MP/M™

MULTI-PROGRAMMING MONITOR CONTROL PROGRAM

USER'S GUIDE

DIGITAL RESEARCH™

MP/M™ MULTI-PROGRAMMING MONITOR CONTROL PROGRAM USER'S GUIDE

The "MP/M User's Guide" was prepared using the Digital Research TEX-80™ Text Formatter.

****
* Third Printing: March 1981 *
****

Appendix

1. MP/M FEATURES AND FACILITIES

1.1 Introduction

The purpose of the MP/M multi-programming monitor control program is to provide a microcomputer operating system which supports multi-terminal access with multi-programming at each terminal.

OVERVIEW

The MP/M operating system is an upward compatible version of CP/M 2.Ø with a number of added facilities. These added facilities are contained in new logical sections of MP/M called the extended I/O system and the extended disk operating system. In this manual the name XIOS will refer to the combined basic and extended I/O system. BDOS will refer to the standard CP/M 2.Ø basic disk operating system functions and XDOS will refer to the extended disk operating system. As an upward compatible version, users can easily make the transition from CP/M to the MP/M operating system. In fact, existing CP/M *.COM files can be run under MP/M, providing that the program has been correctly written. That is, BDOS calls are made for I/O, and the only direct BIOS calls made are for console and printer I/O. There must also be at least 4 bytes of extra stack in the CP/M *.COM program.

The following basic facilities are provided:

- Multi-terminal support
- Multi-Programming at each terminal
- Support for bank switched memory and memory protection
- Concurrency of I/O and CPU operations
- Inter-process communication, mutual exclusion and synchronization
- Ability to operate in sequential, polled or interrupt driven environments
- System timing functions
- Logical interrupt system utilizing flags
- Selection of system options at system generation time
- Dynamic system configuration at load time

The following optional facilities are provided:

- Spooling list files to the printer
- Scheduling programs to be run by date and time
- Displaying complete system run-time status
- Setting and reading of the date and time

1.2 Functional Description of MP/M

The MP/M Operating System is based on a real-time multi-tasking nucleus. This nucleus provides process dispatching, queue management, flag management, memory management and system timing functions.

MP/M is a priority driven system. This means that the highest priority ready process is given the CPU resource. The operation of determining the highest priority ready process and then giving it the CPU is called dispatching. Each process in the system has a process descriptor. The purpose of the process descriptor is to provide a data structure which contains all the information the system needs to know about a process. This information is used during dispatching to save the state of the currently running process, to determine which process is to be run, and then to restore that processes state. Process dispatching is performed at each system call, at each interrupt, and at each tick of the system clock. Processes with the same priority are "round-robin" scheduled. That is, they are given equal slices of CPU time.

Queues perform several critical functions in a real-time multi-tasking environment. They can be used for the communication of messages between processes, to synchronize processes, and for mutual exclusion. As the name "queue" implies, they provide a first in first out list of messages, and as implemented in MP/M, a list of processes waiting for messages.

The flag management provided by MP/M is used to synchronize processes by signaling a significant event. Flags provide a logical interrupt system for MP/M which is independent of the physical interrupt system. Flags are used to signal interrupts, mapping an arbitrary physical interrupt environment into a regular structure.

MP/M manages memory in pre-defined memory segments. Up to eight memory segments of 48K can be managed by MP/M. This management of memory is consistent with hardware environments where memory is banked and/or protected in fixed segments.

System timing functions provide time of day, the capability to schedule programs to be loaded from disk and executed, and the ability to delay the execution of a process for a specified period of time.

RUNNING A PROGRAM

A program is run by typing in the program name followed by a carriage return, <cr>. Some programs obtain parameters on the same line following the program name. Characters on the line following the program name constitute what is called the command tail. The command tail is copied into location 0080H (relative to the base of the memory segment in which the program resides) and converted to upper case by the Command Line Interpreter (CLI). The CLI also parses the command tail producing two file control blocks at 005CH and 006CH respectively.

The programs which are provided with MP/M are described in sections 1.4 and 1.5.

ABORTING A PROGRAM

A program may be aborted by typing a control C (^C) at the console. The affect of the ^C is to terminate the program which currently owns the console. Thus, a detached program cannot be aborted with a C. A detached program must first be attached and then aborted. A running program may also be aborted using the ABORT command (see ABORT in section 1.5).

RUNNING A RESIDENT SYSTEM PROCESS

At the operator interface there is no difference between running a program from disk and running a resident system process. The actual difference is that resident system processes do not need to be loaded from disk because they are loaded by the MP/M loader when a system cold start is performed and remain resident.

DETACHING FROM A PROGRAM

There are two methods for detaching from a running program. The first is to type a control D (^D) at the console. The second method is for a program to make an XDOS detach call.

The restriction on the former method, typing D, is that the running program must be performing a check console status to observe the detach request. A check console status is automatically performed each time a user program makes a BDOS disk function call.

ATTACHING TO A DETACHED PROGRAM

A program which is detached from a console, that is it does not own a console, may be attached to a console by typing 'ATTACH' followed by the program name. A program may only be attached to the console from which it was detached. If the terminal message process (TMP) has ownership of the console and

not obtain the printer mutual exclusion message prior to accessing the printer. If the list device is not available a 'Printer busy' message is displayed on the console.

ctl-S Stop the console output temporarily. Program execution and output continue when the next character is typed at the console (e.g., another ctl-S). This feature is used to stop output on high speed consoles, such as CRT's, in order to view a segment of output before continuing.

```
1A>DSKRESET  B:,E:
```

If there are any open files on the drive(s) to be reset, the disk reset is denied and the cause of the disk reset failure is shown:

```
1A>DSKRESET B:

Disk reset denied, Drive B: Console Ø Program Ed
```

The reason that disk reset is treated so carefully is that files left open (e.g. in the process of being written) will lose their updated information if they are not closed prior to a disk reset.

ERASE FILE *

The ERA (erase) command removes specified files having the current user code. If no files can be found on the selected diskette which satisfy the erase request, then the message "No file" is displayed at the console.

An attempt to erase all files,

```
2B>ERA *.*
```

will produce the following response from ERA:

```
Confirm delete all user files (Y/N)?
```

A second form of the erase command (ERAQ) enables the operator to selectively delete files that match the specified filename reference. For example:

```
ØA>ERAQ *.LST

A:XIOS    LST? y
A:MYFILE  LST? n
```

TYPE A FILE *

The TYPE command displays the contents of the specified ASCII source file on the console device. The TYPE command expands tabs (ctl-I characters), assuming tab positions are set at every eighth column.

The TYPE command has a pause mode which is specified by entering a 'P' followed by two decimal digits after the filename. For example:

```
ØA>TYPE DUMP.ASM P23
```

STATUS *

The STAT (status) command provides general statistical information about the file storage. See the Digital Research document titled "CP/M 2.Ø User's Guide for CP/M 1.4 Owners" for a detailed description of new STAT operations.

DUMP *

The DUMP command types the contents of the specified disk file on the console in hexadecimal form.

LOAD *

The LOAD command reads the specified disk file of type HEX and produces a memory image file of type COM which can subsequently be executed.

GENMOD

The GENMOD command accepts a file which contains two concatenated files of type HEX which are offset from each other by Ø1ØØH bytes, and produces a file of type PRL (page relocatable). The form of the GENMOD command is as follows:

```
1A>genmod b:file.hex b:file.prl $1ØØØ
```

The first parameter is the file which contains two concatenated files of type HEX. The second parameter is the name of the destination file of type PRL. The optional third parameter is a specification of additional memory required by the program beyond the explicit code space. The form of the third parameter is a '$' followed by four hex ASCII digits. For example, if the program has been written to use all of 'available' memory for buffers, specification of the third parameter will ensure a minimum buffer allocation.

GENHEX

The GENHEX command is used to produce a file of type HEX from a file of type COM. This is useful to be able to generate HEX files for GENMOD input. The GENHEX command has two parameters, the first is the COM file name and the second is the offset for the HEX file. For example:

```
ØA>GENHEX PROG.COM 1ØØ
```

PRLCOM

The PRLCOM command accepts a file of PRL type and produces a file of COM type. If the destination COM file exists, a query is made to determine if the file should be deleted before continuing.

1.5 Standard Resident System Processes

The standard resident system processes (RSPs) are new programs specifically designed to facilitate use of the MP/M operating system. The RSPs may either be present on disk as files of the PRL type, or they may be resident system processes. Resident system processes are selected at the time of system generation.

SYSTEM STATUS

The MPMSTAT command allows the user to display the run-time status of the MP/M operating system. MPMSTAT is invoked by typing 'MPMSTAT' followed by a <cr>. A sample MPMSTAT output is shown below:

```
****** MP/M Status Display ******

Top of memory = FFFFH
Number of consoles = Ø2
Debugger breakpoint restart # = Ø6
Stack is swapped on BDOS calls
Z8Ø complementary registers managed by dispatcher
Ready Process(es):
  MPMSTAT  Idle
Process(es) DQing:
 [Sched   ] Sched
 [ATTACH  ] ATTACH
 [CliQ    ] cli
Process(es) NQing:
Delayed Process(es):
Polling Process(es):
  PIP
Process(es) Flag Waiting:
  Ø1 -  Tick
  Ø2 -  Clock
Flag(s) Set:
  Ø3
Queue(s):
   MPMSTAT   Sched     CliQ      ATTACH    MXParse
   MXList   [TmpØ    ] MXDisk
Process(es) Attached to Consoles:
  [Ø] - MPMSTAT
  [1] - PIP
Process(es) Waiting for Consoles:
  [Ø] - TMPØ     DIR
  [1] - TMP1
Memory Allocation:
  Base = ØØØØH  Size = 4ØØØH  Allocated to PIP      [1]
  Base = 4ØØØH  Size = 2ØØØH  * Free *
  Base = 6ØØØH  Size = 11ØØH  Allocated to DIR      [Ø]
```

have detached from the console and are then waiting for the console before they can continue execution.

Memory Allocation: The memory allocation map shows the base, size, bank, and allocation of each memory segment. Segments which are not allocated are shown as '* Free *', while allocated segments are identified by process name and the console in brackets associated with the process. Memory segments which are set as pre-allocated during system generation by specifying an attribute of ØFFH are shown as '* Reserved *'.

SPOOLER

The SPOOL command allows the user to spool ASCII text files to the list device. Multiple file names may be specified in the command tail. The spooler expands tabs (ctl-I characters), assuming tab positions are set at every eighth column.

The spooler queue can be purged at any time by using the STOPSPLR command.

An example of the SPOOL command is shown below:

```
1A>SPOOL LOAD.LST,LETTER.PRN
```

The non-resident version of the spooler (SPOOL.PRL) differs in its operation from the SPOOL.RSP as follows: it uses all of the memory available in the memory segment in which it is running for buffer space; it displays a message indicating its status and then detaches from the console; it may be aborted from a console other than the initiator only by specifying the console number of the initiator as a parameter of the STOPSPLR command.

```
3B>STOPSPLR 2
```

DATE AND TIME

The TOD (time of day) command allows the user to read and set the date and time. Entering 'TOD' followed by a <cr> will cause the current date and time to be displayed on the console. Entering 'TOD' followed by a date and time will set the date and time when a <cr> is entered following the prompt to strike a key. Each of these TOD commands is illustrated below:

```
1A>TOD <cr>

Wed Ø2/Ø6/8Ø Ø9:15:37

-or-

1A>TOD 2/9/8Ø 1Ø:3Ø:ØØ
```

2. MP/M INTERFACE GUIDE

This section describes MP/M system organization including the structure of memory and system call functions. The intention is to provide the necessary information required to write page relocatable programs and resident system processes which operate under MP/M, and which use the real-time, multi-tasking, peripheral, and disk I/O facilities of the system.

2.1 Introduction

MP/M is logically divided into several modules. The three primary modules are named the Basic and Extended I/O System (XIOS), the Basic Disk Operating System (BDOS), and the Extended Disk Operating System (XDOS). The XIOS is a hardware-dependent module which defines the exact low level interface to a particular computer system which is necessary for peripheral device I/O. Although a standard XIOS is supplied by Digital Research, explicit instructions are provided for field reconfiguration of the XIOS to match nearly any hardware environment.

MP/M memory structure is shown below:

The memory segments are described as follows:

SYSTEM.DAT The SYSTEM.DAT segment contains 256 bytes used by the loader to dynamically configure the system. After loading, the segment is used for storage of system data such as submit flags. See section 3.4 under SYSTEM DATA for a detailed description of the byte allocation.

CONSOLE.DAT The CONSOLE.DAT segment varies in length with the number of consoles. Each console requires 256 bytes which contains the TMP's process descriptor, stack and buffers.

USERSYS.STK The USERSYS.STK segment is optional depending upon whether or not the user intends to run CP/M *.COM files. This segment contains 64 bytes of stack space per user memory segment and is used as a temporary stack when user programs make BDOS calls. Specification of the option to include this segment is made during system generation. The size of the USERSYS.STK segment varies as follows:

ØØØH - No user system stacks
1ØØH - 1 to 4 memory segments
2ØØH - 5 to 8 memory segments

XIOS The XIOS segment contains the user customized basic and extended I/O system in page relocatable format.

BDOS/ODOS The BDOS segment contains the disk file and multiple console management functions. The segment is about 14ØØH bytes in length.

The ODOS segment contains the resident portion of the banked BDOS file and console management functions. The segment is about 8ØØH bytes in length.

XDOS The XDOS segment contains the MP/M nucleus and the extended disk operating system. The segment is about 2ØØØH bytes in length.

RSPs The operator makes a selection of Resident System Processes during system generation. The RSPs require varying amounts of memory.

BNKBDOS (Optional) The BNKBDOS segment is present only in systems with a bank switched BDOS. It contains the non-resident portion of the banked BDOS disk file management. This segment is about EØØH bytes in length.

segment, relocated, and it is executed, completing the CLI operation.

If the PRL file type open fails then the file type of COM is entered for the parsed file name and a file open is attempted. If the open succeeds then a memory request is made for an absolute TPA, memory segment based at 0000H. If this request is satisfied the COM file is read into the absolute TPA and it is executed, completing the CLI operation.

If the command is followed by one or two file specifications, the CLI prepares one or two file control block (FCB) names in the system parameter area. These optional FCB's are in the form necessary to access files through MP/M BDOS calls, and are described in the next section.

The CLI creates a process descriptor for each program which is loaded, setting up a 20 level stack which forces a branch to the base of the user code area of the memory segment. The default stack is set up so that a return from the loaded program causes a branch to the MP/M facility which terminates the process. This stack has 19 levels available which can generally be used by the transient program since it is sufficiently large to handle system calls.

The transient program then begins execution, perhaps using the I/O facilities of MP/M to communicate with the operator's console and peripheral devices, including the disk subsystem. The I/O system is accessed by passing a "function number" and an "information address" to MP/M through the entry point at the memory segment base +0005H. In the case of a disk read, for example, the transient program sends the number corresponding to a disk read, along with the address of an FCB to MP/M. MP/M, in turn, performs the operation and returns with either a disk read completion indication or an error number indicating that the disk read was unsuccessful. The function numbers and error indicators are given in sections 2.2 and 2.4.

OPERATING SYSTEM CALL CONVENTIONS

The purpose of this section is to provide detailed information for performing direct operating system calls from user programs. Many of the functions listed below, however, are more simply accessed through the I/O macro library provided with the MAC macro assembler, and listed in the Digital Research manual entitled "MAC Macro Assembler: Language Manual and Applications Guide."

MP/M facilities which are available for access by transient programs fall into two general categories: simple device I/O, disk file I/O, and the XDOS functions.

As mentioned above, access to the MP/M functions is accomplished by passing a function number and information address through the primary entry point at location memory segment base +ØØØ5H. In general, the function number is passed in register C with the information address in the double byte pair DE. Single byte values are returned in register A, with double byte values returned in HL (a zero value is returned when the function number is out of range). For reasons of compatibility, register A = L and register B = H upon return in all cases. Note that the register passing conventions of MP/M agree with those of Intel's PL/M systems programming language.

The list of MP/M BDOS function numbers is given below.

Ø	System Reset	21	Write Sequential
1	Console Input	22	Make File
2	Console Output	23	Rename File
3	Raw Console Input	24	Return Login Vector
4	Raw Console Output	25	Return Current Disk
5	List Output	26	Set DMA Address
6	Direct Console I/O	27	Get Addr(Alloc)
7	Get I/O Byte	28	Write Protect Disk
8	Set I/O Byte	29	Get R/O Vector
9	Print String	3Ø	Set File Attributes
1Ø	Read Console Buffer	31	Get Addr(Disk Parms)
11	Get Console Status	32	Set/Get User Code
12	Return Version Number	33	Read Random
13	Reset Disk System	34	Write Random
14	Select Disk	35	Compute File Size
15	Open File	36	Set Random Record
16	Close File	35	Compute File Size
17	Search for First	36	Set Random Record
18	Search for Next	37	Reset Drive
19	Delete File	38	Access Drive
2Ø	Read Sequential	39	Free Drive
		4Ø	Write Random With Zero Fill

65536 records of 128 bytes each, numbered from Ø through 65535, thus allowing a maximum of 8 megabytes per file. Note, however, that although the records may be considered logically contiguous, they are not necessarily physically contiguous in the disk data area. Internally, all files are broken into 16K byte segments called logical extents, so that counters are easily maintained as 8-bit values. Although the decomposition into extents is discussed in the paragraphs which follow, they are of no particular consequence to the programmer since each extent is automatically accessed in both sequential and random access modes.

In the file operations starting with function number 15, DE usually addresses a file control block (FCB). Transient programs often use the default file control block area reserved by MP/M at location memory segment base +ØØ5CH for simple file operations. The basic unit of file information is a 128 byte record used for all file operations, thus a default location for disk I/O is provided by MP/M at location memory segment base +ØØ8ØH which is the initial default DMA address (see function 26). All directory operations take place in a reserved area which does not affect write buffers as was the case in CP/M release 1, with the exception of Search First and Search Next, where compatibility is required.

The File Control Block (FCB) data area consists of a sequence of 33 bytes for sequential access and a series of 36 bytes in the case that the file is accessed randomly. The default file control block normally located at memory segment base +ØØ5CH can be used for random access files, since the three bytes starting at memory segment base +ØØ7DH are available for this purpose.

information for all subsequent file operations. When accessing files, it is the programmer's responsibility to fill the lower sixteen bytes of the FCB and initialize the "cr" field. Normally, bytes 1 through 11 are set to the ASCII character values for the file name and file type, while all other fields are zero.

FCB's are stored in a directory area of the disk, and are brought into central memory before proceeding with file operations (see the OPEN and MAKE functions). The memory copy of the FCB is updated as file operations take place and later recorded permanently on disk at the termination of the file operation (see the CLOSE command).

The CLI constructs the first sixteen bytes of two optional FCB's for a transient by scanning the remainder of the line following the transient name, denoted by "file1" and "file2" in the prototype command line described above, with unspecified fields set to ASCII blanks. The first FCB is constructed at location memory segment base +ØØ5CH, and can be used as-is for subsequent file operations. The second FCB occupies the dØ ... dn portion of the first FCB, and must be moved to another area of memory before use. If, for example, the operator types

PROGNAME B:X.ZOT Y.ZAP

the file PROGNAME.PRL is loaded into a user memory segment or if it is not on the disk, the file PROGNAME.COM is loaded into the TPA, and the default FCB at memory segment base +ØØ5CH is initialized to drive code 2, file name "X" and file type "ZOT". The second drive code takes the default value Ø, which is placed at memory segment base +ØØ6CH, with the file name "Y" placed into location memory segment base +ØØ6DH and file type "ZAP" located 8 bytes later at memory segment base +ØØ75H. All remaining fields through "cr" are set to zero. Note again that it is the programmer's responsibility to move this second file name and type to another area, usually a separate file control block, before opening the file which begins at memory segment base +ØØ5CH, due to the fact that the open operation will overwrite the second name and type.

If no file names are specified in the original command, then the fields beginning at memory segment base +ØØ5DH and +ØØ6DH contain blanks. In all cases, the CLI translates lower case alphabetics to upper case to be consistent with the MP/M file naming conventions.

As an added convenience, the default buffer area at location memory segment base +ØØ8ØH is initialized to the command line tail typed by the operator following the program name. The first position contains the number of characters, with the characters themselves following the character count.

2.2 Basic Disk Operating System Functions

In general, the Basic Disk Operating System (BDOS) facilities are identical to that of CP/M 2.Ø. Each function is covered in this section by describing the entry parameters, returned values, and any differences between CP/M and MP/M.

```
****************************************
*                                      *
*  FUNCTION Ø:  SYSTEM RESET           *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  ØØH              *
****************************************
```

The SYSTEM RESET function terminates the calling program, releasing the memory segment, console, and mutual exclusion messages owned by the calling program. When the console is released it is usually given back to the terminal message process (TMP) for that console.

Effectively the operation of the SYSTEM RESET function is the same for MP/M as it is for CP/M 2.Ø because the program is terminated and the operator receives the prompt to enter another command. However, MP/M does not re-initialize the disk subsystem by selecting and logging-in disk drive A.

```
****************************************
*                                      *
*  FUNCTION 1:. CONSOLE INPUT          *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  Ø1H              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  ASCII Character  *
****************************************
```

The CONSOLE INPUT function reads the next console character to register A. Graphic characters, along with carriage return, line feed, and backspace (ctl-H) are echoed to the console. Tab characters (ctl-I) are expanded in columns of eight characters. A check is made for start/stop scroll (ctl-S) and start/stop printer echo (ctl-P). The BDOS does not return to the calling program until a character has been typed, thus suspending execution if a character is not ready.

```
**************************************
*                                    *
*  FUNCTION 4:  RAW CONSOLE OUTPUT   *
*                                    *
**************************************
*  Entry Parameters:                 *
*      Register   C:  Ø4H            *
*      Register   E:  ASCII Character *
*                                    *
**************************************
```

The RAW CONSOLE OUTPUT function sends the ASCII character from register E to the console device. There is no testing of the output character, that is, tabs are not expanded and no checks are made for start/stop scroll and printer echo. This function does not require that the console be attached, nor does it attach the console. Thus, unsolicited messages may be sent to other consoles by simply changing the console byte of the process descriptor and then using this function.

The PUNCH OUTPUT function is not supported under MP/M.

```
**************************************
*                                    *
*  FUNCTION 5:  LIST OUTPUT          *
*                                    *
**************************************
*  Entry Parameters:                 *
*      Register   C:  Ø5H            *
*      Register   E:  ASCII Character *
*                                    *
**************************************
```

The LIST OUTPUT function sends the ASCII character in register E to the logical listing device.

Caution must be observed in the use of the printer since there is no implicit list device ownership. That is, the list device is not "opened" or "closed". MP/M affords a secondary explicit means to resolve printer mutual exclusion. A queue named 'MXList' is created by the system to handle mutual exclusion. To properly obtain use of the printer a program should open the 'MXList' queue and read the message. When the message is obtained the printer may be used. When printing is completed the message should be written back to the 'MXList' queue. This technique is used by the MP/M PIP, SPOOLer, and TMP ctl-P operations.

```
****************************************
*                                      *
*  FUNCTION 7:  GET I/O BYTE           *
*                                      *
****************************************
*                                      *
*    Not supported under MP/M          *
*                                      *
****************************************
```

The GET I/O BYTE function is not supported under MP/M.

```
****************************************
*                                      *
*  FUNCTION 8:  SET I/O BYTE           *
*                                      *
****************************************
*                                      *
*    Not supported under MP/M          *
*                                      *
****************************************
```

The SET I/O BYTE function is not supported under MP/M.

```
****************************************
*                                      *
*  FUNCTION 9:  PRINT STRING           *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  Ø9H              *
*      Registers DE:  String Address   *
*                                      *
****************************************
```

The PRINT STRING function sends the character string stored in memory at the location given by DE to the console device, until a "$" is encountered in the string. Tabs are expanded as in function 2, and checks are made for start/stop scroll and printer echo.

```
****************************************
*                                      *
*  FUNCTION 11: GET CONSOLE STATUS     *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  ØBH              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Console Status   *
****************************************
```

The CONSOLE STATUS function checks to see if a character has been typed at the console. If a character is ready, the value ØFFH is returned in register A. Otherwise a ØØH value is returned.

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

Function 12 provides information which allows version independent programming. A two-byte value is returned, with H = ØØ designating the CP/M release (H = Ø1 for MP/M), and L = ØØ for all releases previous to 2.Ø. CP/M 2.Ø returns a hexadecimal 2Ø in register L, with subsequent version 2 releases in the hexadecimal range 21, 22, through 2F. Using function 12, for example, you can write application programs which provide both sequential and random access functions, with random access disabled when operating under early releases of CP/M.

```
**************************************
*                                    *
*  FUNCTION 15: OPEN FILE            *
*                                    *
**************************************
*  Entry Parameters:                 *
*      Register   C:  ØFH            *
*      Registers DE:  FCB Address    *
*                                    *
*  Returned   Value:                 *
*      Register   A:  Directory Code *
**************************************
```

The OPEN FILE operation is used to activate a file which currently exists in the disk directory for either the currently active user code or user code Ø. The BDOS scans the referenced disk directory for a match in positions 1 through 14 of the FCB referenced by DE (byte s1 is automatically zeroed), where an ASCII question mark (3FH) matches any directory character in any of these positions. Normally, no question marks are included and, further, bytes "ex" and "s2" of the FCB are zero.

If a directory element is matched, the relevant directory information is copied into bytes dØ through dn of the FCB, thus allowing access to the files through subsequent read and write operations. Note that an existing file must not be accessed until a sucessful open operation is completed. Upon return, the open function returns a "directory code" with the value Ø through 3 if the open was successful, or ØFFH (255 decimal) if the file cannot be found. If question marks occur in the FCB then the first matching FCB is activated. Note that the current record ("cr") must be zeroed by the program if the file is to be accessed sequentially from the first record.

The open file operation will succeed for files with either the current user code or user code Ø. This presents a problem when files with the same name exist under both the current user code and under user code Ø. When such a situation exists the first one found in the directory will be opened. Even though this should not present a problem because user code Ø is intended only for system and commonly used files, a potential problem can be detected by using the search file function. The search file function enables examination of the directory FCB and thus the actual file user code can be determined.

Opening a file sets the appropriate bit in the drive active vector of the calling processes process descriptor. This bit is cleared only by terminating the process or making a free drive (function 39) call. Setting of the bit in the drive active vector will prevent any other process from resetting the drive on which the file was opened.

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.

An ASCII question mark (63 decimal, 3F hexadecimal) in any position from "f1" through "ex" matches the corresponding field of any directory entry on the default or auto-selected disk drive. If the "dr" field contains an ASCII question mark, then the auto disk select function is disabled, the default disk is searched, with the search function returning any matched entry, allocated or free, belonging to any user number. This latter function is not normally used by application programs, but does allow complete flexibility to scan all current directory values. If the "dr" field is not a question mark, the "s2" byte is automatically zeroed.

To determine the user code of a successful search (it may be the currently active user code or user code Ø), the returned directory code can be used as described above to index into the DMA buffer and the user code of the directory FCB can be obtained.

```
****************************************
*                                      *
*  FUNCTION 18: SEARCH FOR NEXT        *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  12H              *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The SEARCH NEXT function is similar to the Search First function, except that the directory scan continues from the last matched entry. Similar to function 17, function 18 returns the decimal value 255 in A when no more directory items match.

```
****************************************
*                                      *
*  FUNCTION 21: WRITE SEQUENTIAL       *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  15H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

Given that the FCB addressed by DE has been activated through an open or make function (numbers 15 and 22), the WRITE SEQUENTIAL function writes the 128 byte data record at the current DMA address to the file named by the FCB. the record is placed at position "cr" of the file, and the "cr" field is automatically incremented to the next record position. If the "cr" field overflows then the next logical extent is automatically opened and the "cr" field is reset to zero in preparation for the next write operation. Write operations can take place into an existing file, in which case newly written records overlay those which already exist in the file. Register A = ØØH upon return from a successful write operation, while a non-zero value indicates a full disk.

```
****************************************
*                                      *
*  FUNCTION 22: MAKE FILE              *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  16H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The MAKE FILE operation is similar to the open file operation except that the FCB must name a file which does not exist in the currently referenced disk directory (i.e., the one named explicitly by a non-zero "dr" code, or the default disk if "dr" is zero). The FDOS creates the file and initializes both the directory and main memory value to an empty file. The programmer must ensure that no duplicate file names occur, and a preceding delete operation is sufficient if there is any possibility of duplication. Upon return, register A = Ø, 1, 2, or 3 if the operation was successful and ØFFH (255 decimal) if no more directory space is available. The make function has the side-effect of activating the FCB and thus a subsequent open is

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 Ø through 15 corresponding to drives A through P.

```
****************************************
*                                      *
*  FUNCTION 26: SET DMA ADDRESS        *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1AH              *
*      Registers DE:  DMA Address      *
*                                      *
****************************************
```

"DMA" is an acronym for Direct Memory Address, which is often used in connection with disk controllers which directly access the memory of the mainframe computer to transfer data to and from the disk subsystem. Although many computer systems use non-DMA access (i.e., the data is transfered through programmed I/O operations), the DMA address has, in MP/M, come to mean the address at which the 128 byte data record resides before a disk write and after a disk read. Upon cold start, warm start, or disk system reset, the DMA address is automatically set to BOOT+ØØ8ØH. The Set DMA function, however, can be used to change this default value to address another area of memory where the data records reside. Thus, the DMA address becomes the value specified by DE until it is changed by a subsequent Set DMA function, cold start, warm start, or disk system reset.

```
****************************************
*                                      *
*  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 which indicates drives which have the temporary read/only bit set. Similar to function 24, the least significant bit corresponds to drive A, while the most significant bit corresponds to drive P. The R/O bit is set either by an explicit call to function 28, or by the automatic software mechanisms within MP/M which detect changed disks.

```
****************************************
*                                      *
*  FUNCTION 3Ø: SET FILE ATTRIBUTES    *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  1EH              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Directory Code   *
****************************************
```

The SET FILE ATTRIBUTES function allows programmatic manipulation of permanent indicators attached to files. In particular, the R/O, System, and Update attributes (t1', t2', and t3') can be set or reset. The DE pair addresses an unambiguous file name with the appropriate attributes set or reset. Function 3Ø searches for a match, and changes the matched directory entry to contain the selected indicators. Indicators f1' through f4' are not presently used, but may be useful for applications programs, since they are not involved in the matching process during file open and close operations. Indicators f5' through f8' are reserved for future system expansion.

```
****************************************
*                                      *
*  FUNCTION 33: READ RANDOM            *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  21H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Return Code      *
****************************************
```

The READ RANDOM function is similar to the sequential file read operation of previous releases, except that the read operation takes place at a particular record number, selected by the 24-bit value constructed from the three byte field following the FCB (byte positions rØ at 33, rl at 34, and r2 at 35). Note that the sequence of 24 bits is stored with least significant byte first (rØ), middle byte next (rl), and high byte last (r2). MP/M does not reference byte r2, except in computing the size of a file (function 35). Byte r2 must be zero, however, since a non-zero value indicates overflow past the end of file.

Thus, the rØ,rl byte pair is treated as a double-byte, or "word" value, which contains the record to read. This value ranges from Ø to 65535, providing access to any particular record of the 8 megabyte file. In order to process a file using random access, the base extent (extent Ø) must first be opened. Although the base extent may or may not contain any allocated data, this ensures that the file is properly recorded in the directory, and is visible in DIR requests. The selected record number is then stored into the random record field (rØ,rl), and the BDOS is called to read the record. Upon return from the call, register A either contains an error code, as listed below, or the value ØØ indicating the operation was successful. In the latter case, the current DMA address contains the randomly accessed record. Note that contrary to the sequential read operation, the record number is not advanced. Thus, subsequent random read operations continue to read the same record.

Upon each random read operation, the logical extent and current record values are automatically set. Thus, the file can be sequentially read or written, starting from the current randomly accessed position. Note, however, that in this case, the last randomly read record will be re-read as you switch from random mode to sequential read, and the last record will be re-written as you switch to a sequential write operation. You can, of course, simply advance the random record position following each random read or write to obtain the effect of a sequential I/O operation.

Error codes returned in register A following a random read

the random read operation with the addition of error code Ø5, which indicates that a new extent cannot be created due to directory overflow.

```
****************************************
*                                      *
*  FUNCTION 35: COMPUTE FILE SIZE      *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  23H              *
*      Registers DE:  FCB Address      *
*                                      *
*  Returned   Value:                   *
*      Random Record Field Set         *
****************************************
```

When computing the size of a file, the DE register pair addresses an FCB in random mode format (bytes rØ, rl, and r2 are present). The FCB contains an unambiguous file name which is used in the directory scan. Upon return, the random record bytes contain the "virtual" file size which is, in effect, the record address of the record following the end of the file. if, following a call to function 35, the high record byte r2 is Ø1, then the file contains the maximum record count 65536. Otherwise, bytes rØ and rl constitute a 16-bit value (rØ is the least significant byte, as before) which is the file size.

Data can be appended to the end of an existing file by simply 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.

The virtual size of a file corresponds to the physical size when the file is written sequentially. If, instead, the file was created in random mode and "holes" exist in the allocation, then the file may in fact contain fewer records than the size indicates. If, for example, only the last record of an eight megabyte file is written in random mode (i.e., record number 65535), then the virtual size is 65536 records, although only one block of data is actually allocated.

```
****************************************
*                                      *
*  FUNCTION 37:  RESET DRIVE           *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  25H              *
*      Register  DE:  Drive Vector     *
*                                      *
*  Returned   Value:                   *
*      Register   A:  Return Code      *
****************************************
```

The RESET DRIVE function allows resetting of specified drive(s). The passed parameter is a 16 bit vector of drives to be reset, the least significant bit is drive A:. If there are any open files on a specified drive, the reset drive is denied and the reason is displayed on the console.

The returned value indicates whether or not the reset drive was successful. If any process is currently accessing a drive to be reset, an error code of ØFFH is returned in the A register. A return code of Ø indicates success.

```
****************************************
*                                      *
*  FUNCTION 38:  ACCESS DRIVE          *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  26H              *
*      Register  DE:  Drive Vector     *
*                                      *
****************************************
```

The ACCESS DRIVE function allows setting the drive access bit(s) in the calling processes process descriptor. The passed parameter is a 16 bit vector of drive(s) to be accessed, the least significant bit is drive A:.

2.3 Queue and Process Descriptor Data Structures

This section contains a description of the queue and process descriptor data structures used by the MP/M Extended Disk Operating System (XDOS).

QUEUE DATA STRUCTURES

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

MP/M has been designed to simplify queue management for both user and system processes. In fact, queues are treated in a manner similar to disk files. Queues can be created, opened, written to, read from, and deleted.

A few illustrations should suffice to describe applications for queues:

COMMUNICATION:

A queue can be used for communication to provide a FIFO list of messages produced by a producer for consumption by a consumer. For example, consider a data logging application where data is continuously received via a serial communication link and is to be written to a disk file. This would be a difficult application for a sequential operating system such as CP/M because arriving serial data would be lost while buffers were being written to disk. Under MP/M a queue could be used by the producer to send blocks of received serial data (or simply buffer pointers) to a consumer which would write the blocks on disk. MP/M supports concurrency of these operations, allowing the producer to quickly write a buffer to the queue and then resume monitoring the serial input.

SYNCHRONIZATION:

When a process attempts to read a message at a queue and there are no messages posted at the queue, the process is placed in a priority ordered list of processes waiting for messages at the queue. The process will remain in that state until a message arrives. Thus synchronization of processes can be achieved, allowing the waiting (DQing) process to continue execution when a message is sent to the queue.

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 as follows:

```
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, DQ process head,
          set by system
NQPH    = 2 bytes, NQ process head,
          set by system
MSG$IN  = 2 bytes, pointer to next
          message in, set by system
MSG$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 two byte buffer for the owner process
           descriptor address.
```

Queue Overhead = 24 bytes

LINKED QUEUES

The following example illustrates how to setup a queue control block for a linked queue containing 4 messages, each 33 bytes in length:

```
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
```

USER QUEUE CONTROL BLOCK

The user queue control block data structure is used to provide read/write access to queues in much the same manner that a file control block provides access to a disk file. Queues are "opened", an operation which fills in the actual queue control block address, and then can be read from or written to.

If the actual queue address is known it can be filled in the pointer field of the user queue control block, the 8 byte name field can be omitted, and an open operation is not required in order to access the queue.

The following example illustrates a user queue control block:

PL/M:

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

DECLARE BUFFER (33) BYTE;
```

Assembly Language:

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

BUFFER:
        DS      33      ; BUFFER
```

PROCESS DESCRIPTOR

Each process in the MP/M system has a process descriptor which defines all the characteristics of the process. The following example illustrates the process descriptor:

PL/M:

```
DECLARE CNS$HNDLR STRUCTURE (
  PL ADDRESS,
  STATUS BYTE,
  PRIORITY BYTE,
  STKPTR ADDRESS,
  NAME (8) BYTE,
  CONSOLE BYTE,
  MEMSEG BYTE,
  B ADDRESS,
  THREAD ADDRESS,
  DISK$SET$DMA ADDRESS,
  DISK$SLCT BYTE,
  DCNT ADDRESS,
  SEARCHL BYTE,
  SEARCHA ADDRESS,
  DRVACT ADDRESS,
  REGISTERS (20) BYTE,
  SCRATCH (2) BYTE )
  INITIAL (0,0,200,.CNS$STK(19),
           'CNS     ',1,0FFH);

DECLARE CNS$STK (20) ADDRESS INITIAL (
  0C7C7H,0C7C7H,0C7C7H,0C7C7H,0C7C7H,0C7C7H,
  0C7C7H,0C7C7H,0C7C7H,0C7C7H,0C7C7H,0C7C7H,
  0C7C7H,0C7C7H,0C7C7H,0C7C7H,0C7C7H,0C7C7H,
  0C7C7H,STRT$CNS);
```

CONSOLE	= 1 byte, console to be used by process, set by user
MEMSEG	= 1 byte, memory segment table index
B	= 2 bytes, system scatch area
THREAD	= 2 bytes, process list thread, set by system
DISKSETDMA	= 2 bytes, default DMA address, set by user
DISK$SLCT	= 1 byte, default disk/user code
DCNT	= 2 bytes, system scratch byte
SEARCHL	= 1 byte, system scratch byte
SEARCHA	= 2 bytes, system scratch bytes
DRVACT	= 2 bytes, 16 bit vector of drives being accessed by the process
REGISTERS	=20 bytes, 8080 / Z80 register save area
SCRATCH	= 2 bytes, system scratch bytes

PROCESS NAMING CONVENTIONS

The following conventions should be used in the naming of processes. Processes which wait on queues that are to be sent command tails from the TMPs are given the console resource if their name matches that of the queue which they are reading. Processes which are to be protected from abortion by an operator using the ABORT command must have at least one lower case character in the process name.

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


****************************************
*                                      *
*  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 program. The single passed parameter is the address of a memory descriptor. The only memory descriptor parameter filled in by the calling program is the size, the other parameters, base, attributes and bank, are filled in by XDOS.

The operation returns a boolean indicating whether or not the memory request could be satisifed. A returned value of FFH indicates failure to satisfy the request and a value of Ø indicates success.

Note that base and size specify base page address and page size where a page is 256 bytes. (See function 128: ABSOLUTE MEMORY REQUEST for a description of the memory descriptor data structure.)

```
****************************************
*                                      *
*  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 processor 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.

The operation returns a boolean 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 Ø indicates success.

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

The operation returns a boolean indicating whether or not a successful FLAG SET was performed. A returned value of FFH indicates that a flag over-run has occurred, i.e. the flag was already set when a flag set function was called. A returned value of Ø indicates success.

```
****************************************
*                                      *
*  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 queue control block address into the user queue control block. The result of this function is that a user program can obtain access to queues by knowing only the queue name, the actual address of the queue itself is obtained as a result of opening the queue. Once a queue has been opened, the queue may be read from or written to using the queue read and write operations.

The function returns a boolean indicating whether or not the open queue operation found the queue to be opened. A returned value of ØFFH indicates failure while a zero indicates success.

The user queue control block data structure is described in section 2.3.

```
****************************************
*                                      *
*  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. A single parameter is passed to delete a queue, the address of the actual queue.

The function returns a boolean indicating whether or not the delete queue operation deleted the queue. A returned value of ØFFH indicates failure, usually because some process is DQing from the queue. A returned value of Ø indicates success.

```
**************************************
*                                    *
*  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 specified queue. If no buffers are available at the queue, the calling process relinquishes the processor until a buffer is available at the queue. The single passed parameter is the address of a user queue control block. When a buffer is available at the queue, the buffer pointed to by the MSGADR field of the user queue control block is copied into the actual queue.

```
**************************************
*                                    *
*  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 a specified queue if a buffer is available. The single passed parameter is the address of a user queue control block. If a buffer is available at the queue, the buffer pointed to by the MSGADR field of the user queue control block is copied into the actual queue.

The operation returns a boolean indicating whether or not a buffer was available at the queue. A returned value of ØFFH indicates no buffer while a zero indicates that a buffer was available and that the user buffer was copied into it.

```
****************************************
*                                      *
*  FUNCTION 143:  TERMINATE PROCESS    *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  8FH              *
*                 D:  Conditional      *
*                      Memory Free     *
*                 E:  Terminate Code   *
*                                      *
****************************************
```

The TERMINATE PROCESS function terminates the calling process. The passed parameters indicate whether or not the process should be terminated if it is a system process and if the memory segment is to be released. A ØFFH 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 a user process is being terminated and Register D is a ØFFH, the memory segment is not released. Thus a process which is a child of a parent process both executing in the same memory segment can terminate without freeing the memory segment which is also occupied by the parent.

There are no results returned from this operation, the calling process simply ceases to exist as far as MP/M is concerned.

```
****************************************
*                                      *
*  FUNCTION 144:  CREATE PROCESS       *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  9ØH              *
*                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 ready list.

A single parameter is passed, the address of a process descriptor. The first field of the process descriptor is a link field which may point to another process descriptor.

Processes can only be created either in common memory or by

```
****************************************
*                                      *
*  FUNCTION 147:  DETACH CONSOLE       *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  93H              *
*                                      *
****************************************
```

The DETACH CONSOLE function detaches the console specified in the CONSOLE field of the process descriptor from the calling process. If the console is not currently attached no action takes place.

There are no passed parameters and there are no returned results.

```
****************************************
*                                      *
*  FUNCTION 148:  SET CONSOLE          *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  94H              *
*                 E:  Console          *
*                                      *
****************************************
```

The SET CONSOLE function detaches the currently attached console and then attaches the console specified as a calling parameter. If the console to be attached is already attached to another process descriptor, the calling process relinquishes the processor until the console is available.

A single passed parameter contains the console number to be attached. There are no returned results.

```
****************************************
*                                      *
*  FUNCTION 150:  SEND CLI COMMAND     *
*                                      *
****************************************
*  Entry Parameters:                   *
*      Register   C:  96H              *
*                 DE:  CLICMD Address  *
*                                      *
****************************************
```

The SEND CLI COMMAND function permits running programs to send command lines to the Command Line Interpreter. A single parameter is passed which is the address of a data structure containing the default disk/user code, console and command line itself (shown below).

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 program being executed. The ASCII command line begins with the third byte and is terminated with a null byte.

There are no results returned to the calling program.

The following example illustrates the SEND CLI COMMAND data structure:

```
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
```

```
****************************************
*                                      *
*  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 prepares a file control block from an input ASCII string containing a file name terminated by a null or a carriage return. The parameter is the address of a data structure (shown below) which contains the address of the ASCII file name string followed by the address of the target file control block.

The operation returns an FFFFH if the input ASCII string contains an invalid file name. A zero is returned if the ASCII string contains a single valid file name, otherwise the address of the first character following the file name is returned.

The following example illustrates the parse file name control block data structure:

PL/M:

```
Declare Parse$FN$CB structure (
  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
        DS      36
```

```
****************************************
*                                      *
*  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 obtains the current encoded date and time. A single passed parameter is the address of a data structure (shown below) which is to contain the date and time. The date is represented as a 16-bit integer with day 1 corresponding to January 1, 1978. The time is respresented as three bytes: hours, minutes and seconds, stored as two BCD digits.

The following example illustrates the TOD data structure:

```
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 calling processes process descriptor. By definition this is the head of the ready list.

2.5 Preparation of Page Relocatable Programs

A page relocatable program is stored on diskette as a file of type 'PRL'. Appendix K contains a PRL file specification describing the file format. A page relocatable program is prepared by assembling the source program twice, in which the second assembly has 100H added to each ORG statement. The two hex files generated by assembling the source file twice are concatenated with PIP and then provided as input to the GENMOD program. The GENMOD program (described in section 1.4) produces a file of type 'PRL'.

This section describes APPENDIX G: Sample Page Relocatable Program. The example program illustrates the required use of ORG statements to access the BDOS and the default file control block. Note that the initial ORG is 0000H. Its purpose is to establish the equate for the symbol BASE, the base of the relocatable segment. Next an 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. Note that the first assembly will generate a file which can be LOADed to produce an executable 'COM' file. In fact, it is desirable to first debug the program as a 'COM' file and then proceed to make the 'PRL' file.

It is VERY important to use BASE to offset all memory segment base page references! Do not make a call to absolute 0005H for BDOS calls. In this example BASE is used to offset the BDOS, FCB, and BUFF equates. When a user program needs to determine the top of its memory segment the following equate and code sequence should 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 Macro Assembler (MAC):

* Prepare the user program, DUMP.ASM in this example, with proper origin statements as described above.

* Assuming a system disk in drive A: and the DUMP.ASM file is on drive B:, enter the commands-

```
1A>MAC B:DUMP $PP+S
      ;assemble and list the DUMP.ASM file
1A>ERA B:DUMP.HX0
```

2.6 Installation of Resident System Processes

This section contains a description of APPENDIX H: Sample Resident System Process. The example program illustrates the required structure of a resident system process as well as the BDOS/XDOS access mechanism.

The first two bytes of a resident system process are set to the address of the BDOS/XDOS entry point. The address is filled in by the loader, providing a simple means for a resident system process to 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 resident system process must immediately follow the first two bytes which contain the address of the BDOS/XDOS entry point. Observe the manner in which the process descriptor is initialized in the example. The DS's are used where storage is simply allocated. The DB's and DW's are used where data in the process descriptor must be initialized. Note that the stack pointer field of the process descriptor points to the address immediately following the stack allocation. This is the return address which is the actual process entry point.

It is important that the HEX file generated by assembling the RSP span the entire program and data area. For this reason the first two bytes of the resident system process which will contain the address of the BDOS/XDOS entry point are defined with a DW. Using a DS would not generate any HEX file code for those two bytes. The end of the program and data area must be defined in a likewise manner. If your RSP has DS statements preceding the END statement it will be necessary to place a DB statment after the DS statements before the END statement.

The steps to produce a resident system process closely follow those illustrated in the previous section on page relocatable programs. The only exception to the procedure is that the GENMOD output file should have a type of 'RSP' rather than 'PRL' and the code in the RSP is ORGed at ØØØH rather than 1ØØH.

In addition to resident system processes MP/M supports resident system procedures. The purpose of a resident system procedure is to provide a means to use a piece of code as a serially reusable resource. A resident system procedure is set up by a resident system process. The function of the process is to create a queue which has the name of the resident system procedure and to send it one 16 bit message containing the address of the resident system procedure. Once this is accomplished the resident system process terminates itself. Access to the resident system procedure is made by opening the queue with the resident system procedure name and then reading the two byte message to obtain the actual memory address of the

3. MP/M ALTERATION GUIDE

3.1 Introduction

The standard MP/M system assumes operation on an Intel MDS-800 microcomputer development system, but is designed so that the user can alter a specific set of subroutines which define the hardware operating environment. In this way, the user can produce a diskette which operates with any IBM-3741 format compatible diskette subsystem and other peripheral devices.

Although standard MP/M is configured for single density floppy disks, field-alteration features allow adaptation to a wide variety of disk subsystems from single drive minidisks through high-capacity "hard disk" systems.

In order to achieve device independence, MP/M is distinctly separated into an XIOS module which is hardware environment dependent and several other modules which are not dependent upon the hardware configuration.

The user can rewrite the distribution version of the MP/M XIOS to provide a new XIOS which provides a customized interface between the remaining MP/M modules and the user's own hardware system. The user can also rewrite the distribution version of the LDRBIOS which is used to load the MP/M system from disk.

The purpose of this section is to provide the following step-by-step procedure for writing both your LDRBIOS and new XIOS for MP/M:

(1) Implement CP/M 2.0 on the target computer

To simplify the MP/M adaptation process, we assume (and STRONGLY recommend) that CP/M 2.0 has already been implemented on the target MP/M machine. If this is not the case it will be necessary for the user to implement the CP/M 2.0 BIOS as described in the Digital Research document titled "CP/M 2.0 Alteration Guide" in addition to the MP/M XIOS. The reason that both the BIOS and XIOS have to be implemented is that the MP/M loader uses the CP/M 2.0 BIOS to load and relocate MP/M. Once loaded, MP/M uses the XIOS and not the BIOS. The CP/M 2.0 BIOS used by the MP/M loader is called the LDRBIOS.

Another good reason for implementing CP/M 2.0 on the target MP/M machine is that debugging your XIOS is greatly simplified by bringing up MP/M while running SID or DDT under a CP/M 2.0 system.

E.) Write the updated memory image onto a disk file using the CP/M 'SAVE' command. The 'X' placed in front of the file name is used simply to designate an experimental version, preserving the orginal.

```
A>SAVE 26 XMPMLDR.COM
```

F.) Test XMPMLDR.COM and then rename it to MPMLDR.COM.

(3) Prepare your custom XIOS

If MP/M is being tailored to your computer system for the first time, the new XIOS requires some relatively simple software development and testing. The standard XIOS is listed in APPENDIX I, and can be used as a model for the customized package.

The XIOS entry points, including both basic and extended, are described in sections 3.2 and 3.3. These sections along with APPENDIX I provides you with the necessary information to write your XIOS. We suggest that your initial implementation of an XIOS utilize polled I/O without any interrupts. The system will run without even a clock interrupt. The origin of your XIOS should be ØØØØH. Note the two equates needed to access the dispatcher and XDOS from the XIOS:

```
        ORG     ØØØØH
PDISP   EQU     $-3
XDOS    EQU     PDISP-3
```

The procedure to prepare an XIOS.SPR file from your customized XIOS is as follows:

A.) Assemble your XIOS.ASM and then rename the XIOS.HEX file to XIOS.HXØ.

B.) Assemble your XIOS.ASM again specifying the +R option which offsets the ORG statements by 1ØØH bytes. Or, edit your XIOS.ASM and change the initial ORG ØØØH to an ORG 1ØØH and assemble it again.

C.) Use PIP to concatenate your two HEX files:

```
A>PIP XIOS.HEX=XIOS.HXØ,XIOS.HEX
```

D.) Run the GENMOD program to produce the XIOS.SPR file from the concatenated HEX files.

```
A>GENMOD XIOS.HEX XIOS.SPR
```

```
...
Breakpoint RST #  = 7
...
```

C.) If a resident system process is being debugged make certain that it is selected for inclusion in MPM.SYS.

D.) Using CP/M 1.4 or 2.Ø, load the MPMLDR.COM file into memory.

```
A>DDT MPMLDR.COM
DDT VERS 2.Ø
NEXT  PC
1AØØ ØlØØ
```

E.) Place a 'B' character into the second position of default FCB. This operation can be done with the 'I' command:

```
-IB
```

F.) Execute the MPMLDR.COM program by entering a 'G' command:

```
-G
```

G.) At point the MP/M loader will load the MP/M operating system into memory, displaying a memory map.

H.) If you are debugging an XIOS, note the address of the XIOS.SPR memory segment. If you are debugging a resident system process, note the address of the resident system process. This address is the relative ØØØØH address of the code being debugged. You must also note the address of SYSTEM.DAT.

I.) Using the 'S' command, set the byte at SYSTEM.DAT + 2 to the restart number which you want the MP/M debugger to use. Do not select the same restart as that being used by the CP/M debugger.

```
...
Memory Segment Table:
SYSTEM  DAT  D6ØØH  ØlØØH
...

-SD6Ø2
D6Ø2 Ø7 Ø5
```

J.) Using the 'X' command, determine the MP/M beginning execution address. The address is the first location past the current program counter.

```
-X
```

Either the SID or DDT debugger can be used in place of writing a GETSYS program as is shown in the following example which also uses SYSGEN in place of PUTSYS. Sample skeletal GETSYS and PUTSYS programs are described later in this section (for a more detailed description of GETSYS and PUTSYS see the "CP/M 2.Ø Alteration Guide").

In order to make the MP/M system load and run automatically, the user must also supply a cold start loader, similar to the one described in the "CP/M 2.Ø Alteration Guide". The purpose of the cold start loader is to load the MP/M loader into memory from the first two tracks of the diskette. The CP/M 2.Ø cold start loader must be modified in the following manner: the load address must be changed to Ø1ØØH and the execution address must also be changed to Ø1ØØH.

The following techniques are specifically for the MDS-8ØØ which has a boot ROM that loads the first track into location 3ØØØH. However, the steps shown can be applied in general to any hardware.

If a SYSGEN program is available, the following steps can be used to prepare a diskette that cold starts MP/M:

A.) Prepare the MPMLDR.COM file by integrating your custom LDRBIOS as described earlier in this section. Test the MPMLDR.COM and verify that it operates properly.

B.) Execute either DDT or SID.

```
A>DDT
DDT VERS 2.Ø
```

C.) Using the input command ('I') specify that the MPMLDR.COM file is to be read in and then read ('R') in the file with an offset of 88ØH bytes.

```
-IMPMLDR.COM
-R88Ø
NEXT  PC
248Ø Ø1ØØ
```

D.) Using the 'I' command specify that the BOOT.HEX file is to be read in and then read in the file with an offset that will load the boot into memory at 9ØØH. The 'H' command can be used to calculate the offset.

```
-H9ØØ 3ØØØ
39ØØ D9ØØ

-IBOOT.HEX
-RD9ØØ
NEXT  PC
```

```
;   PUTSYS PROGRAM - WRITE TRACKS Ø AND 1 FROM MEMORY AT 3380H
;   REGISTER                USE
;      A              (SCRATCH REGISTER)
;      B              TRACK COUNT (Ø, 1)
;      C              SECTOR COUNT (1,2,...,26)
;      DE             (SCRATCH REGISTER PAIR)
;      HL             LOAD ADDRESS
;      SP             SET TO STACK ADDRESS
;
START: LXI   SP,3380H     ;SET STACK POINTER TO SCRATCH AREA
       LXI   H, 3380H     ;SET BASE LOAD ADDRESS
       MVI   B, Ø         ;START WITH TRACK Ø
WRTRK:                    ;WRITE NEXT TRACK (INITIALLY Ø)
       MVI   C,1          ;WRITE STARTING WITH SECTOR 1
WRSEC:                    ;WRITE NEXT SECTOR
       CALL  WRITESEC      ;USER-SUPPLIED SUBROUTINE
       LXI   D,128        ;MOVE LOAD ADDRESS TO NEXT 1/2 PAGE
       DAD   D            ;HL = HL + 128
       INR   C            ;SECTOR = SECTOR + 1
       MOV   A,C          ;CHECK FOR END OF TRACK
       CPI   27
       JC    WRSEC        ;CARRY GENERATED IF SECTOR < 27
;
;   ARRIVE HERE AT END OF TRACK, MOVE TO NEXT TRACK
       INR   B
       MOV   A,B          ;TEST FOR LAST TRACK
       CPI   2
       JC    WRTRK        ;CARRY GENERATED IF TRACK < 2
;
;   ARRIVE HERE AT END OF LOAD, HALT FOR NOW
       HLT
;
;   USER-SUPPLIED SUBROUTINE TO WRITE THE DISK
WRITESEC:
;   ENTER WITH TRACK NUMBER IN REGISTER B,
;         SECTOR NUMBER IN REGISTER C, AND
;         ADDRESS TO FILL IN HL
;
       PUSH   B           ;SAVE B AND C REGISTERS
       PUSH   H           ;SAVE HL REGISTERS
       ..........................................
       perform disk write at this point, branch to

       label START if an error occurs
       ..........................................
       POP    H           ;RECOVER HL
       POP    B           ;RECOVER B AND C REGISTERS
       RET                ;BACK TO MAIN PROGRAM

       END    START
```

Track#	Sector#	Page#	Memory Address (boot address)	MP/M Cold	Module name Start Loader
00	01				
00	02	00	0100H		MPMLDR
"	03	"	0180H		"
"	04	01	0200H		"
"	05	"	0280H		"
"	06	02	0300H		"
"	07	"	0380H		"
"	08	03	0400H		"
"	09	"	0480H		"
"	10	04	0500H		"
"	11	"	0580H		"
"	12	05	0600H		"
"	13	"	0680H		"
"	14	06	0700H		"
"	15	"	0780H		"
"	16	07	0800H		"
"	17	"	0880H		"
"	18	08	0900H		"
"	19	"	0980H		"
"	20	09	0A00H		"
"	21	"	0A80H		"
"	22	10	0B00H		"
"	23	"	0B80H		"
"	24	11	0C00H		"
00	25	"	0C80H		MPMLDR
00	26	12	0D00H		LDRBDOS
01	01	"	0D80H		"
"	02	13	0E00H		"
"	03	"	0E80H		"
"	04	14	0F00H		"
"	05	"	0F80H		"
"	06	15	1000H		"
"	07	"	1080H		"
"	08	16	1100H		"
"	09	"	1180H		"
"	10	17	1200H		"
"	11	"	1280H		"
"	12	18	1300H		"
"	13	"	1380H		"
"	14	19	1400H		"
"	15	"	1480H		"
"	16	20	1500H		"
"	17	"	1580H		"
"	18	21	1600H		"
01	19	"	1680H		LDRBDOS
01	20	22	1700H		LDRBIOS
"	21	"	1780H		"
"	22	23	1800H		"
"	23	"	1880H		"
"	24	24	1900H		"
"	25	"	1980H		"
01	26	25	1A00H		LDRBIOS

and CONOUT subroutines (LIST and LSTST may be used by PIP, but not the BDOS).

The characteristics of each device are

CONSOLE — The principal interactive consoles which communicate with the operators, accessed through CONST, CONIN, and CONOUT. Typically, CONSOLEs are devices such as CRTs or Teletypes.

LIST — The principal listing device, if it exists on your system, which is usually a hard-copy device, such as a printer or Teletype.

DISK — Disk I/O is always performed through a sequence of calls on the various disk access subroutines which set up the disk number to access, the track and sector on a particular disk, and the direct memory access (DMA) address involved in the I/O operation. After all these parameters have been set up, a call is made to the READ or WRITE function to perform the actual I/O operation. Note that there is often a single call to SELDSK to select a disk drive, followed by a number of read or write operations to the selected disk before selecting another drive for subsequent operations. Similarly, there may be a single call to set the DMA address, followed by several calls which read or write from the selected DMA address before the DMA address is changed. The track and sector subroutines are always called before the READ or WRITE operations are performed.

Note that the READ and WRITE routines should perform several retries (10 is standard) before reporting the error condition to the BDOS. If the error condition is returned to the BDOS, it will report the error to the user. The HOME subroutine may or may not actually perform the track 00 seek, depending upon your controller characteristics; the important point is that track 00 has been selected for the next operation, and is often treated in exactly the same manner as SETTRK with a parameter of 00.

The exact responsibilities of each entry point subroutine are given below:

BOOT — The BOOT entry point gets called from the MP/M loader after it has been loaded by the cold start

SELDSK — Select the disk drive given by register C for further operations, where register C contains Ø for drive A, 1 for drive B, and so-forth up to 15 for drive P (the standard MP/M distribution version supports four drives). On each disk select, SELDSK must return in HL the base address of a 16-byte area, called the Disk Parameter Header, described in the digital research document titled "CP/M 2.Ø Alteration Guide". For standard floppy disk drives, the contents of the header and associated tables does not change, and thus the program segment included in the sample XIOS performs this operation automatically. If there is an attempt to select a non-existent drive, SELDSK returns HL=ØØØØH as an error indicator.

On entry to SELDSK it is possible to determine whether it is the first time the specified disk has been selected. Register E, bit Ø (least significant bit) is a zero if the drive has not been previously selected. This information is of interest in systems which read configuration information from the disk in order to set up a dynamic disk definition table.

Although SELDSK must return the header address on each call, it is advisable to postpone the actual physical disk select operation until an I/O function (seek, read or write) is actually performed, since disk selects often occur without utimately performing any disk I/O, and many controllers will unload the head of the current disk before selecting the new drive. This would cause an excessive amount of noise and disk wear.

SETTRK — Register BC contains the track number for subsequent disk accesses on the currently selected drive. You can choose to seek the selected track at this time, or delay the seek until the next read or write actually occurs. Register BC can take on values in the range Ø-76 corresponding to valid track numbers for standard floppy disk drives, and Ø-65535 for non-standard disk subsystems.

SETSEC — Register BC contains the sector number (1 through 26) for subsequent disk accesses on the currently selected drive. You can choose to send this information to the controller at this point, or instead delay sector selection until a read or write operation occurs.

SECTRAN — Performs sector logical to physical sector translation in order to improve the overall response of MP/M. Standard MP/M systems are shipped with a "skew factor" of 6, where six physical sectors are skipped between each logical read operation. This skew factor allows enough time between sectors for most programs to load their buffers without missing the next sector. In particular computer systems which use fast processors, memory, and disk subsystems, the skew factor may be changed to improve overall response. Note, however, that you should maintain a single density IBM compatible version of MP/M for information transfer into and out of your computer system, using a skew factor of 6. In general, SECTRAN receives a logical sector number in BC, and a translate table address in DE. The sector number is used as an index into the translate table, with the resulting physical sector number in HL. For standard systems, the tables and indexing code is provided in the XIOS and need not be changed.

ticks of the system time unit, the start clock procedure is called.

The purpose of the STARTCLOCK procedure is to eliminate unneccessary system clock interrupt overhead when there are not any delayed processes.

In some hardware environments it is not acutally possible to shut off the system time unit clock while still maintaining the one second flag used for the purposes of keeping time of day. In this situation the STARTCLOCK procedure simply sets a boolean variable to true, indicating that there is a delayed process. The clock interrupt handler can then determine if system time unit flag is to be set by testing the boolean.

STOPCLOCK When the system delay list is emptied the stop clock procedure is called.

The purpose of the STOPCLOCK procedure is to eliminate unneccessary system clock interrupt overhead when there are no delayed processes.

In some hardware environments it is not acutally possible to shut off the system time unit clock while still maintaining the one second flag used for the purposes of keeping time of day. (i.e. a single clock/timer interrupt source is used.) In this situation the STOPCLOCK procedure simply sets a boolean variable to false, indicating that there are no delayed processes. The clock interrupt handler can then determine if the system time unit flag is to be set by testing the boolean.

EXITREGION The purpose of the exit region procedure is to test a preempted flag, set by the interrupt handler, enabling interrupts if preempted is false. This procedure allows interrupt service routines to make MP/M system calls, leaving interrupts disabled until completion of the interrupt handling.

MAXCONSOLE The purpose of the maximum console procedure is to enable the calling program to determine the number of physical consoles which the BIOS is capable of supporting. The number of physical consoles is returned in the A register.

SYSTEMINIT The purpose of the system initialization

INTERRPUT SERVICE ROUTINES

The MP/M operating system is designed to work with virtually any interrupt architecture, be it flat or vectored.. The function of the code operating at the interrupt level is to save the required registers, determine the cause of the interrupt, remove the interrupting condition, and to set an appropriate flag. Operation of the flags are described in section 2.4. Briefly, flags are used to synchronize asynchronous processes. One process, such as an interrupt service routine, sets a particular flag while another process waits for the flag to be set.

At a logical level above the physical interrupts the flags can be regarded as providing 256 levels of virtual interrupts (32 flags are supported under release 1 of MP/M). Thus, logical interrupt handlers wait on flags to be set by the physical interrupt handlers. This mechanism allows a common XDOS to operate on all microcomputers, regardless of the hardware environment.

As an example consider a hardware environment with a flat interrupt structure. That is, a single interrupt level is provided and devices must be polled to determine the cause of the interrupt. Once the interrupt cause is determined a specific flag is set indicating that that particular interrupt has occurred.

At the conclusion of the interrupt processing a jump should be made to the MP/M dispatcher. This is done by jumping to the PDISP entry point. The effect of this jump is to give the processor to the highest priority ready process, usually the process readied by setting the flag in the interrupt handler, and then to enable interrupts before jumping to resume execution of the process.

The only XDOS or BDOS call which should be made from an interrupt handler is FUNCTION 133: FLAG SET. Any other XDOS or BDOS call will result in a dispatch which would then enable interrupts prior to completing execution of the interrupt handler.

It is recommended that interrupts only be used for operations which are asynchronous, such as console input or disk operation complete. In general, operations such as console output should not be interrupt driven. The reason that interrupts are not desirable for console output is that the system is afforded some elasticity by performing polled console outputs while idling, rather than incurring the dispatch overhead for each character transmitted. This is particularily true at higher baud rates.

3.4 System File Components

The MP/M system file, 'MPM.SYS' consists of five components: the system data page, the customized XIOS, the BDOS or ODOS, the XDOS, and the resident system processes. MPM.SYS resides in the directory with a user code of Ø and is usually read only. The MP/M loader reads and relocates the MPM.SYS file to bring up the MP/M system.

SYSTEM DATA

The system data page contains 256 bytes used by the loader to dynamically configure the system. The system data page can be prepared using the GENSYS program or it can be manually prepared using DDT or SID. The following table describes the byte assignments:

Byte	Assignment
ØØØ-ØØØ	Top page of memory
ØØ1-ØØ1	Number of consoles
ØØ2-ØØ2	Breakpoint restart number
ØØ3-ØØ3	Allocate stacks for user system calls, boolean
ØØ4-ØØ4	Bank switched memory, boolean
ØØ5-ØØ5	Z8Ø CPU, boolean
ØØ6-ØØ6	Banked BDOS file manager, boolean
ØØ7-Ø15	Unassigned, reserved
Ø16-Ø47	Initial memory segment table
Ø48-Ø79	Breakpoint vector table, filled in by DDTs
Ø8Ø-111	Stack addresses for user system calls
112-122	Scratch area for memory segments
123-127	Unassigned, reserved
128-143	Submit flags
144-255	Reserved

CUSTOMIZED XIOS

The customized XIOS is obtained from a file named 'XIOS.SPR'. The 'XIOS.SPR' file is actually a file of type PRL containing the page relocatable version of the user customized XIOS. A submit file on the distribution diskette named 'MACSPR.SUB' or 'ASMSPR.SUB' can be used to generate the user customized XIOS. The following sequence of commands will produce a 'XIOS.SPR' file given a user 'XIOS.ASM' file:

* The process descriptor for the resident system process must begin at the third byte position. The contents of the process descriptor are described in section 2.3.

BNKBDOS

In addition to the MPM.SYS file a file named 'BNKBDOS.SPR' is used in systems with a banked BDOS. It is a page relocatable file containing the non-resident portion of the banked BDOS. This file is not used by systems without banked memory.

Breakpoint RST #: The breakpoint restart number to be used by the SID and DDT debuggers is specified. Restart Ø is not allowed. Other restarts required by the XIOS should also not be used.

Add system call user stacks (Y/N)?: If you desire to execute CP/M *.COM files then your response should be Y. A 'Y' response forces a stack switch with each system call from a user program. MP/M requires more stack space than CP/M.

Bank switched memory (Y/N)?: If your system does not have bank switched memory then you should respond with a 'N'. Otherwise respond with a 'Y' and additional questions and responses (as shown in the second example) will be required.

Memory segment bases: Memory segmentation is defined by the entries which are made. Care must be taken in the entry of memory bases as all entries must be made with successively higher bases. If your system has ROM at ØØØØH then the first memory segment base which you specify should be your first actual RAM location. Only page relocatable (PRL) programs can be run in systems that do not have RAM at location ØØØØH.

Select Resident System Processes: A directory search is made for all files of type RSP. Each file found is listed and included in the generated system file if you respond with a 'Y'.

The second example illustrates a more complicated GENSYS in which a system is setup with bank switched memory and a banked BDOS. This procedure requires an intial GENSYS and MPMLDR execution to determine the exact size of the operating system, followed by a second GENSYS.

```
A>GENSYS

MP/M System Generation
======================

Top page of memory = ff
Number of consoles = 2
Breakpoint RST #   = 6
Add system call user stacks (Y/N)? y
Z8Ø CPU (Y/N) y
Bank switched memory (Y/N)? y
Banked BDOS file manager (Y/N)? y
Enter memory segment table: (ff terminates list)
 Base,size,attrib,bank = Ø,5Ø,Ø,Ø
 Base,size,attrib,bank = ff
Select Resident System Processes: (Y/N)
ABORT    ? n
SPOOL    ? n
```

```
XIOS     SPR  F600H  0600H
BDOS     SPR  EE00H  0800H
XDOS     SPR  CF00H  1F00H
Sched    RSP  CA00H  0500H
BNKBDOS  SPR  BC00H  0E00H
-------------------------
Memseg   Usr  0000H  5000H  Bank 00H
```

Using the information obtained from the initial GENSYS and MPMLDR execution the following GENSYS can be executed:

```
A>GENSYS

MP/M System Generation
======================

Top page of memory = ff
Number of consoles = 2
Breakpoint RST #   = 6
Add system call user stacks (Y/N)? y
Z80 CPU (Y/N)? y
Bank switched memory (Y/N)? y
Banked BDOS file manager (Y/N)? y
Enter memory segment table: (ff terminates list)
 Base,size,attrib,bank = 0,bc,0,0
 Base,size,attrib,bank = 0,c0,0,1
 Base,size,attrib,bank = 0,c0,0,2
 Base,size,attrib,bank = ff
Select Resident System Processes: (Y/N)
ABORT    ? n
SPOOL    ? n
MPMSTAT  ? n
SCHED    ? y
```

In the following example the 'MPM.SYS' file prepared by the second GENSYS example shown in section 3.5 is loaded:

```
A>MPMLDR

MP/M Loader
===========

Number of consoles = 2
Breakpoint RST #   = 6
Z8Ø CPU
Banked BDOS file manager
Top of memory      =  FFFFH

Memory Segment Table:
SYSTEM  DAT  FFØØH  Ø1ØØH
CONSOLE DAT  FDØØH  Ø2ØØH
USERSYS STK  FCØØH  Ø1ØØH
XIOS    SPR  F6ØØH  Ø6ØØH
BDOS    SPR  EEØØH  Ø8ØØH
XDOS    SPR  CFØØH  1FØØH
Sched   RSP  CAØØH  Ø5ØØH
BNKBDOS SPR  BCØØH  ØEØØH
-------------------------
Memseg  Usr  ØØØØH  CØØØH  Bank Ø2H
Memseg  Usr  ØØØØH  CØØØH  Bank Ø1H
Memseg  Usr  ØØØØH  BCØØH  Bank ØØH

MP/M
ØA>
```

APPENDIX B: Process Priority Assignments

Ø - 31	: Interrupt handlers
32 - 63	: System processes
64 - 197	: Undefined
198	: Teminal message processes
199	: Command line interpreter
2ØØ	: Default user priority
2Ø1 - 254	: User processes
255	: Idle process

APPENDIX D: XDOS Function Summary

FUNC	FUNCTION NAME	INPUT PARAMETERS	OUTPUT RESULTS
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 Que	DE = .UQCB	A = err code
139	Write Queue	DE = .UQCB	none
140	Conditional Write Que	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 Pr	DE = .CPB	HL = result
152	Parse Filename	DE = .PFCB	see def
153	Get Console Number	none	A = console #
154	System Data Address	none	HL = sys data adr
155	Get Date and Time	DE = .TOD	none
156	Return Proc. Dsc. Adr	none	HL = proc descr adr
157	Abort Spec. Process	DE = .ABTPB	A = err code

Appendix F: Operation of MP/M on the Intel MDS-800

This section gives operating procedures for using MP/M on the Intel MDS microcomputer development system. A basic knowledge of the MDS hardware and software systems is assumed.

MP/M is initiated in essentially the same manner as Intel's ISIS operating system. The disk drives labelled 0 through 3 on the MDS, correspond to MP/M drives A through D, respectively. The MP/M system diskette is inserted into drive 0, and the BOOT and RESET switches are depressed in sequence. The interrupt 2 light should go on at this point. The space bar is then depressed on either console device, and the light should go out. The BOOT switch is then turned off, and the MP/M sign-on message should appear at both consoles, followed by the "0A>" for the CRT or "1A>" for the TTY. The user can then issue MP/M commands.

Use of the interrupt switches on the front panel is not recommended. Effective 'warm-starts' should be initiated at the console by aborting the running program rather than pushing the INT 0 switch. Also, depending on the choice of restart for the debugger the INT switch which will invoke the debugger is not necessarily #7.

Diskettes should not be removed from the drives until the user verifies that there are no other users with open files on the disk. This can be done with the 'DSKRESET' command.

When performing GENSYS operations on the MDS-800, make certain that a negative response is always made to the Z80 CPU question. Responding with a 'Y' will lead to unpredictable results.

```
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,0
0103 39                 dad     sp
                ;       entry stack pointer in hl from the ccp
0104 221f02             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
010d feff               cpi     255     ;255 if file not present
010f c21b01             jnz     openok  ;skip if open is ok
                ;
                ;       file not there, give error message and return
0112 11fd01             lxi     d,opnmsg
0115 cda101             call    err
0118 c35601             jmp     finis   ;to return
                ;
                openok: ;open operation ok, set buffer index to end
011b 3e80               mvi     a,80h
011d 321d02             sta     ibp     ;set buffer pointer to 80h
                ;       hl contains next address to print
0120 210000             lxi     h,0     ;start with 0000
                ;
                gloop:
0123 e5                 push    h       ;save line position
0124 cda701             call    gnb
0127 e1                 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     0fh     ;check low 4 bits
012f c24401             jnz     nonum
                ;       print line number
0132 cd7701             call    crlf
                ;
                ;       check for break key
0135 cd5e01             call    break
                ;       accum lsb = 1 if character ready
0138 0f                 rrc             ;into carry
0139 da5101             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
```

```
018e c637       pl0:    adi     'a' - 10
0190 cd6a01     prn:    call    pchar
0193 c9                 ret
                ;
                phex:   ;print hex char in reg a
0194 f5                 push    psw
0195 0f                 rrc
0196 0f                 rrc
0197 0f                 rrc
0198 0f                 rrc
0199 cd8201             call    pnib    ;print nibble
019c f1                 pop     psw
019d cd8201             call    pnib
01a0 c9                 ret
                ;
                err:    ;print error message
                ;       d,e addresses message ending with "$"
01a1 0e09               mvi     c,printf        ;print buffer function
01a3 cd0500             call    bdos
01a6 c9                 ret
                ;
                ;
                gnb:    ;get next byte
01a7 3a1d02             lda     ibp
01aa fe80               cpi     80h
01ac c2b801             jnz     g0
                ;       read another buffer
                ;
                ;
01af cdd301             call    diskr
01b2 b7                 ora     a       ;zero value if read ok
01b3 cab801             jz      g0      ;for another byte
                ;       end of data, return with carry set for eof
01b6 37                 stc
01b7 c9                 ret
                ;
                g0:     ;read the byte at buff+reg a
01b8 5f                 mov     e,a     ;ls byte of buffer index
01b9 1600               mvi     d,0     ;double precision index to de
01bb 3c                 inr     a       ;index=index+1
01bc 321d02             sta     ibp     ;back to memory
                ;       pointer is incremented
                ;       save the current file address
01bf 218000             lxi     h,buff
01c2 19                 dad     d
                ;       absolute character address is in hl
01c3 7e                 mov     a,m
                ;       byte is in the accumulator
01c4 b7                 ora     a       ;reset carry bit
01c5 c9                 ret
                ;
                setup:  ;set up file
                ;       open the file for input
```

APPENDIX H: 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     1ah             ; 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 0a                 db      10              ; priority (initial)
0006 1001               dw      stack+38        ; stack pointer
0008 5459504520         db      'type    '      ; name in upper case
```

```
                        ;
                        ; type stack & other local data structures
                        ;
                        stack:
00ea                            ds      38              ; 20 level stack
0110 ba01                       dw      type            ; process entry point

0112                    fcb:    ds      36              ; file control block

0136                    buff:   ds      128             ; file buffer

                        ;
                        ; bdos call procedure
                        ;
                        bdos:
01b6 2a0000                     lhld    bdosadr         ; hl = bdos address
01b9 e9                         pchl

                        ;
                        ; type main program
                        ;
                        type:
01ba 0e86                       mvi     c,mkque
01bc 113600                     lxi     d,typelqcb
01bf cdb601                     call    bdos            ; make typelqcb
01c2 0e91                       mvi     c,stprior
01c4 11c800                     lxi     d,200
01c7 cdb601                     call    bdos            ; set priority to 200

                        forever:
01ca 0e89                       mvi     c,rdque
01cc 119800                     lxi     d,typeuserqcb
01cf cdb601                     call    bdos            ; read from type queue
01d2 0e98                       mvi     c,parsefn
01d4 11e600                     lxi     d,pcb
01d7 cdb601                     call    bdos            ; parse the file name
01da 23                         inx     h
01db 7c                         mov     a,h
01dc b5                         ora     l               ; test for 0ffffh
01dd ca1f02                     jz      error
01e0 3a9d00                     lda     console
01e3 321000                     sta     pdconsole       ; typepd.console = con

01e6 0e0f                       mvi     c,openf
01e8 111201                     lxi     d,fcb
01eb cdb601                     call    bdos            ; open file
01ee 3c                         inr     a               ; test return code
01ef ca1f02                     jz      error           ; if it was 0ffh, no f
01f2 af                         xra     a               ; else,
01f3 323201                     sta     fcb+32          ;   set next record to
                        new$sector:
01f6 0e14                       mvi     c,readf
01f8 111201                     lxi     d,fcb
```

APPENDIX I: Sample XIOS

```
**********************************************
*  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
                ;
                ; note: this module assumes that an org statement will
                ;   provided by concatenating either base0000.asm or b
                ;   to the front of this file before assembling.
                ;
                ;       title   'xios for the mds-800'

                ;       (four drive single density version)
                ;                       -or-
                ;       (four drive mixed double/single density)

                ;       version 1.1 january, 1980

                ;       copyright (c) 1979, 1980
                ;       digital research
                ;       box 579, pacific grove
                ;       california, 93950

0000 =          false   equ     0
ffff =          true    equ     not false

ffff =          asm     equ     true
0000 =          mac     equ     not asm

ffff =          sgl     equ     true
0000 =          dbl     equ     not sgl

                        if      mac
                        maclib  diskdef
                        endif

0004 =          numdisks equ    4       ;number of drives available

                ;       external jump table (below xios base)
fffd =          pdisp   equ     $-3
```

```
                coldstart:
                warmstart:
004b 0e00               mvi     c,0             ; see system init
                                                ; cold & warm start in
                                                ; for compatibility wi
004d c3faff             jmp     xdos            ; system reset, termin

                ;  mp/m 1.0   console handlers

0002 =          nmbcns  equ     2       ; number of consoles

0083 =          poll    equ     131     ; xdos poll function

0000 =          pllpt   equ     0       ; poll printer
0001 =          pldsk   equ     1       ; poll disk
0002 =          plco0   equ     2       ; poll console out #0 (crt:)
0003 =          plcol   equ     3       ; poll console out #1 (tty:)
0004 =          plci0   equ     4       ; poll console in #0 (crt:)
0005 =          plcil   equ     5       ; poll console in #1 (tty:)

                ;
                const:                  ; console status
0050 cd6500             call    ptbljmp ; compute and jump to hndlr
0053 7900               dw      pt0st   ; console #0 status routine
0055 c900               dw      ptlst   ; console #1 (tty:) status rt

                conin:                  ; console input
0057 cd6500             call    ptbljmp ; compute and jump to hndlr
005a 8100               dw      pt0in   ; console #0 input
005c d100               dw      ptlin   ; console #1 (tty:) input

                conout:                 ; console output
005e cd6500             call    ptbljmp ; compute and jump to hndlr
0061 8d00               dw      pt0out  ; console #0 output
0063 dd00               dw      ptlout  ; console #1 (tty:) output

                ;
                ptbljmp:                ; compute and jump to handler
                                        ; d = console #
                                        ; do not destroy <d>
0065 7a                 mov     a,d
0066 fe02               cpi     nmbcns
0068 da6e00             jc      tbljmp
006b f1                 pop     psw     ; throw away table address
                rtnempty:
006c af                 xra     a
006d c9                 ret
                tbljmp:                 ; compute and jump to handler
                                        ; a = table index
006e 87                 add     a       ; double table index for adr o
006f e1                 pop     h       ; return adr points to jump tb
0070 5f                 mov     e,a
0071 1600               mvi     d,0
```

```
0099 79                 mov     a,c
009a d3f6               out     data0           ; transmit character
009c c9                 ret
                ;
                ; wait for console #0 output ready
                ;
                pt0wait:
009d 0e83               mvi     c,poll
009f 1e02               mvi     e,plco0
00a1 c3faff             jmp     xdos            ; poll console #0 outp
                ;       ret

                ;
                ; poll console #0 output
                ;
                polco0:
                                        ; return 0ffh if ready,
                                        ;        000h if not
00a4 dbf7               in      sts0
00a6 e601               ani     01h
00a8 c8                 rz
00a9 3eff               mvi     a,0ffh
00ab c9                 ret
                ;
                ;
                ; line printer driver:
                ;
                list:                   ; list output
00ac dbfb               in      lptsts
00ae e601               ani     01h
00b0 c2bc00             jnz     lptrdy
00b3 c5                 push    b
00b4 0e83               mvi     c, poll
00b6 1e00               mvi     e, pllpt
00b8 cdfaff             call    xdos
00bb c1                 pop     b
                lptrdy:
00bc 79                 mov     a,c
00bd 2f                 cma
00be d3fa               out     lptport
00c0 c9                 ret
                ;
                ; poll printer output
                ;
                pollpt:
                                        ; return 0ffh if ready,
                                        ;        000h if not
00c1 dbfb               in      lptsts
00c3 e601               ani     01h
00c5 c8                 rz
00c6 3eff               mvi     a,0ffh
00c8 c9                 ret
                ;
```

```
00f8 c8                 rz
00f9 3eff               mvi     a,0ffh
00fb c9                 ret
                ;
                ;
                ;  mp/m 1.0   extended i/o system
                ;
                ;
0006 =          nmbdev  equ     6       ; number of devices in poll tb

                polldevice:
                                        ; reg c = device # to be polle
                                        ; return 0ffh if ready,
                                        ;        00h if not
00fc 79                 mov     a,c
00fd fe06               cpi     nmbdev
00ff da0401             jc      devok
0102 3e06               mvi     a,nmbdev; if dev # >= nmbdev,
                                        ; set to nmbdev
                devok:
0104 cd6e00             call    tbljmp  ; jump to dev poll code

0107 c100               dw      pollpt  ; poll printer output
0109 7d02               dw      poldsk  ; poll disk ready
010b a400               dw      polco0  ; poll console #0 output
010d f400               dw      polcol  ; poll console #1 (tty:) outpu
010f 7900               dw      polci0  ; poll console #0 input
0111 c900               dw      polcil  ; poll console #1 (tty:) input
0113 6c00               dw      rtnempty; bad device handler


                ; select / protect memory

                selmemory:
                                        ; reg bc = adr of mem descript
                                        ; bc ->  base   1 byte,
                                        ;        size   1 byte,
                                        ;        attrib 1 byte,
                                        ;        bank   1 byte.

                ; this hardware does not have memory protection or
                ;  bank switching

0115 c9                 ret


                ; start clock

                startclock:
                                        ; will cause flag #1 to be set
                                        ;  at each system time unit ti
0116 3eff               mvi     a,0ffh
0118 32e301             sta     tickn
```

```
                                                ;  of idle must be use
                                                ;  without interrupts,

                ;       -or-

                ;       ei                      ; simply halt until aw
                ;       hlt                     ;  interrupt
                ;       ret


                ;  mp/m 1.Ø   interrupt handlers

0085 =          flagset equ     133
008e =          dsptch  equ     142

                intlhnd:
                                        ; interrupt 1 handler entry po
                                        ;
                                        ;  location 0008h contains a j
                                        ;  to intlhnd.
0145 f5                 push    psw
0146 3e02               mvi     a,2h
0148 d3ff               out     rtc     ; reset real time clock
014a d3fd               out     revrt   ; revert intr cntlr
014c 3aab01             lda     slice
014f 3d                 dcr     a       ; only service every 16th slic
0150 32ab01             sta     slice
0153 ca5901             jz      tl6ms   ; jump if 16ms elapsed
0156 f1                 pop     psw
0157 fb                 ei
0158 c9                 ret

                tl6ms:
0159 3el0               mvi     a,16
015b 32ab01             sta     slice   ; reset slice counter
015e f1                 pop     psw
015f 22dd01             shld    svdhl
0162 e1                 pop     h
0163 22e101             shld    svdret
0166 f5                 push    psw
0167 210000             lxi     h,0
016a 39                 dad     sp
016b 22df01             shld    svdsp           ; save users stk ptr
016e 31dd01             lxi     sp,intstk+48    ; lcl stk for intr hnd
0171 d5                 push    d
0172 c5                 push    b

0173 3eff               mvi     a,0ffh
0175 32e401             sta     preemp  ; set preempted flag

0178 3ae301             lda     tickn
017b b7                 ora     a               ; test tickn, indicate
                                                ;  delayed process(es)
```

```
                seldsk: ;select disk given by register c
01e5 210000             lxi     h, 0
01e8 79                 mov     a,c
01e9 fe04               cpi     numdisks
01eb d0                 rnc             ; first, insure good select
01ec e602               ani     2
01ee 32ba02             sta     dbank   ; then save it
01f1 21c202             lxi     h,sel$table
01f4 0600               mvi     b,0
01f6 09                 dad     b
01f7 7e                 mov     a,m
01f8 32bc02             sta     iof
01fb 60                 mov     h,b
01fc 69                 mov     l,c
01fd 29                 dad     h
01fe 29                 dad     h
01ff 29                 dad     h
0200 29                 dad     h       ; times 16
0201 11c602             lxi     d,dpbase
0204 19                 dad     d
0205 c9                 ret

                home:   ;move to home position
                ;       treat as track 00 seek
0206 0e00               mvi     c,0
                ;
                settrk: ;set track address given by c
0208 21be02             lxi     h,iot
020b 71                 mov     m,c
020c c9                 ret
                ;
                setsec: ;set sector number given by c
020d 79                 mov     a,c     ;sector number to accum
020e 32bf02             sta     ios     ;store sector number to iopb
0211 c9                 ret
                ;
                setdma: ;set dma address given by regs b,c
0212 69                 mov     l,c
0213 60                 mov     h,b
0214 22c002             shld    iod
0217 c9                 ret

                sect$tran:              ; translate the sector # in <c
0218 60                 mov     h,b
0219 69                 mov     l,c
021a 23                 inx     h  ; in case of no translation
021b 7a                 mov     a, d
021c b3                 ora     e
021d c8                 rz
021e eb                 xchg
021f 09                 dad     b       ; point to physical sector
0220 6e                 mov     l,m
0221 2600               mvi     h,0
```

```
                ;       check io completion ok
0263 cd9302             call    intype          ;must be io complete (
                ;       00 unlinked i/o complete,    01 linked i/o com
                ;       10 disk status changed       11 (not used)
0266 fe02               cpi     10b             ;ready status change?
0268 ca8602             jz      wready

                ;       must be 00 in the accumulator
026b b7                 ora     a
026c c28c02             jnz     werror          ;some other condition,

                ;       check i/o error bits
026f cda002             call    inbyte
0272 17                 ral
0273 da8602             jc      wready          ;unit not ready
0276 1f                 rar
0277 e6fe               ani     11111110b       ;any other errors?  (d
0279 c28c02             jnz     werror

                ;       read or write is ok, accumulator contains zero
027c c9                 ret

                poldsk:
027d cdad02             call    instat                  ; get current
0280 e604               ani     iordy                   ; operation co
0282 c8                 rz                              ; not done
0283 3eff               mvi     a,0ffh                  ; done flag
0285 c9                 ret                             ; to xdos


                wready: ;not ready, treat as error for now
0286 cda002             call    inbyte          ;clear result byte
0289 c38c02             jmp     trycount

                werror: ;return hardware malfunction (crc, track, seek
                ;       the mds controller has returned a bit in each
                ;       of the accumulator, corresponding to the condi
                ;       0       - deleted data (accepted as ok above)
                ;       1       - crc error
                ;       2       - seek error
                ;       3       - address error (hardware malfunction)
                ;       4       - data over/under flow (hardware malfu
                ;       5       - write protect (treated as not ready)
                ;       6       - write error (hardware malfunction)
                ;       7       - not ready
                ;       (accumulator bits are numbered 7 6 5 4 3 2 1 0

                trycount:
                ;       register c contains retry count, decrement 'ti
028c 0d                 dcr     c
028d c23c02             jnz     rewait  ;for another try

                ;       cannot recover from error
```

```
                        disks   numdisks                ; generate dri
                        diskdef 0,1,26,6,1024,243,64,64,2
                        diskdef 1,0
                        diskdef 2,0
                        diskdef 3,0
                        endef
                        endif

                        if      mac and dbl
                        disks   numdisks                ; generate dri
                        diskdef 0,1,52,,2048,243,128,128,2,0
                        diskdef 1,0
                        diskdef 2,1,26,6,1024,243,64,64,2
                        diskdef 3,2
                        endef
                        endif

                        if      asm
02c6 =          dpbase  equ     $               ;base of disk param bl
02c6 15030000   dpe0:   dw      xlt0,0000h      ;translate table
02ca 00000000           dw      0000h,0000h     ;scratch area
02ce 2f030603           dw      dirbuf,dpb0     ;dir buff, parm block
02d2 ce03af03           dw      csv0,alv0       ;check, alloc vectors
02d6 15030000   dpe1:   dw      xlt1,0000h      ;translate table
02da 00000000           dw      0000h,0000h     ;scratch area
02de 2f030603           dw      dirbuf,dpb1     ;dir buff, parm block
02e2 fd03de03           dw      csv1,alv1       ;check, alloc vectors
02e6 15030000   dpe2:   dw      xlt2,0000h      ;translate table
02ea 00000000           dw      0000h,0000h     ;scratch area
02ee 2f030603           dw      dirbuf,dpb2     ;dir buff, parm block
02f2 2c040d04           dw      csv2,alv2       ;check, alloc vectors
02f6 15030000   dpe3:   dw      xlt3,0000h      ;translate table
02fa 00000000           dw      0000h,0000h     ;scratch area
02fe 2f030603           dw      dirbuf,dpb3     ;dir buff, parm block
0302 5b043c04           dw      csv3,alv3       ;check, alloc vectors
0306 =          dpb0    equ     $               ;disk param block
                        endif

                        if      asm and dbl
                        dw      52              ;sec per track
                        db      4               ;block shift
                        db      15              ;block mask
                        db      0               ;extnt mask
                        dw      242             ;disk size-1
                        dw      127             ;directory max
                        db      192             ;alloc0
                        db      0               ;alloc1
                        dw      32              ;check size
                        dw      2               ;offset
                xlt0    equ     0               ;translate table
                dpb1    equ     dpb0
                xlt1    equ     xlt0
                dpb2    equ     $
```

```
0306 =          dpbl    equ     dpb0
0315 =          xlt1    equ     xlt0
0306 =          dpb2    equ     dpb0
0315 =          xlt2    equ     xlt0
0306 =          dpb3    equ     dpb0
0315 =          xlt3    equ     xlt0
                        endif

                        if      asm and dbl
                dpb3    equ     dpb2
                xlt3    equ     xlt2
                        endif

                        if      asm
032f =          begdat  equ     $
032f            dirbuf: ds      128             ;directory access buff
                        endif

                        if      asm and sgl
03af            alv0:   ds      31
03ce            csv0:   ds      16
03de            alv1:   ds      31
03fd            csv1:   ds      16
                        endif

                        if      asm and dbl
                alv0:   ds      31
                csv0:   ds      32
                alv1:   ds      31
                csv1:   ds      32
                        endif

                        if      asm
040d            alv2:   ds      31
042c            csv2:   ds      16
043c            alv3:   ds      31
045b            csv3:   ds      16
046b =          enddat  equ     $
013c =          datsiz  equ     $-begdat
                        endif

046b 00                 db      0       ; this last db is req'd to
                                        ; ensure that the hex file
                                        ; output includes the entire
                                        ; diskdef

046c                    end
```

APPENDIX K: Page Relocatable (PRL) File Specification

Page relocatable files are stored on diskette in the following format:

Address:	Contents:
0001-0002H	Program size
0004-0005H	Minimum buffer requirements (additional memory)
0006-00FFH	Currently unused, reserved for future allocation

0100H + Program size = Start of bit map

The bit map is a string of bits identifying which bytes are to be relocated. There is one bit map byte per 8 bytes of program. The most significant bit (7) of the first byte of the bit map indicates whether or not the first byte of the program is to be relocated. A bit which is on indicates that relocation is required. The next bit, bit(6), of the first byte of the bit map corresponds to the second byte of the program.