Rqst	DE = .MD	A = Err Code
130	Memory Free	DE = .MD	none
131	Poll	E = Device	none
132	Flag Wait	E = Flag	A = Err Code
133	Flag Set	E = Flag	A = Err Code
134	Make Queue	DE= .QCB	none
135	Open Queue	DE= .UQCB	A = Err Code
136	Delete Queue	DE = .QCB	A = Err Code
137	Read Queue	DE = .UQCB	none
138	Conditional Read Queue	DE = .UQCB	A = Err Code
139	Write Queue	DE = .UQCB	none
140	Conditional Write Queue	DE = .UQCB	A = Err Code
141	Delay	DE = #ticks	none
142	Dispatch	none	none
143	Terminate Process	E = Term. Code	none
144	Create Process	DE = .PD	none
145	Set Priority	E = Priority	none
146	Attach Console	none	none
147	Detach Console	none	none
148	Set Console	E = Console	none
149	Assign Console	DE = .APB	A= Err Code
150	Send CLI Command	DE = .CLICMD	none
151	Call Resident Sys Proc	DE = .CPB	HL= result
152	Parse Filename	DE = .PFCB	see def
153	Get Console Number	none	A= console #
154	System DE~ta Address	none	HL = Sys Data Addr
155	Get Date and Time	DE = .TOD	none
156	Return PD Addr	none	HL = PD Addr
157	Abort Spec. Process	DE = .ABTPB	A = Err Code
158	Attach List	none	none
159	Detach List	none	none
160	Set List	E = List #	none
161	Cond. Attach List	none	A = Err Code
162	Cond. Attach Console	none	A = Err Code
163	NIPM Version Number	none	HL = Version#
164	Get List Number	none	A = list #

The following abbreviations are used in the table.

Addr = Address
Cond. = Conditional
Proc = Process
Rqst = Request
Spec. = Specified
term. = Terminate

Appendix E
Sample Page Relocatable Program

```
      *******************************************
      *  Note:                                  *
      *       This program listing has been     *
      *  included only as a sample and may not  *
      *  reflect changes required by later MP/M *
      *  releases. For this reason the reader   *
      *  should assemble and list the program   *
      *  as provided on the distribution disk.  *
      *******************************************
                  page  0
0000              org   0000h
0000        base  equ   $
0100              org   0100h

            ;note: either base0l00.asm or base0200.asm must be ap
            ;to the beginning of this file before assembling.

            ;     title 'file dump program'
            ;     file dump program, reads an input file and
            ;     prints in hex

            ;copyright (c) 1975, 1976, 1977,.1978, 1979, 19
            ;digital research
            ;box 579, pacific grove
            ;california, 93950

0005  =     bdos        equ   base+5      ;dos entry point
0001  =     cons        equ   1           ;read console
0002  =     typef equ   2           ;type function
0009  =     printf      equ   9           ;buffer print entry
000b  =     brkf        equ   11          ;break key function
000f  =     openf equ   15          ;file open
0014  =     readf equ   20          ;read function

005c =      fcb         equ   base+5ch    ;file control block address
0080  =     buff        equ   base+80h    ;input disk buffer address

            ;non graphic characters
000d  =     cr          equ   Odh         ;carriage return
000a  =     lf          equ   Oah         ;line feed

            ;file control block definitions
005c  =     fcbdn equ   fcb+O ;disk name
005d  =     fcbfn equ   fcb+l ;pfile name
0065  =     fcbft equ   fcb+9 ;disk file type (3 characters)
0068  =     fcbrl equ   fcb+12      ;file's current reel number
006b =      fcbrc equ   fcb+15      ;file's record count (0 to 128
007c =      fcbcr equ   fcb+32      ;current (next) record number
```

```
007d =       fcbln       equ   fcb+33      ;fcb length
             ;           set up stack
0100 210000              lxi   h,O
0103 39                  dad   sp
             ;           entry stack pointer in hl from the ccp
0104 221fO2              shld  oldsp
             ;           set sp to local stack area (restored at finis)
0107 316102              lxi   sp,stktop
             ;           read and print successive buffers
010a cdc601              call  setup ;set up input file
OlOd feff                cpi   255         ;255 if file not present
010f c21bOl              jnz   openok      ;skip if open is ok

             ;           file not there, give error message and return
0112 llfdOl              lxi   d,opnmsg
0115 cdalOl              call  err
0118 c35601              jmp   finis ;to return
             openok:     ;open operation ok, set buffer index to end
Ollb 3e8O                mvi   a,80h
Olld 321dO2              sta   ibp         ;set buffer pointer to 80h
             ;           hl contains next address to print
0120 210000              lxi   h,O         ;start with 0000

             gloop:
0123  e5                 push  h           ;save line position
0124  cda701             call  gnb
0127  el                 POP   h           ;recall line position
0128  da5601             jc    finis ;carry set by gnb if end file
012b  47                 mov   b,a
             ;           print hex values
             ;           check for line fold
012c 7d                  mov   a,l
012d e60f                ani   Ofh         ;check low 4 bits
012f c24401              jnz   nonum
             ;           print line number
0132 cd7701              call  crlf
             ;           check for break key
0135 cd5eOl              call  break
             ;           accum lsb = 1 if character ready
0138  Of                 rrc               ;into carry
0139  da51Ol             jc    purge ;don't print any more
013c  7c                 mov   a,h
013d  cd9401             call  phex
0140  7d                 mov   a,l
0141  cd9401             call  phex
             nonum:
0144 23                  inx   h           ;to next line number
0145  3e2O               mvi   a,’ ‘
0147  cd6aOl             call  pchar
014a  78                 mov   a,b
014b  cd9401             call  phex
```

```
014e  c32301            jmp   gloop

            purge:
0151 OeOl               mvi   c,cons
0153 cd05OO             call  bdos
finis:
            ;           end of dump, return to cap
            ;           (note that a jmp to 0000h reboots)
0156 cd7701             call  crlf
0159 2alfO2             lhld  oldsp
015C f9                 sphl
            ;           stack pointer contains cap's stack location
015d c9                 ret         ;to the ccp

suDroutines

            break:      ;check break key (actually any key will do)
015e e5d5c5             push h! push d! push b; environment saved
0161 OeOb               mvi   c,brkf
0163 cd05OO             call  bdos
0166 cldlel             pop b! pop d! pop h; environment restored
0169 c9                 ret
            pchar:      ;print a character
016a e5d5cS             push h! push d! push b; saved
016d OeO2               mvi   c,typef
016f 5f                 mov   e,a
0170 cd05OO             call  bdos
0173 cldlel             pop b! pop d! pop h; restored
0176 c9                 ret

crlf:
0177  3eOd              mvi   a,cr
0179  cd6aOl            call  pchar
017c  3eOa              mvi   a,lf
017e  cd6aOl            call  pchar
0181  c9                ret

            pnib: ;print nibble in reg a
0182 e60f               ani         Ofh   ;low 4 bits
0184 fe0a               cpi         10
0186 d28eOl             jnc         plo
            ;           less than or equal to 9
0189 c630               adi         '0'
018b c39001             jmp         prn
            ;           greater     or equal to 10
018e c637               pl0:  adi   'a' - 10
0190 cd6aOl prn:        call  pchar
0193 c9                 ret
```

```
            phex: ;print hex char in reg a
0194  f5                push  psw
0195  Of                rrc
0196  Of                rrc
0197  Of                rrc
0198  Of                rrc
0199  cd8201            call  pnib              ;print nibble
019C  fl                POP   psw
019d  cd8201            call  pnib
OlaO  c9                ret

            Err:        ;print error message
            ;           d,e addresses message ending with
Olal OeO9               mvi   c,printf    ;print buffer function
Ola3 cd05OO             call  bdos
Ola6 c9                 ret
            gnb:        ;get next byte
Ola7 3aldO2             lda   ibp
Olaa fe8O               cpi   80h
Olac c2b801             jnz   go
                        ;read another buffer
Olaf cdd30l             call  diskr
Olb2 b7                 ora   a           ;zero value if read ok
Olb3 cab.801            jz    go          ;for another byte
            ;           end of data, return with carry set for eof
Olb6 37                 stc
Olb7 c9                 ret
            go:         ;read the byte at buff+reg a
Olb8 5f                 mov   e,a         ;ls byte of buffer index
Olb9 1600               mvi   d,O         ;double precision index to de
Olbb 3c                 inr   a           ; i nd ex= i nd ex+l
Olbc 321dO2             sta   ibp         ;back to memory
            ;           pointer     is incremented
            ;           save the current file address
Olbf 218000             lxi   h,buff
O.lc2 19                dad   d
            ;           absolute character address is in hl
Olc3 7e                 mov   a,m
            ;           byte is     in the accumulator
Olc4 b7                 ora   a           ;reset carry bit
Olc5 c9                 ret

            setup:      ;set up file
            ;           open the file for input
Olc6  af                xra   a                 ;zero to accum
Olc7  327cOO            sta   fcbcr       ;clear current record
Olca  115cOO            Ixi   d,fcb
Olcd  OeOf              mvi   c,openf
Olcf  cd05OO            call  bdos
            ;           255 in accum if open error
```

```
Old2 c9                 ret

            diskr:      ;read disk file record
Old3 e5d5c5             push hl push d! push b
Old6 115cOO             lxi   d,fcb
Old9 Oe14               mvi   c,readf
Oldb cd05OO             call  bdos
Olde cldlel             pop b! pop d! pop h
Olel c9                 ret
            ;           fixed message area
            signon:
Ole2 46696c6520         db    'file dump mp/m version 1.0$'
            opnmsg:
Olfd OdOa4e6f2O         db    cr,lf,'no input file present on disk$'

            ;           variable area
021d        ibp:        ds    2     ;input buffer pointer
021f        oldsp:      ds    2     ;entry sp value from ccp
            ;           stack area
0221                    ds    64    reserve 32 level stack
            stktop:
0261                    end
```

Appendix F
Sample Resident System Process

```
******************************************
*  Note:                                  *
*  This program listing has been          *
*  included only as a sample and may not  *
*  reflect changes required by later MP/M *
*  releases. For this reason the reader   *
*  should assemble and list the program   *
*  as provided on the distribution disk.  *
******************************************
```

```
                        page  0

                        title 'type file on console'
                        ;file type program, reads an input file and pri
                        ;it on the console

                        ;copyright (c) 1979, 1980
                        ;digital research
                        ;p.o. box 579
                        ;pacific grove, ca 93950

0000                          org   0000h       ;standard rsp start

001a        =     ctlz        equ   lah               ; control-z used for e
0002        =     conout      equ   2                 ; bdos conout function
0009        =     printf      equ   9                 ; print buffer
0014        =     readf equ   20                ; read next record
000f        =     openf equ   15                ; open fcb
0098        =     parsefn     equ   152         ; parse file name
0086        =     mkque equ   134         ; make queue
0089        =     rdque equ   137         ; read queue
0091        =     stprior     equ   145         ; set priority
0093        =     detach      equ   147         ; detach console

                  ; bdos entry point address
                        bdosadr:
0000 0000                     dw    $-$   ldr will fill this i

                  ; type process descriptor

                        typepd:
0002 0000                     dw    0                 ;link
0004 00                       db    0                 ;status
0005 Oa                       db    10                ;priority (initial)
0006 1001                     dw    stack+38          ;stack pointer
0008 5459504520         db    'type    `  ;name in upper case
pdconsole:
```

```
0010                    ds    1                 ;console
0011 ff                 db    Offh              ;memseg
0012                    ds    2                 ;b
0014                    ds    2                 ;thread
0016 3601               dw    buff              ;disk set dma address
0018                    ds    1                 ;user code & disk sel
0019                    ds    2                 ;dcnt
001b                    ds    1                 ;searchl
001C                    ds    2                 ;searcha
001e                    ds    2                 ;active drives
0020                    ds    20                ;register save area
0034                    ds    2                 ;scratch

                  ; type linked queue control block

            typelqcb:
0036 0000               dw    0                 ;link
0038 5459504520         db    'type     `  ;name in upper case
0040 4800               dw    72                ;msglen
0042 0100               dw    I                 ;nmbmsgs
0044                    ds    2                 ;dqph
0046                    ds    2                 ;nqph
0048                    ds    2                 ;mh
004a                    ds    2                 ;mt
004c                    ds    2                 ;bh
004e                    ds    74                ;buf(72 + 2 byte lin

                  ; type user queue control block

            typeuserqcb:
0098 3600               dw    typelqcb          ; pointer
009a 9cOO               dw    field      ; msgadr

                        ;field for message read from type linked qcb

            field:
009C                    ds    1                 ;disk select
            console:
009d                    ds    1                 ;console
            filename:
009e                    ds    72                ;message body

                  ; parse file name control block

            pcb:
OOe6 9eOO               dw                      ;filename   file name address
OOe8 1201               dw    fcb               ;file control block a
;type stack & other local data structures

                  stack:
```

```
00ea                    ds    38                  ;20 level stack
0110        ba0:        dw    type                ;process entry point
0112        fcb:        ds    36                  ;file control block
0136        buff:       ds    128                 ;file buffer

                  ;bdos call prdeedure

                  bdos:
Olb6 2aOOOO             lhld  bdosadr             ;hl = bdos address
Olb9 e9                 pchl

            ;type main program

                  type:
Olba  Oe86              mvi   c,mkque
Olbc  113600            lxi   d,typelqcb
Olbf  cdb60l            call  bdos                ; make typelqcb
Olc2  Oe9l              mvi   c,stprior
Olc4  llc800            lxi   d,200
Olc7  cdb601            call  bdos                ; set priority to 200

            forever:
Olca  Oe89              mvi   c,rdque
Olcc  119800            lxi   d,typeuserqcb
Olcf  cdb60l            call  bdos                ; read from type queue
Old2  0698              mvi   c,parsefn
Old4  lle600            lxi   d,pcb
Old7  cdb601            call  bdos                ; parse the file name
Olda  23                inx   h
Oldb  7c                mov   a,h
Oldc  b5                ora   1                   ; test for Offffh
Oldd  calf02            jz    error
OleO  3a9dOO            Ida   console
Ole3  321000            sta   pdconsole           ; typepd.console = con
Ole6  OeOf              mvi   c,openf
Ole8  111201            Ixi   d,fcb
Oleb  cdb601            call  bdos                ; open file
Olee  3c                inr   a                   ;test return code
Olef  calf02            jz    error         ;if it was Offh, no f
Olf2  af                xra   a                   ;else,
Olf3  323201            sta   fcb+32              ;set next record to
            new$sector:
Olf6  Oe14              mvi   c,readf
Olf8  111201            lXi   d,fcb
Olfb  cdb60l            call  bdos                ;read next record
Olfe  b7                ora   a
Olff  c22702            jnz   done                ;exit if eof or error
0202  213601            Ixi   h,buff              ;point to data sector
```

```
0205  0e80               mvi   c,128       ;get byte count
            next$byte:
0207  7e                 mov   a,m               ;get the byte
0208  5f                 mov   e,a               ;save in e
0209  fe1a               cpi   ctlz
020b  ca2702             jz                      ;done exit if eof
020e  c5                 push  b                 ;save byte counter
020f  e5                 push  h                 ;save address registe
0210  0e02               mvi   c,conout
0212  cdb601             call  bdos              ;write console
0215  e1                 POP   h                 ;restore pointer
0216  c1                 POP   b                 ;and counter
0217  23                 inx   h                 ;bump pointer
0218  0d                 dcr   c                 ;dcr byte counter
0219 c20702              jnz   next$byte         ; more in this sector
021c c3f601              jmp   new$sector  ;else, we need a new

            error:
021f 112f02              lxi   d,err$msg         ;point to error messa
0222 0e09                mvi   c,printf          ; get function code to
0224 cdb601              call  bdos

                  done:
0227 0e93                mvi   c,detach
0229 cdb601              call  bdos              ;detach the console
022c c3ca01              jmp   forever

            err$msg:
022f 0d0a46696c          db    0dh,0ah,'file not found or bad file na

0251                     end
```

Appendix G
Acronyms and Conventions

Throughout this manual, the following conventions are used in describing the physical modules of MP/M II, its functional parts, and data structures:

```
I           PHYSICAL MODULES

      BDOS        - Basic Disk Operating System
      XDOS        - Extended Disk Operating System
      XIOS        - Extended I/O System

II          FUNCTIONAL PARTS

      CLI         - Command Line Interpreter
      TMP         - Terminal Message Processor

III         DATA STRUCTURES

      FCB         -     File Control Block
      MD          -     Memory Descriptor
      PD          -     Process Descripotr
      QCB         -     Queue Control Block
      UQCB        -     User Queue Control Block
      XFCB        -     Extended File Control Block

IV          NAMING CONVENTIONS

      Filenames:

      PRL         -     Page Relocatable File
      SPR         -     System Page Relocatable
      RSP         -     Resident System Process (or Procedure)
      BRS         -     Banked RSP

      System entry point  - First letter(s) are capitals and function
                            numbers are in parentheses
```

Appendix H
Glossary

BCD: See binary coded decimal.

binary coded decimal: Representation of decimal numbers using binary digits. See Appendix I for binary representations of ASCII codes.

block: Basic unit of disk space allocation under MP/M II. Each disk drive has a fixed block size defined in its Disk Parameter Block in the XIOS. The block size can be 1K, 2K, 4K, 8K or 16K consecutive bytes. Blocks are numbered relative to zero so that each block in a file is unique and has a byte displacement of the Block Number times the Block Size.

boolean: Variable that can only have two values; usually interpreted as true/false, or on/off.

Checksum Vector: Contiguous data area in the XIOS with one byte for each directory sector to be checked. A Checksum Vector is initialized and maintained for each logged-in drive. Each directory access by the system results in a checksum calculation which is compared with that in the Checksum Vector. If there is a discrepancy, the drive is set to read-only status. This prevents the user from inadvertently switching disks without logging-in the new disk. If not logged-in, the new disk is treated the same as the old one and data on it may be destroyed by writing to it.

COM: Filetype for MP/M II command files. These are machine language object modules ready to be loaded and executed. Any file with this type may be executed by simply typing the filename after the drive prompt (e.g.OA>) . For example, the program PIP.COM can be executed by simply typing PIP.

command: Set of instructions that are executed when the command name is typed after the system prompt. These instructions can be "built-in" to the MP/M II system or can reside on disk as a file of type COM, or PRL. In general, MP/M II commands consist of three parts: the command name, the command tail, and a carriage return
CSV: See checksum vector.

default buffer: 128-byte buffer maintained at 0080H in Page Zero. When the CLI loads a COM file,the CLI initializes this buffer to the command tail, i.e. any characters typed after the COM filename. The first byte at 0080H contains the length of the command tail while the command tail itself begins at 0081H. A binary zero terminates the command tail. The I command under DDT initializes this buffer in the same way as the CLI does.

default FCB: One of two FCBs maintained at 005CH and 006CH respectively, in Page Zero. The CLI initializes the first default FCB from the first delimited field in the command tail and initializes the second default FCB from the next field in the command tail.

delimiters: ASCII characters that separate constituent parts of a file specification. The CLI recognizes certain delimiter characters as :.=;<> , blank and carriage return. Several MP/M II commands also treat []()$ as delimiter characters. It is advisable to avoid the use of delimiter characters and lower-case characters in filenames.

directory: Portion of a disk containing entries for each file on the disk and locations of the blocks allocated to the files. Each file directory element is in the form of a 32-byte FCB, although one file can have several elements depending on its size. The maximum number of directory elements supported is specified in the drive's Disk Parameter Block.

directory element: 32-byte element associated with each disk file. A file can have more than one directory element associated with it. There are four directory elements per directory sector. Directory elements can also be refereed to as directory FCBs.

directory entry: File entry displayed when using the DIR command. This term also refers to a physical directory element (FCB).

Disk Parameter Block: Table residing in the XIOS that defines the characteristics of a drive in the disk subsystem used with MP/M II. The address of the DPB is in the Disk Parameter Header at DPbase + OAH. Drives with the same characteristics can use the same Disk Parameter Header, and thus the same DPB. However, drives with different characteristics must each have their own Disk Parameter Header and DPB. The address of the drive's Disk Parameter Header must be returned in registers HL when the BDOS calls the SELDSK entry point in the BIOS. BDOS Function 31 returns the DPB address.

Disk Parameter Header: 16-byte area in the XIOS that contains information about the disk drive and a scratchpad area for certain BDOS operations. Given n disk drives, the Disk Parameter Headers are arranged in a table with the first row of 16 bytes corresponding to drive 0, and the last row corresponding to drive n-l.

DPB: See Disk Parameter Block.

DPH: See Disk Parameter Header.
EX: See extent.

extent: 16K consecutive bytes in a file. Extents are numbered from 0 to 31. One extent can contain 1, 2, 4, 8 or 16 blocks. EX is the extent number field of an FCB and is a one byte field at FCB + 12, where FCB labels the first byte in the FCB. Depending on the Block Size and the maximum data Block Number, an FCB can contain 1, 2, 4, 8 or 16 extents. The EX field is normally set to 0 by the user but contains the current extent number during file I/O. The term 'FCB Folding' describes FCBs containing more than one extent. In CP/M version 1.4, each FCB contains only one extent. Users attempting to perform Random Record I/O and maintain CP/M 1.4 compatibility should be aware of the implications of this difference.

FCB: See File Control Block.

file: Collection of data containing from zero to 242,144 records. Each record contains 128 bytes and can contain either binary or ASCII data. ASCII data files consist of lines of data delineated by carriage return line feed sequences, meaning that one 128-byte record might contain one or more lines of text. Files consist of one or more extents, with 128 records per extent. Each file has one or more directory elements yet shows as only one directory entry when using the DIR command.

File Control Block: Thirty-six consecutive bytes designated by the user for file I/O functions. The FCB fields are explained in Section 2.2.3. The term FCB also refers to a directory element in the directory portion of the allocated disk space. These contain the same first 32 bytes of the FCB explained in Section 2.1 lacking only the Current Record and Random Record Number bytes.

HEX file format: Absolute output of ASM and MAC for the Intel 8080. A HEX file contains a sequence of absolute records which gives a load address and byte values to be stored starting at the load address. (see Section 4).

I/0: See Input/Output.

Input/Output: Operations or routines that handle the input and output of data in the computer system.

logical drive: Logically distinct region of a physical drive. A physical drive can be divided into one or more logical drives, and designated with specific drive references (i.e., d:a or d:f , etc.) . Thus at the user interface, it appears that there are several disks in the system.

Page Zero: Memory region between OOOOH and 0100H that holds critical system parameters and functions primarily as an interface region between user programs and the BDOS module.

parse: Separate a command line into its constituent parts.

physical drive: Peripheral hardware device used for mass storage of data within the computer system.

read-only: Condition in which a drive can be read but not written to. A drive can be set to read-only status by using the SET or STAT utilities or the SET File Attributes function (BDOS Function 30) . A drive can also be set to read-only status if the checksum computed on a directory access does not match that stored in CSV when the drive is logged-in. This protects the user from switching disks without executing a disk reset.

record: Smallest unit of data in a disk file that can be read or written. A record consists of 128 consecutive bytes whose byte displacement in a file is the product of the Record Number times 128. A 128-byte record in a file occupies one 128-byte sector on the disk. If the blocking and deblocking algorithm is used, several records can occupy each disk sector.

reentrant code: Code that one process can use while another is already executing it. The data for reentrant code is typically kept on the stack.

R/O: See read-only.

sector: Basic unit of data read and written on the disk by the XIOS. A sector can be one 128-byte record in a file or a sector of the directory. In some disk subsystems, the disk sector size is larger than 128 bytes, usually a power of two such as 256, 512, 1024 or 2048 bytes. These disk sectors are referred to as Host Sectors. When the Host Sector size is larger than 128 bytes, Host Sectors must be buffered in memory, and the 128-byte sectors must be blocked and deblocked from them.

spooling: Printing a file from disk. The SPOOL program, which is detached from a console, can print a file from a disk. This leaves your console free for other tasks while your file is being printed.

stack: Reserved area of memory where the processor saves the return address when it receives a Call instruction. When the processor encounters a Return instruction, it restores the current address on the stack to the Program Counter. Data such as the contents of the registers can also be saved on the stack. The Push instruction places data on the stack and the Pop instruction removes it. 8080 stacks are 16 bits wide; instructions operating on the stack add and remove stack items one word at a time. An item is pushed onto the stack by decrementing the stack pointer (SP) by two and writing the item at the SP address. In other words, the stack grows downward in memory.

track: Concentric ring on the disk; the standard IBM single density diskettes have 77 tracks. Each track consists of a fixed number of numbered sectors. Tracks are numbered from 0 to one less than the number of tracks on the disk. Data on the disk media is accessed by combinations of track and sector numbers.

user: Logically distinct subdivision of the directory. Each directory can be divided into 16 user areas.

vector: Memory location; an entry point into the operating system used for making system calls or interrupt handling.

wildcard character: Either ? or * characters. The BDOS directory search functions match ? with any single character and * with multiple characters.

Appendix I
ASCII and Hexadecimal Conversions

This appendix contains tables of the ASCII symbols, including their binary, decimal, and hexadecimal conversions.

Table I-1. ASCII Symbols

Symbol	Meaning	Symbol	Meaning
ACK	acknowledge	FS	file separator
BEL	bell	GS	group separator
BS	backspace	HT	horizontal tab
CAN	cancel	LF	line feed
CR	carriage return	NAK	negative acknowledge
DC	device control	NUL	null
DEL	delete	RS	record separator
DLE	data link escape	SI	shift in
EM	end of medium	SO	shift out
ENQ	enquiry	SOH	start of heading
EOT	end of transmission	SP	space
ESC	escape	STX	start of text
ETB	end of transmission	SUB	substitute
ETX	end of text	SYN	synchronous idle
FF	form feed	US	unit separator
		VT	vertical tabulation

Table 1-2. ASCII Conversion Table

Binary	Decimal	Hexadecimal	ASCII
0000000	000	00	NUL
0000001	001	01	SOH (CTRL-A)
0000010	002	02	STX (CTRL-B)
0000011	003	03	ETX (CTRL-C)
0000100	004	04	EOT (CTRL-D)
0000101	005	05	ENQ (CTRL-E)
0000110	006	06	ACK (CTRL-F)
0000111	007	07	BEL (CTRL-G)
0001000	008	08	BS (CTRL-H)
0001001	009	09	HT (CTRL-I)
0001010	010	OA	LF (CTRL-J)
0001011	Oil	OB	VT (CTRL-K)
0001100	012	0C	FF (CTRL-L)
0001101	013	OD	CR (CTRL-M)
0001110	014	OE	SO (CTRL-N)
0001111	015	OF	SI (CTRL-0)
0010000	016	10	DLE (CTRL-P)
0010001	017	11	DCI (CTRL-Q)
0010010	018	12	DC2 (CTRL-R)
0010011	019	13	DC3 (CTRL-S)
0010100	020	14	DC4 (CTRL-T)
0010101	021	15	NAK (CTRL-U)
0010110	022	16	SYN (CTRL-V)
0010111	023	17	ETB (CTRL-W)
0011000	024	18	CAN (CTRL-X)
0011001	025	19	EM (CTRL-Y)
0011010	026	IA	SUB (CTRL-Z)
0011011	027	1B	ESC (CTRL-[)
0011100	028	IC	FS (CTRL-\)
0011101	029	1D	GS (CTRL-])
0011110	030	1E	RS (CTRL-^)
0011111	031	IF	US (CTRL-_)
0100000	032	20	(SPACE)
oiO0001	033	21	!
0100010	034	22	"
0100011	035	23	#
0100100	036	24	$
0100101	037	25	%
0100110	038	26	&
0100111	039	27	'
0101000	040	28	(
0101001	041	29	)
0101010	042	2A	*
0101011	043	2B	+
0101100	044	2C	,
0i0iIOl	045	2D	-
0101110	046	2E	.
0101111	047	2F	/
0110000	048	30	0
0110001	049	31	1
0110010	050	32	2

Table 1-2. (continued)

Binary	Decimal	Hexadecimal	ASCII
0110011	051	33	3
0110100	052	34	4
0110101	053	35	5
0110110	054	36	6
0110111	055	37	7
0111000	056	38	8
0111001	057	39	9
0111010	058	3A	:
0111011	059	3B	;
0111100	060	3C	<
0111101	061	3D	=
0111110	062	3E	>
0111111	063	3F	?
1000000	064	40	@
1000001	065	41	A
1000010	066	42	B
1000011	067	43	C
1000100	068	44	D
1000101	069	45	E
1000110	070	46	F
1000111	071	47	G
1001000	072	48	H
1001001	073	49	1
1001010	074	4A	i
1001011	075	4B	K
1001100	076	4C	L
1001101	077	4D	M
1001110	078	4E	N
1001111	079	4F	0
1010000	080	50	P
1010001	081	51	Q
1010010	082	52	R
1010011	083	53	S
1010100	084	54	T
1010101	085	55	U
1010110	086	56	V
1010111	087	57	w
1011000	088	58	X
1011001	089	59	y
1011010	090	5A	z
1011011	091	5B	[
1011100	092	5C	
1011101	093	5D	]
1011110	094	5E	^
1011111	095	5F	<
1100000	096	60	`
1100001	097	61	a
1100010	098	62	b
1100011	099	63	c
1100100	100	64	d

Binary	Decimal	Hexadecimal	ASCII
1100101	101	65	e
1100110	102	66	f
1100111	103	67	g
1101000	104	68	h
1101001	105	69	i
1101010	106	6A	j
1101011	107	6B	k
1101100	108	6C	l
1i0iIOl	109	6D	m
1101110	110	6E	n
1101111	111	6F	o
1110000	112	70	p
1110001	113	71	q
1110010	114	72	r
1110011	115	73	s
1110100	116	74	t
1110101	117	75	u
1110110	118	76	v
1110111	119	77	w
1111000	120	78	x
1111001	121	79	y
1111010	122	7A	z
1111011	123	7B	{
1111100	124	7C	|
1111101	125	7D	}
1111110	126	7E	~
1111111	127	7F	DEL

Index

MP/M Operating System

Release 2.1

Release Notes

Digital Research
P.O. Box 579
160 Central Avenue
Pacific Grove, CA 93950
(408) 649-3896
TWX 910 360 5001

Dear MP/M II User:

Digital Research has developed the MP/M JIT.M. operating system in response to numerous customer requests to add file sharing capability to MP/M Release 1.1. The design of MP/M II is a reflection of our goal to provide you with a state of the art operating system that can be configured for a wide variety of computer hardware.

This shipment contains the version 2.1 release of our MP/M II operating system. We have been pleased with the response to MP/M II Release 2.0 and hope to see comparable response to MP/M II Release 2.1 regarding design, possible extensions, and errors in implementation. We hope to maintain the same level of confidence that the computer industry has had in our CP/M 0 operating system.

On the basis of our experience and the experience of MP/M II users, we estimate it requires less than a week to implement a simple polled MP/M II on a computer that has a running version of CP/M Release 2.2. Implementing a highly optimized MP/M II system with full interrupts and bank switched memory can require several weeks. Of course, the time to perform such a reconfiguration will vary widely depending on the experience of the programmer and the complexity of the hardware.

Note: Make sure you use the SET or STAT command to make the USER.PRL file into a system file.

Contact the Digital Research Technical Support staff (408) 375 6262 if you experience difficulties reconfiguring MP/M II. By sending in your registration card you can insure that we will mail MP/M II application notes and patches that correct implementation errors.

Sincerely,

TECHNICAL SUPPORT

MP/M II T.M. Operating System
Release 2.1
Extended File Locking

Addendum to the MP/M II Operating SZstem Programmer's Guide

Compiled January 1982

Extended file locking is a new facility implemented in release 2.1 of MP/M II T.M. . Extended file locking enables a process to maintain a lock on a file even after the file is closed. This facility allows a process to rename, set the attributes, or delete a file without having to contend with the possibility of interference from other processes after the file is closed. Also, a process can reopen a file with an extended lock and continue normal file processing. For example, a process can open a file, perform file operations on the file, close the file, rename the file, reopen the file under its new name, and proceed with file operations, without ever losing the file's lock list item and control over the file.

Extended file locking is only available to files that are opened in the default open mode (locked mode). To extend a file's lock, set interface attribute FV when closing the file. This attribute is only interrogated by the Close function when it is closing a file permanently. Thus, interface attribute F51 must be reset when the close call is made. Also, if a file has been opened N times (more than once), this attribute is only interrogated when the file is closed for the Nth time.

To maintain an extended file lock through a Rename File call or a Set File Attributes call, set interface attribute F5' of the referenced FCB when making the call. This attribute is only honored for extended file locks, not normal locks. Setting attribute F5' also maintains an extended file lock for the Delete File function, but setting this attribute also changes the nature of the Delete function to an XFCB-Only delete. If successful, all three of these functions delete a file's extended lock item when with attribute F51 reset. On the other hand, if they return with an error code, the extended lock item is not deleted.

A standard open call can be made to resume file operations on a file with an extended lock. The open mode, however, is restricted to the default locked mode. The following list illustrates uses of extended locks.

1

-Open file EXLOCK.TST in locked mode.

-Perform file operations on the file EXLOCK.TST using the open FCB.

-Close file EXLOCK.TST with interface attribute F6’ set to retain the file's lock item.

-Use the Rename File function to change the name of the file to EXLOCK.NEW with interface attribute F5’ set to retain the file's extended lock item.

-Open the file EXLOCK.NEW in locked mode.

-Perform file operations on the file EXLOCK.NEW using the opened FCB.

-Close file EXLOCK.NEW with interface attribute F6' set to retain the file's lock item.

-Set the Read-Only attribute and release the file's lock item by using the Set File Attributes function with interface attribute F51 reset. At this point, the file EXLOCK.NEW becomes available for access by another process.

MP/M II T.M. Operating System
Release 2.1
Compatibility Attributes

Addendum to the MP/M II Operating System Programmer's Guide

Compiled February 1982

The MP/M II T.M. file system introduced some new restrictions relating to file operations that were not present in MP/M T.M. 1. 1 or CP/M© . For example, if a process opens a file in the default mode (locked), MP/M II does not allow other processes on the system to open, delete, or rename the file until the process opening the file either closes the file or terminates. In addition, MP/M II does not allow a process to perform file operations with an FCB that has not been activated by a successful open or make operation, or with an FCB that has been deactivated by a close operation. These restrictions protect an MP/M II user from interference from other users on his open files. To illustrate, it is this protection that enables an MP/M II user to edit a file with the assurance that another user cannot delete or modify his file during his edit session.

The new restrictions added to MP/M II provide file security when multiple users are running the system. The preceding example describes restrictions required to prevent collisions on file activity between independent processes. Another new MP/M II restriction sets limits on how a process can modify open FCBs. These limits are enforced by checksum verification of open FCBs and protect the integrity of the MP/M II file system from corrupted FCBs. Note that the new MP/M II restrictions are not intended to protect a user from his own actions. Instead, they ensure that the activity of one user does not adversely affect other users on the system.

Generally, the new MP/M II file system restrictions create little difficulty for new application development. In fact, they enforce good programming practice. However, because of these new restrictions, some CP/M and MP/M software written before MP/M II's release does not run on MP/M II. In addition, multiple copies of some software do not run because the default open mode for MP/M II is a locked mode in which only one process can open a file.

To address these problems, Digital Research has added compatibility attributes to MP/M II, Release 2.1. The compatibility attributes are defined as attributes Fl' through F4’ of program files. A new GENSYS option determines whether the attributes are to be activated. If activated, the Command Line Interpreter (CLI) interrogates these attributes during program loading and modifies the MP/M II ground rules for the loaded program as described below.

Note that the compatibility attributes should not be used with new software. They are intended for use with working software developed for CP/M and MP/M 1.1. This especially applies to compatibility attribute F4' , which disables FCB checksum verification on read and write operations. Use this attribute sparingly and only with programs that are known to work.

COMPATIBILITY ATTRIBUTE DEFINITIONS

F1’ MP/M 1.1 Default Open. Processes running with this attribute have all files opened in locked mode marked as Read-Only in the System Lock List. This allows all processes with this attribute set to read and write to common files with no restrictions. There is, however, no record locking provided. In addition, this attribute also allows a process to write to a file opened by another process in Read-Only mode. To be safe, all static files such as program and help files should be made Read-Only when this compatibility attribute is used.

F2’ Partial Close default. Processes running with this attribute have their default close mode changed from permanent close to partial close. This attribute is for programs that close a file to update the directory but continue to use the file. Note that MP/M II assumes a process has finished with a file when the number of closes issued to the file equals the number of opens. A side effect of this attribute is that files opened by a process are not released until the process terminates. It might be necessary to set the System Lock List parameters to high values when using this attribute.

F3’ Ignore Close Checksum Errors. This attribute changes the way Close Checksum errors are handled for a process. Usually, a message is printed on the console, and the process is terminated. When this attribute is set and a checksum error is detected during a close operation, the file is closed if a lock list item exists for the file. Otherwise, an unsuccessful close error code is returned to the calling process.

F4’ Disable FCB Checksum verification for read and write operations. Setting this attribute also sets attributes F2' and F3'. This attribute should be used carefully because it effectively disables MP/M II's file security. Use this attribute only with software that is known to work.

PROCEDURE FOR USING THE COMPATIBILITY ATTRIBUTES

1) Answer yes to the GENSYS question "Enable Compatibility Attributes (N) ?".

2) Use the MP/M II Utility SET to set the desired combination of computability attributes in the program name.

EXAMPLES:

```
OA>SET filespec [Fl=on]
OA>SET filespec [Fl=on,F3=on]
OA>SET filespec [F4=on]
```

If you have a program that runs under CP/M or MP/M 1.1 but does not run properly under MP/M II, use the following guidelines to select the compatibility attributes to set for the program.

1) If the program ends with the message, "File Currently Opened" when multiple copies of the program are run, set compatibility attribute F1’. As an alternative, you might consider placing all common static files under User 0 with the SYS and R/O attributes set.

2) If the program terminates with the message, "Close Checksum Error", set compatibility attribute F3’.

3) If the program terminates with an I/O error, try running the program with attribute F2' set. If the problem still occurs, try attributes F2' and F3’. If the problem still persists, then try attribute F4’. Use attribute FV only as a last resort.

It might be necessary to increase the GENSYS parameters that set the maximum number of files a process can open and the size of the System Lock List when using compatibility attributes F2’ and F4’. This might be required because both default to partial closes. As a result, system lock list entries consumed by opening files are not released until the process ends. Generally, if a process ends with the message "No Room in System Lock List" or "Open File Limit Exceeded", it usually indicates that the above GENSYS parameters need to be increased. Another option is to patch in a BDOS Free Drive call at a point in the program where no files are active. Note that a Free Drive call specifying all drives, purges all file system lock entries belonging to the calling process.

When GENSYS activates compatibility attributes, the Command Line Interpreter copies the settings for attributes Fl' through F4' of the filename of the loaded program into byte lDH of the process descriptor as shown below:

```
PROCESS DESCRIPTOR BYTE IDH

(Bits defined 7-0 high-order to low-order)

Bit 7 set = Fl
Bit 6 set = F2
Bit 5 set = F3
Bit 4 set = F4
```

MP/M II T.M. Operating System
Release 2.1
Programming Guidelines

Addendum to the MP/M - II Operating System Programmer's Guide

Compiled January 1982

This guideline provides additional discussion on the information presented in the MP/M II TM Operating System Programmer's Guide. In particular, this (5-ocument emphasizes those areas of MP/M II where restrictions exist that did not exist in versions 1 and 2 of CP/M (C)and version 1 of MP/M T.M. The intent is to enable the MP/M II application programmer to avoid potential problems with new software. As a prerequisite, the reader should be familiar with the material presented in the MP/M II Operating System Programmer's Guide .

1) Always use the following sequence when performing file operations that require an open file. Under MP/M II, these operations are the BDOS read, write, lock, and unlock record commands.

   Activate a file's FCB with a BDOS Open or Make function call before using the FCB in a file operation. Verify that the Open or Make operation was successful. MP/M II only accepts FCBs activated by a successful Open or Make call for open file operations. If an FCB that has not been activated is used, MP/M II returns a checksum error.

   Perform all file operations using activated FCBs. Note that MP/M II does not deactivate an activated FCB when it returns error codes for file operations. Generally, only the current record and random record fields of an activated FCB should be modified. In addition, all file operations with an activated FCB must be made under the user number that was in effect when the FCB was activated. A similar restriction applies to activated FCBs that specify the default drive. All file operations specifying such an FCB must be made under the current drive that was in effect when the FCB was activated. Item 3 in this list covers the complete rules regarding activated FCB modification.

1

1

o If a process has completed file operations on a file but still has a significant amount of processing left to do, the file should be closed. This applies even if the file was not modified. With some exceptions, the lock list entry associated with a file in the system lock list is not released until a file is permanently closed (MP/M II Operating System Programmer's Guide see Section 2.2.9.) MP/M II restricts access to a file by other processes while a lock list item for the file resides in the system lock list. It is not necessary to close input files if a process is about to end. At termination, all lock items belonging to a process are released. Output files, however, must always be closed or data might be lost. Note that a successful permanent close operation deactivates the FCB and removes the file's item from the system lock list. If the deactivated FCB is used in a subsequent open file operation, MP/M II returns a checksum error.

2) If a process opens the same file more than once, a matching number of close commands must be issued to the file to remove the file's lock list item from the system lock list. Thus, if a file has been opened N times, the first N-1 close operations issued to the file default to partial close operations. Only the last close, close operation N, is interpreted as a permanent close. By definition, a permanent close is a close operation that removes the referenced file's item from the system lock list. Note that only one lock list item is allocated in the system lock list for a file regardless of the number of FCBs a process has opened for the file.

3) The following list specifies how an activated FCB can be changed without affecting the FCB checksum. MP/M II returns a checksum error code and does not perform the requested operation if an FCB with an invalid checksum is used in an open file operation.

o FCB(O) cannot specify a new drive.

o With the exception of interface attributes FV and F6' for the BDOS Close function, FCB(l) through FCB(11) cannot be changed.

o The high-order 3 bits of FCB(12) cannot be changed. The low order 5 bits can be changed. Note that when a file is opened in the default open mode (locked mode), the high-order 3 bits of this FCB field are set to zeros.

o FCB(13) cannot be changed.

o FCB(14) and FCB(15) can be changed.

o FCB(16) through FCB(31) cannot be changed.

O FCB(32) through FCB(35) can be changed.

2

If compatibilty with future releases of MP/M and CP/M is a requirement, programs should restrict open FCB modification to the FCB fields 32 through 35. In particular, Digital Research does not support techniques that involve modifying fields 12, 14, and 15 of open FCBs.

4) Processes that access a printer must issue a Detach List device to free the printer before another process can use the printer. If the Detach List call is not made, a process that accesses a printer continues to own the printer until it ends.

5) CP/M programs that create submit files for chaining must be modified to work under MP/M II. MP/M II requires a different filename for submit files, that includes the originating console number, and requires that a submit flag be set in the System Data Page. The technique for creating and executing submit files is described in MP/M II Application Note 07. Note that MP/M II also has a Program Chain (Function 47) command that provides an efficient mechanism for program chaining.

6) CP/M programs that make direct BIOS calls for disk I/O do not work under MP/M II. MP/M II does support direct XIOS calls for the console and printer, but not to the disk. If programs must make direct XIOS disk calls, a technique strongly discouraged in a multi-user environment, two levels of indirection must be used to obtain the real XIOS jump table address. The second level of indirection is required because an intercept table handles the console and printer.

The following two steps should be performed in a program before making direct XIOS calls to a disk. The f first step is to make a BDOS Write Protect Disk (Function 28) call to the disk to ensure that no other process has open files on the disk. Secondly, the MXDisk mutual exclusion queue message should be read to prevent other programs from making BDOS disk function calls while your program is making direct XIOS calls. After completing your direct XIOS calls, write back the MXDisk message and then reset the drives you have set to Read-Only.

7) The following procedure is a protocol that multiple processes can use to coordinate record update and addition operations to a shared file. Each process must open the shared file in unlocked mode. This procedure also assumes that records containing binary zeros are null records.

O Attempt to lock the record.

O If the lock attempt fails because another process has locked the record, delay and repeat the procedure.

o If the lock attempt fails because the record does not exist in the file, add a record initialized to binary zeros to the file with the BDOS Write Random with Zero Fill command and repeat the procedure. Note that files opened in unlocked mode are extended in block units and not in record units as is the case for files opened in the default locked mode.

O If the lock attempt succeeds, read the record, update it, and then unlock it.

8) Multiple FCB I/O is a technique that involves opening each extent for a file independently and maintaining them in a table in memory. Then random I/O is handled by selecting the proper FCB from the table, setting the current record field to the proper record number within the extent, and making a sequential Read or Write command. When processing is completed, each FCB is closed. The maximum file size that can be accessed with this technique is 512K bytes. This limits the maximum table size to 32 FCBs. Note that this technique provides a method of performing random I/O that is compatible with CP/M 1.4.

Multiple FCB I/O has to be performed carefully under MP/M II because of the restrictions MP/M II places on file operations to provide file security. Generally, an FCB should not be used in file I/O unless it has been activated and it should not be modified while it is activated (see items 1 and 3). In addition, the number of opens and closes issued to a file is important (see item 2). Note that all 32 bytes of each extent's FCB should be maintained in the open FCB table. Also, verify that interface attribute F8' is set to 1 in all FCBs if the first FCB has F8' set to 1. F8' set to 1 indicates the file was opened under user 0 although the current user number is nonzero (see Function 15 in the MP/M II Operating System Programmer's Guide).

MP/M II T.M. Operating System Release 2.0
Application Note 01, 9/14/81

Compiled September 1981

SUPPRESSING THE MP/M T.M.LOADER DISPLAY

Applicable products and version numbers: MP/M II Release 2.0

Program: MPMLDR.COM

When the MP/M II loader reads the MPM.SYS file, it displays a load map on console #0. In some applications you might want to suppress this display.

To suppress the load map display on console #0, type the following RET instruction into the LDRBIOS.ASM file using any standard editor. The RET instruction replaces the console output code.

```
;Loader BIOS jump vector:
      . . .
      jmp   conout
      . . .

conout:
      ret
```

Assemble LDRBIOS.ASM to create LDRBIOS.HEX. Integrate the new LDRBIOS.HEX file into the MPMLDR.COM file according to instructions provided in the MP/M II Operating System Guide. Then, update the system tracks of the boot disk with the new loader.

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

1

MP/M II T.M. Operating System Release 2.0
Application Note 02, 9/14/81

SETTING AND RESETTING THE RAW CONSOLE I/O MODE

Applicable products and version numbers: MP/M II". Release 2.0

Some application programs require raw input from the console. Raw input implies that the operating system takes no action on special characters, such as CTRL-C.

Execute the following code to place an application program into a raw console input mode.

```
        MVI         C,9CH
        CALL        XDOS  ; get process descriptor address
        LXI         D,6
        DAD         D
        MOV         A,M
        ORI         80H   ; turn 'on' the high-order bit of first
        MOV         M,A   ; character in the process name
```

Execute the following code to exit the raw console input mode.

```
        MVI         C,9CH
        CALL        XDOS  ; get process descriptor address
        LXI         D,6
        DAD         D
        MOV         A,M
        ANI         7FH   ; turn 'off' the high-order bit of first
        MOV         M,A   ; character in the process name
```

Functions 3, 4, and 6 place the system into raw console input mode. All other console I/O functions reset the system to normal console input mode.

Raw console input mode can cause problems. You cannot abort a process running in raw mode because the system ignores all control characters. To abort a process, use Function 11 before using any disk I/O functions. Function 11 returns the system to normal console mode.

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

MP/M II T.M. Operating System Release 2.0
Application Note 03, 9/14/81

CHANGING PRL FILE MINIMUM BUFFER SIZE REQUIREMENTS

Applicable products and version numbers: MP/M T.M. Release 2.0

You might want to allocate a larger default buffer for a program such as the editor. You can change the minimum buffer size requirements for PRL files. The following procedure demonstrates how to change the minimum buffer size requirements for ED from 4k to 8k bytes.

```
OA>ddt ed.prl
[MP/M] DDT VERS 1.1
NEXT PC
2300 0100
-s104
0104 00 00
0105 10 20
0106 .
-v2300
0044
-ied.prl
-w44
-go
```

Bytes 4 and 5 of the PRL header record (relative to the base) contain the low- and high-order bytes for the minimum buffer size specification.

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

MP/M II T.M. Operating System Release 2.0
Application Note 04, 9/14/81

ACCESSING THE INTERNAL MP/M TM TOD

Applicable products and version numbers: MP/M II Release 2.0

Some application programs might require access to the internal MP/M II time and date fields to set initial values. Execute the following code sequence at the end of your MP/MT.M. XIOS system initialization procedure. Place the code at the end because the XDOS call to obtain the system data page address might cause interruptions.

```
MVI   C,9AH
CALL XDOS         ;obtain the system data page address
                  ;*** warning ***
                  ;the XDOS call enables interrupts

LXI   D,OOFCH
DAD   D           ; hl  pointer -> TOD
MOV   E,M
INX   H
MOV   D,M         ; de  TOD
```

The assembly language subroutine TODCNV.ASM distributed on the MP/M II release disk converts from ASCII string representation of the time and date to MP/M II internal time and date format.

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

1

MP/M II T.M. Operating System Release 2.0
Application Note 05, 9/14/81

DMA DISK CONTROLLERS WITH BANKED MEMORY SYSTEMS

Applicable products and Version Numbers: MP/M II Release 2.0

Be extra careful with bank switched memory systems that have Direct Memory Access disk controllers. Bank switching is not allowed during a transfer of data from the disk controller to a target bank.

DMA from the disk controller is obtained through common memory then copied from common memory into the user buffer that you wanted. Sectors larger than 128 bytes are placed in a common memory buffer. The specified sector is then transferred to the target buffer. This is a reasonable technique in systems where deblocking is required

Use the following procedure if DMA is to occur directly into the user buffer bypassing common memory. Set a DMA active flag to true before each DMA operation. Reset the flag following each operation.

```
            MVI   A,FFH
            STA   DMACTVE

;           initiate DMA operation

;           perform flag wait or poll for operation complete

            XRA A
            STA DMACTVE
```

Place the following code sequence in the XIOS select memory procedure to ensure that the bank cannot be switched during a DMA operation:

```
      SELMEMORY:
            LDA   DMACTVE
            ORA   A
            jz    OKTOSWITCH  ;jump if not in DMA operation

      ; Next, the bank to be switched can be
      ; compared with the current bank. If
      ; it matches, the DMA operation will not be affected.

            jz    OKTOSWITCH  ;no bank change required
```

1

```
; A new bank is specified and a DMA operation is in
; progress. A busy wait must now be performed to wait
; until the DMA operation is complete.

*** warning ***

;The selmemory call is made from inside the dispatcher
;therefore interrupts are disabled and nothing must
;be done that could force a dispatch.

BUSYWAIT:
      IN    DMASTATUSPORT      ;This is a "BUSY-WAIT"
      ANI   DMADONE
      jz    BUSYWAIT           ;loop until the DMA is complete
```

Place the following code into the remaining select memory procedure.

```
OKTOSWITCH:

      . . .
      . . .

      RET
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

USING THE SEND CLI COMMAND XDOS FUNCTION

Applicable products and version numbers: MP/M II Release 2.0

Use of the Send CLI Command XDOS Function can effectively implement a menu driven application program. The following steps outline use of the SEND CLI XDOS Function.

1) Change the priority of the calling process so that it is higher (actually a lower value) than the TMP.

2) Obtain the console number of the calling process.

3) Assign the console to the Command Line Interpreter.

4) Issue the send CLI command function call.

5) Issue an ATTACH console function to get the console back after the initiated process has terminated.

6) Restore the priority of the calling process to its original value (usually 200).

Segments of a menu driven program named MENU appear in the following example.

```
;XDOS Function Equate Table

setpriority       equ   145
attachconsole     equ   146
assignconsole     equ   149
sendCLIcommand    equ   150
getconsole        equ   153

MENU:

                  mvi   e,190
                  mvi   c,setpriority
                  call  BDOS               ;set priority to 190
                  mvi   c,getconsole
                  call  BDOS               ;get console # in A reg
                  sta   AssignPB           ;fill in
                  sta   CLIcommand+l       ;console fields
                  lxi   d,AssignPB
                  mvi   c,assignconsole
```

```
                call  BDOS              ;assign console to CLI
                inr   a
                jz    cannotassign      ;assign failed
                lxi   d,CLIcommand
                mvi   c,sendCLIcommand
                call  BDOS              ;send CLI command
                mvi   c,attachconsole
                call  BDOS              ;attach console
                mvi   e,200
                mvi   c,setpriority
                call  BDOS              ;set priority back to 200

      AssignPB:
                db    $-$               ;console number
                db    Icli  I           ;name (cli is lower case)
                db    0

      CLIcommand:
                db    0           ;default disk / user code
                db    $-$         ;console number
                db                this is an ASCII string terminated with a
                                  ;null that is exactly as you would run the
                                  ;program from the console. e.g.

                                  'PIP LST:=MYPROG.LST[PT8]',O
                . . .
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

2

MP/M II T.M. Operating System Release 2.0
Application Note 07, 9/14/81

CREATING A SUBMIT FILE FROM AN APPLICATIONS PROGRAM

Applicable products and version numbers: MP/M II T.M. Release 2.0

The following procedure shows you how to create a submit file from an applications program and force its execution. The procedure to terminate a submit file job is included.

1) Obtain the temporary file drive from the system data page

2) Obtain the console number at which the program is executing.

3) Create the n.SUB file. Use n to specify the console number.

4) Set the appropriate submit flag in the array to on. The array is contained in the system data page.

```
;BDOS / XDOS Function Equate Table

closefile        equ   16
searchfirst      equ   17
deletefile       equ   19
makefile         equ   22
getconsole       equ   153
getsysdatadr     equ   154
subflgofst       equ   128

                 mvi          c,getsysdatadr
                 call         BDOS
                 lxi          d,196        ;temp file drive offset
                 dad          d
                 mov          a,m
                 sta          FCB
                 mvi          c,getconsole
                 call         BDOS
                 sta          console
                 adi          lot
                 sta          FCB+2        ;put console # in fname
                 lxi          d,FCB
                 mvi          c,searchfirst
                 call         BDOS         ;see if file there
```

1

```
                        inr   a
                        jz    nofile
                        lxi   d,FCB
                        mvi   c,deletefile
                        call  BDOS         ; delete old version first
                  nofile:
                        lxi   d,FCB
                        mvi   c,makefile
                        call  BDOS         ; make the $n$.SUB file
```

Now, write the records into the n.SUB file as
follows:

```
            -one line of the submit file per record

            -last record first (i.e. in reverse order
            that they are to be executed

            -each record in the following form:

            [CNTI[ASCII command line][NULL]
             where:     CNT = # chrs in cmnd ln, 1 byte
                        ASCII command line <= 125 chrs
                        NULL  zero, 1 byte

                        d,FCB
                        lx1
                        mvi   c,closefile
                        call  BDOS         ;close the $n$.SUB; file
                        mvi   c,getsysdatadr
                        call  BDOS         ;get system data page adr
                        lda   console      ;retrieve the saved con #
                        adi   subflgofst   ;add offset to base of
                        mov   e,a          ;submit flag array
                        mvi   d,O
                        dad   d            ;DE = submitflag(console);
                        mvi   m,Offh       ;set 'on' submit flag

; terminate the program

            FCB:
                        DB                 ;disk drive, usually A:
                        DB    '$n$'        ;filename
                        DB    'SUB'        ;filetype
                        DB    0            ;file extent
                        DS    20           ;remainder of FCB
      console:
                        ds    1            ;temp loc for console #
```

Terminate the operation of a submit job by zeroing a submit flag located in the SYSTEM DATA PAGE region of memory. To locate and zero the submit flag for a console use the following code procedure.

XDOS Function Equate Table

```
      getconsole        equ   153
      getsysdatadr      equ   154
      subflgofst        equ   128
                        mvi         c,getconsole
                        call        BDOS        ; get console #
                        push        psw         ; save console #
                        mvi         c,getsysdatadr
                        call        BDOS        ; get system data page adr
                        POP         psw         ; restore console #
                        adi         subflgofst
                        mov         l,a         ; hl = address of sub. flag
                        mvi         m,O         ; zero submit flag
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

MP/M II T.M. Operating System Release 2.0
Application Note 08, 9/14/81

FILE SHARING

Applicable products and version numbers: MP/M II T.M. Release 2.0

Multiple users can share files using the MP/M II file system. An applications program such as Wordstar requires that files be open while the program is running. Multiple users of the application will need to share the open files. Usually under MP/M II, sharing of files causes problems if the applications program is not structured to open files in Read-Only mode. The default mode for the open function is locked mode which prevents the sharing of files. Files are opened in locked mode for earlier versions of both CP/M 0 and MP/M T.M. as well.

To enable file sharing, place all files to be shared under USER 0 on the default disk. Using the SET utility, assign the attributes SYS (System) and RO (Read-Only) to the files. The BDOS opens the file in Read-Only mode regardless of which mode the open function specified. An example is shown below.

```
OA>set wsmsgs.com [SYS,R0]
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

1

MP/M II T.M. Operating System Release 2.0
Application Mote 09, 9/14/81

PROGRAM CONTROL OF THE CONTROL-P SWITCH

Applicable products and version numbers: MP/M II T.M. Release 2.0

An applications program might need to echo console I/0 to the printer while under program control. Use the following procedures to set and clear the CTRL-P flags. The array of flags is located at the SYSTEM DATA PAGE address + 126.

Setting CTRL-P Flag

```
            mvi         c,9ah       ; Get System Data Page address
            call        BDOS
            lxi         d,126
            dad         d           ; add 126 to Sys. Data Page addr.
            mov         e,m
            inx         h
            mov         d,m         ; DE  addr. of CTRL-P array
            push        d
            mvi         c,Oa4h      ; Get List Number
            call        BDOS
            mov         e,a
            mvi         d,O
            POP         h
            dad         d
            mvi         m,Offh      ; set CTRL-P flag
                                    ; cons. I/O is echoed from now on
```

Clearing CTRL-P Flag

```
            mvi         c,9ah       ; Get System Data Page address
            call        BDOS
            lxi         d,126
            dad         d           ; add 126 to Sys. Data Page addr.
            mov         e,m
            inx         h
            mov         d,m         ; DE  addr. of CTRL-P array
            push        d
            mvi         c,Oa4h      ; Get List Number
            call        BDOS
            mov         e,a
            mvi         d,O
            POP         h
            dad         d
            mvi         m,0         ; reset CTRL-P flag
                                    ; console I/O echo is now off
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

1

MP/M II T.M. Operating System Release 2.0
Application Mote 10, 9/14/81

COLD BOOT STARTUP

Applicable products and version numbers: MP/M II T.M. Release 2.0

MP/M Ii can execute one command upon cold boot. However, the system can execute any number of commands upon cold boot if the initial command is SUBMIT.

To execute the Startup command place the Startup command singularly into a file using standard command format. Name this file n.SUP where n is the console number that executes the command. The n.SUP file resides on the system drive at the desired USER number or at USER 0 with a SYS (SYSTEM) attribute. Examples are shown below.

```
Startup file:     $0$.SUP

Command in the
Startup file:     SUBMIT START$O$
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

MP/M II T.M. Operating System Release 2.0
Application Note 11, 9/14/81

SUBMIT ENHANCEMENTS

Applicable products and version numbers: MP/M II T.M. Release 2.0

Enhancements to SUBMIT include the following new features and facilities.

INCREASED n.SUB FILE SIZE: SUBMIT file size is now unlimited. The n.SUB file originally was limited to one extent, 128 lines.

CHANGING THE USER NUMBER: To change the current USER number in SUBMIT, include the USER command in the SUBMIT file.

INCLUDE FILES: An include file is a standard SUBMIT file subject to all SUBMIT rules and features. Format for the INCLUDE command is demonstrated below.

```
$INCLUDE filename parml parm2 parm3 ...
```

The filename in the INCLUDE command must have the filetype SUB to indicate a SUBMIT file and parameters are standard SUBMIT parameters. An INCLUDE file can nest up to four SUBMITs in a SUBMIT command.

EMBEDDED CONTROL CHARACTERS: Control characters can be embedded in a SUBMIT file by preceding the capitalized character with an ASCII up arrow ^. For example, type ~X to embed a CTRL-X. Embedded control characters are not interpreted by MP/M II, but can be of use to programs that SUBMIT executes.

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

I

SPOOL UTILITY MODIFICATIONS

Applicable products and version numbers: MP/M II Release 2.0

SPOOL can return an error message if the file to be spooled is not found. To utilize this modification, SPOOL is divided into a transient portion (SPOOL.PRL) and a resident portion. The transient portion parses the command tail, opens the file, passes the file to the spool queue (named SPOOLQ) , and displays an error message if the open sequence on the file fails. Then, the transient portion ends itself.

Issue a SPOOLQ command if you do not want to use a memory segment to spool a file. Error messages are not returned, however. Sample commands to spool a file are shown below.

```
SPOOL filel.typ,file2.typ .,.
```

The SPOOL process passes the command tail, checks for errors, and sends the file to the spool queue (SPOOLQ).

```
SPOOLQ filel.typ,file2.typ ...
```

The command tail is sent to the spool queue (SPOOLQ) bypassing error checking or error reporting.

The SPOOL utility sets its priority to 201. Most processes execute ahead of the SPOOLER. To change the SPOOLER priority, the SPOOL.BRS file is modified. Make sure you have a back-up copy of SPOOL.BRS before using RDT to make the following changes.

```
A>rdt spool.brs

-s3b5
03B5 C9 c8
03B6 00 .
-ispool.brs
-w14
-go
A>gensys
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

MP/M II T.M. Operating System Release 2.0
Application Note 13, 9/14/81

RECORD LOCKING/UNLOCKING

Applicable products and version numbers: MP/M II Release 2.0

Record locking/unlocking allows multiple processes to share access of one file. Files are opened in the UNLOCKED mode. A record locked by one process can only be read by a different process, however, a locked record can be modified by the initial process. Avoid reading locked records to prevent reading data that is being updated. To avoid reading locked records let the process try to lock the record. If the attempt fails, do not read the record. The following code segment demonstrates how to lock records.

```
mvi     c,2ch ;set multi-sector cnt.
mvi     e,#   ;# = num. of sectors
call    bdos  ;1<= # <=16
mvi     c,2ah ;lock record
lxi     d,fcb ;record to be locked
call    bdos
```

The following code segment demonstrates how to unlock records.

```
mvi     c,2ch ;set multi-sector cnt.
mvi     e,#   ;# = num. of sectors
call    bdos  ;1<= # <=16
mvi     c,2bh ;unlock record
lxi     d,fcb ;record to be unlocked
call    bdos

fcb:
    db      O,'DATA',AOH,20H,20H,20H,'DAT',O
    ds      20
    db      10,0,0      ;beginning at record 10
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

1

MP/M II T.M. Operating System Release 2.0
Application Note 14, 9/14/81

GENSYS ENHANCEKKNTS

Applicable products and version numbers: MP/M II T.M. Release 2.0

Enhancements to GENSYS include the following new features and facilities.

AUTOMATIC RESIDENT SYSTEM PROCESS INCLUSION FACILITY: The GENSYS automatic system generation facility can be modified to include all default disk RSP files. Type GENSYS $AR to include the RSP files automatically. The R option must be used in conjunction with the A option. Change the filetypes for files that you want to exclude from GENSYS.

ERROR RECOVERY: If an error is encountered running in automatic mode ($A option), GENSYS restarts in manual mode.

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

1

MP/M II T.M. Operating System Release 2.0
Application Note 15, 12/1/81

CHANGING THE PRIORITY OF SPOOL.PRL

Applicable products and version numbers: MP/M II Release 2.0

The SPOOL utility sets its priority to 201, therefore, most other processes execute ahead of the SPOOLER. Modify the SPOOL.PRL file to change the SPOOLER priority. If your product serial number is between 4-000-00001 and 4-000-00464, install MP/M II Patch 11 before changing the SPOOL.PRL default priority.

Make sure you have a back-up copy of SPOOL.PRL before using DDT to make the following changes.

```
A>ren spool.sav=spool.prl
A>ddt spool.sav
[MP/M II] DDT VERS 2.0
NEXT PC
0980 0100
-s269
0269 C9 c8
026A CD .
-ispool.prl
-W11
-go

A>
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.

1

MP/M II T.M. Operating System Release 2.0
Application Note 16, 12/1/81

CHANGING THE BACKSPACE AND RUBOUT KEY FUNCTIONS

Applicable products and version numbers: Mp/M II T.M. Release 2.0

Program: RESBDOS

Under MP/M II, the BACKSPACE key or CTRL-H (ASCII 08H) does a destructive backspace deleting the last character in the command buffer. The RUBOUT key (ASCII 7FH) or DELETE key deletes the last character in the command buffer and echoes it to the screen.

Procedure to reverse the BACKSPACE and RUBOUT key functions:

```
A>ren resbdos.sav=resbdos.spr
A>ddt resbdos.sav
[MP/M II] DDT VERS 2.0
NEXT PC
Of8O 0100
-scOa
OCOA 08 7f
OCOB C2
-sc22
OC22 7F 08
OC23 C2 .
-iresbdos.spr
-wld
-go

OA>gensys
```

Procedure to make RUBOUT identical to BACKSPACE:

```
A>ren resbdos.sav=resbdos.spr
A>ddt resbdos.sav
[MP/M II] DDT VERS 2.0
NEXT PC
Of8O 0100
-lc26
OC26 MOV A,B
OC27 ORA A
OC28 Jz 09F6
-ac26
OC26 jmp aOe
OC29
-bb28,1
-iresbdos.spr
-wld
-go

OA>gensys
```

Procedure to make BACKSPACE identical to RUBOUT:

```
A>ren resbdos.sav=resbdos.spr
A>ddt resbdos.sav
[MP/M II] DDT VERS 2.0
NEXT PC
Of8O 0100
-IcOe
OCOE MOV A,B
OCOF ORA A
oclo iz 09F6
-acOe
OCOE jmp, a26
ocil
-bblO,l
-iresbdos.spr
-wld
-go

OA>gensys
```

Licensed users are granted the right to include these enhancements in MP/M II Release 2.0 software.